Delft University of Technology, Bachelor Seminar of Computer Science and Engineering

Offline Compression of Convolutional Neural Networks on Edge Devices

Simon Tulling[1] , **Lydia Chen**[1] , **Masoud Ghiassi**[1] , **Bart Cox**[1]
[1]TU Delft

Abstract

Edge Devices and Artificial Intelligence are important and ever increasing fields in technology. Yet their combination is lacking because the neural networks used in AI are being made increasingly large and complex while edge devices lack the resources to keep up with these developments. Neural network model compression will allow these edge devices to run these models due to overcoming memory constraints. This paper proposes to use both singular value decomposition and canonical polyadic decomposition as a way to decrease the size of convolutional neural networks at the cost of some accuracy. This compression pipeline can be run on an edge device and is configurable to change the trade-off between file size and accuracy. This creates a possibility to run convolutional neural networks natively on edge devices.

1 Introduction

IoT (Internet of Things) devices and AI (artificial intelligence) are technologies that have been built for decades and are increasing in popularity. These technologies are getting to a point where people are no longer surprised if their fridge is connected to the internet. However, the integration between these two technologies leaves a lot to be desired. Take for instance the smartphone, which is, one of, if not the most used computational device on earth. Yet when we look at the integration of native AI on smartphones, in 2019 26% of all smartphones had AI running natively on the devices[10]. All other smartphones had to offload their AI computations to centralized servers in order to let the more powerful servers do the computations. This is even more apparent for smaller IoT devices such as cameras and wearables where it's not worth it to fit a powerful processor into it in order to run the AI natively.

However, there's currently a big gap between the technology of AI, which is creating larger and larger models that can understand and indentify things in images, and the abilities of edge devices to execute them. Can we find a way to let IoT devices run the neural networks without the need for an internet connection. If this can be done, then a big problem for IoT devices such as longevity and logistics can be solved. Longevity because these devices don't depend on other services such as centralized servers anymore. We shouldn't assume servers that devices need to work stay online forever. Therefor devices will not be made arbitrarily obsolete if they can work without the need of external services. This will also help with logistics since devices will be able to work without an active internet connection so they can be used in places where internet infrastructure is scarce.

The reason why these devices aren't currently running the AI models natively is because they are resource constrained. Wearables and edge devices have memory and processing constraints to a point where it's faster to send the data to a server, run the computations there, and then send it back [9]. There is other research going on within the peer group to improve the speed of the computations on edge devices. Their research is about loading, scheduling and processing the models as fast as possible. A way to complement this research is by making the neural network models smaller to begin with.

The aim for this paper is to explore the optimal method of compressing neural network models to be used for inference in edge devices. We aren't investigating a method to compress a single model for use in edge devices but are looking for a generalized method, a compression pipeline, to compress any given neural network model.

In this paper we present the compression methods and pipeline for compressing models to be used for inference in edge devices. The expected contribution of this research is:

- We develop a compression pipeline to be used in edge devices that compresses models according to the preference of the user.

With preference of the user, we mean that the user can specify if they want more accuracy or a smaller file size. These two are related as in most cases the accuracy of a model decreases when you decrease the file size.

Neural network compression is a well researched field in AI research, however other research have not taken into account the fact that the model will be compressed on the edge device itself and not off the device. This brings several restrictions that we will have to take into account when doing the research. The constraints of the research include:

- Retraining: The results of my research should be a compression pipeline that does not depend on retraining, since this will be inconvenient/impossible on edge devices.

- Resource Constraints: The compression pipeline should work with resource constraints. This is the main reason why there is no retraining.
- Partial Loading: The model resulting from the compression pipeline should be able to be partially loaded.
- No change in architecture: The output model from the compression pipeline needs to be a Caffe model having the same supported layers as the input model.

In order to create this compression pipeline we need to answer the research question and its sub-questions. The research question is:

What is the best way of compressing neural network models, provided the compression has to be done in an environment with resource constraints?

In order to find this method we will inspect multiple compression methods and see if they could be used in our compression pipeline. While there has been a lot of research in the field of neural network compression, most of this research do not take into account the constraints we have specified. Therefore, most methods might look great at first but could be inapplicable because of processing time on an edge device or due to other limitations.

There are some sub-research questions that need to be answered first in order to find the applicable methods for the compression pipeline to answer the main research question.

- What methods are applicable in an environment with memory constraints?
  We're not interested in a method that is unfeasible on an edge device. A method that would take hours to days on an edge device aren't applicable.
- What methods do not change the underlying architecture of the model?
  The output model should work in the same environment as the input model.
- Which methods results in a model that can be partially loaded?

  We want a method that results in a network that can be partially loaded. This way it can be used in other research within the peer group that are focusing on the optimization of partial loading on edge devices.

In the following paragraphs we will go through the different methods for compression and try to apply them to the problem in order to see whether they can help with compressing models on edge devices.

2 Background on Neural Network Compression

We'll start off by clarifying that in neural network models there are two main types of layers that are the most interesting, the fully connected layers and the convolutional layers. This is because those two layers contain most of the weights that are unique to each model and are in turn very large memory-wise. So in my research we are looking for ways to compress these layers in particular.

However the concept of fully connected and convolutional layers may seem foreign at first. But once we translate the layers into their mathematical forms they become more familiar. For instance the fully connected layer, at its core, is a 2D matrix containing the weights for all connections between the nodes. Where the size of one dimension is equal to the amount of input nodes, and the size of the other dimension is equal to the amount of output nodes. In the same way a convolutional layer is a 4D matrix where the dimensions are the dimensions of the individual convolutions and the amount of input and output nodes.

Now that the neural network layers might be a bit more familiar to us we will refer to them as matrices for a lot of the explanations in this paragraph.

Now, in order to answer the research question we'll have to find out different compression techniques and apply them to models and see the results. We'll also need to figure out which compression methods are applicable given the restrictions. Because neural network compression is such a big field, there are quite a lot of methods to consider. The selection of methods that we will consider are:

- Sparse Matrices[11]
- Knowledge distillation[3]
- Quantization[12]
- Singular Value Decomposition[6]
- Canonical Polyadic Decomposition[4]

We will go over each method and show how they work in the context of neural networks and explain whether or not they are applicable for the problem. We will also use a neural network called AgeNet in our examples, this is a network with 3 fully connected and three convolutional layers that is used to classify the age of a person from a photo of their face.

Sparse Matrices

We'll start off with sparse matrices. Since most weights in neural network layers are close to, if not equal to, zero, it might be tempting to only store the weights in the matrix that are significant. This means only storing weight that are not close to zero. However, this will only give a speedup when the amount of weights in the matrix under the significance threshold is more than around 90% [11]. This is unfortunately not the case in the models tested in the paper, and this cannot be lowered without retraining. Apart from that, sparse matrices are also not included in Caffe by default so this violates the "no change in infrastructure" restriction. That is why this method is not used in this paper.

Knowledge distillation

Knowledge distillation is a technique where you train a smaller model based on a larger model in order to create a smaller model with similar accuracy. This technique is really promising in the field of object detection[1]. However we can see immediately that this violates the restriction of retraining so we will not consider this method.

Quantization

The third considered method is Quantization. At it's core this method converts the 32 bit floats of a weight matrix into indices for a list of clustered weights. This turns a weight matrix consisting of 32 bit floats into a weight matrix containing 8 bit indices. This way the size of a single layer is decreased around 4 times. But this method has the same problem as the sparse matrices. These quantization layers are not included in Caffe by default, so this violates the "no change in infrastructure" restriction.

Singular Value Decomposition

This is the first method that's actually used in the compression pipeline. This technique is less reliant on the properties of layer architectures and are more grounded in linear algebra. Singular value decomposition, which we will call SVD going forward, is a technique that splits up a two dimensional matrix into 3 different, smaller, two dimensional matrices. These matrices can then also be interpreted as three different fully connected layers, thus reducing the total amount of memory needed and increasing the amount of total layers for a model, which helps with partial loading. Currently SVD is limited to only working on two dimensional matrices so it can only be used on fully connected layers and it will not work on convolutional layers.

In the explanation of this method, and it's applications for compression, we will use the names of the matrices as defined in figure 1. We will not go into detail on how to calculate these three U, Σ and V^T matrices as it seems out of the scope of this paper, but will instead go into how these matrices can be used in order to compress a fully connected layer.

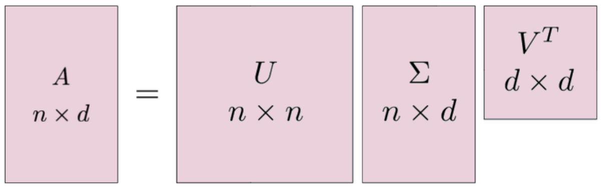

Figure 1: The matrices being created by applying the SVD

These three matrices U, Σ and V^T can then be multiplied together again to create the original matrix A. However, this doesn't look that interesting, we just turned a single matrix to three different matrices where one of the matrices, in this instance matrix U, is even bigger that the original matrix A. The interesting part comes when the matrices are truncated. Figure 2 shows that these three matrices can get truncated by some number r, which we call the rank. What we mean with truncating matrices is that we change the dimensions of the matrices by leaving out a bunch of information. Multiplying these truncated matrices together afterwards can create an approximation of matrix A, in this case called A^*. As the rank r gets bigger, the size of the matrices U, Σ and V^T will increase, however, the similarity of A^* to A will also increase. This can be seen as the trade-off between file size and accuracy when applying this to neural networks.

A big part of the SVD algorithm is the calculation of the optimal rank for the situation. We will explain my own

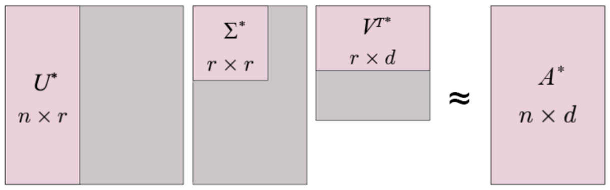

Figure 2: Multiplying truncated matrices

method of calculating the rank for the SVD given the users preference further on in this paper.

In the case of neural network layers we will turn the resulting layers from the SVD into individual layers. However there is a small trick in order to optimize this further. Instead of creating three new fully connected layers to represent the matrices U^*, Σ^* and V^{T*}, we can actually combine the first two matrices U^* and Σ^* into a new layer. Since the formula for dimensions when multiplying two arbitrary matrices of size $(m \times n)$ and $(n \times k)$ is:

$$(m \times n) \cdot (n \times k) = (m \times k)$$

And the matrices U^* and Σ^* with dimensions $(n \times r)$ and $(r \times r)$ gives as a multiplication:

$$(n \times r) \cdot (r \times r) = (n \times r)$$

Which is the same size as U^*, so we can simply multiply U^* by Σ^* and get a resulting matrix which has the same size of U^* while retaining the same information of U^* and Σ^*. This results in the compression saving a whole layer for Σ^*.

So the SVD will actually result in two different fully connected layers, instead of three, in order to save file size.

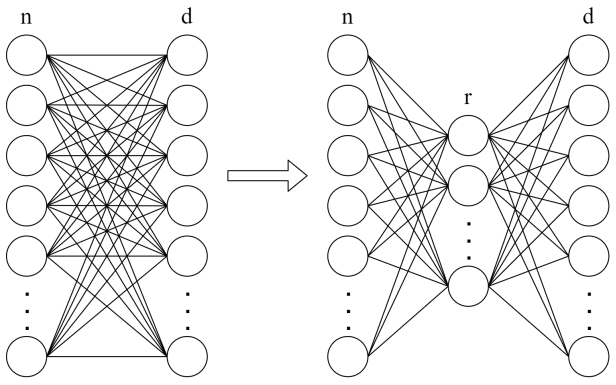

Figure 3: How the SVD compresses individual fully connected layers

Canonical Polyadic Decomposition

Canonical Polyadic Decomposition (CPD), sometimes referred to as CANDECOMP/PARAFAC, is similar to SVD as we are again deconstructing a layer from the neural network, and in this case we are targeting the convolutional layers instead of the fully connected layers. This technique will turn a single convolutional layer into 4 smaller ones. Instead of calculating the decomposed matrices first and then truncating them, CPD needs to know the rank beforehand to calculate the matrices. Another limitation is that the process of calculating these matrices is significantly more computationally expensive than SVD, which is used for fully connected layers. Nevertheless, this seemed like a good way to compress convolutional layers since it does not violate any restrictions.

The main way CPD works is by creating multiple rank-1 4d matrices that try to approximate the original 4d matrix representing the convolutional layer when added together, afterwards it will combine the rank-1 matrices to create the rank-n matrices. This process can be seen in the figure below on a three dimensional matrix, this is done on a three dimensional matrix since it's easier to visualize three dimensional matrices than four dimensional matrices. Nevertheless the outcome is roughly the same.

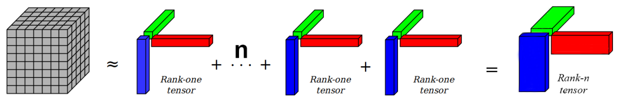

Figure 4: Visualization of CPD with rank n on a three dimensional matrix

The main problem of this method is the way that the rank-1 matrices are calculated. Instead of SVD, where three matrices are calculated and then truncated by a specific rank. CPD given a rank r creates r rank-1 matrices. Afterwards the algorithm tries to optimize these r matrices to match the accuracy of the original as best as possible.

These are the methods considered for the compression on edge devices. In the end we ended up with using singular value decomposition for the fully connected layers and canonical polyadic decomposition for the convolutional layers since these two techniques only apply to those layers. In the upcoming chapters we will look at my contribution when using these techniques in a compression pipeline and their results.

3 Compression Pipeline

Our main contribution is a compression pipeline that can compress models on edge devices. This pipeline gives the user the option to specify the ratio between focusing of accuracy or file size. This ratio goes between a model with minimal ranks which is extremely small but has incredibly low no negligible accuracy and the original model, since you cannot get more accuracy than the original model.

In order for this to work, the pipeline should correctly optimize each layer based on the input ratio and then reconstruct the model afterwards. As stated in the previous section, the SVD and CPD methods rely on rank for their accuracy and file size. So the main objective for the pipeline is to find the optimal rank for the given ratio and apply it to a given layer. It should be noted that these optimizations and rank calculations have to be done separately for each layer in the network as they all will have a different optimal rank.

SVD Rank Optimization

As seen in the previous section, the SVD needs a truncation to be effective otherwise it will only increase the size of the layer instead of compressing it. So we have to figure out a the optimal rank in order to find the best compression of that layer given the ratio from the user. We'll start off by seeing the effect on the accuracy of a network when truncated on different ranks. In this figure we show the validation accuracy of the AgeNet model with different ranks applied to it's largest fully connected layer. The specifics of the model aren't interesting and will be elaborated on in the next section.

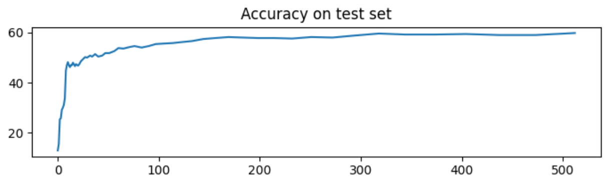

Figure 5: Validation accuracy on the test set using different ranks

This graph in figure 5 is created by validating each model where the biggest fully connected layer is truncated to different ranks with the same test set. The graph is spiking up and down at some random points but the general form can be seen throughout. In general, the accuracy increases as the rank increases. But it should be noted that when the rank increases the model size also increases. So we would want to look for the highest accuracy while keeping the rank as low as possible. it is also clear that the accuracy increases immensely from the start and then flattens out near the end, for a greatly compressed model we would want to take the rank at the point where the flattening starts. In this case it would be around a rank between 20 to 50. However, this way of calculating the accuracy of a rank is computationally expensive and also requires a test set. Since this compression pipeline is used on an edge device without the test set available we need to find another metric that represents this accuracy.

In the case of the SVD we can use the resulting matrices to get important information about the layer. As seen in figure 1, apart from the two matrices of eigenvectors U and V^T, we also get a single diagonal matrix Σ consisting of eigenvalues. These eigenvalues are in decreasing order for the weights, thus when we plot the values along the diagonal of Σ we get this figure.

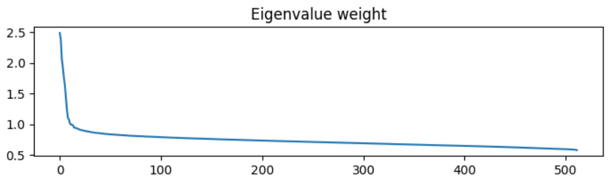

Figure 6: The values on each index of the diagonal in Σ

As you might see this does inversely correlate with the accuracy of the model in figure 6, so we used this metric as a substitute for accuracy, in particular the error rate which is the inverse of accuracy. Meaning that when the values of the ranks go down, then the accuracy goes up. Since we would want the metric to be as low as possible, in order to get a high accuracy, it would be simple to just select the minimal value of the graph, which in most, if not all, cases will be the largest rank. However we want to also see the effect of rank on the file size. For this we create another graph looking at the effects of a different rank to the resulting size of the layer, since this is a paper about method compression after all. As seen in the explanation of the SVD in the methodology and using figure 6 as an example, if the rank r is increased by one then $n * r + d * r = (n + d) * r$ new connections are added. This will create a linear relation between rank and file size. If we plot the file size use of the largest fully connected layer of AgeNet on a graph and we get this as a result.

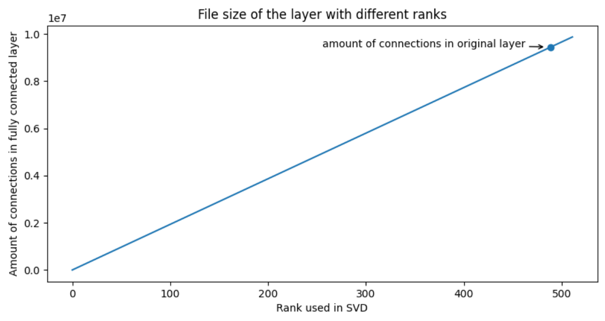

Figure 7: Amount of memory used by the layer compared to their rank

As you can see, there is a point where there is a limit to the positive effect of SVD on file size, at least at very high ranks. This is because there is a point where SVD actually increases filesize, this can be seen if we look at the formulas that calculate the amount of connections in both the original fully connected layer and the layers resulting from the SVD. The formula for the amount of connections in a fully connected layer with n inputs and d outputs is $n * d$. As explained earlier, the amount of connections in the same fully connected layer but with SVD applied and rank r gives us $(n + d) * r$. With this we can clearly see that eventually the layer that had SVD applied to it can be bigger than the original layer at a high rank, specifically at rank $r = \frac{nd}{n+d}$.

For this reason we consider the upper bound of the rank to be at $\frac{nd}{n+d}$ and the lower bound at 1 and normalize the file size according to the rank between these values.

We do the same thing for the eigenvalues, normalizing them between the maximum value and zero.

This results in two different normalized graphs, one representing decrease in error as rank increases. The other graph represent file size, which increases linearly as rank increases. This is where the user specified ratio comes in. The ratio which the user provided gives the tradeoff between filesize and accuracy. We thus add the two graphs together using this ratio like so:

$$filesize\_of(r) * ratio + accuracy\_of(r) * (1 - ratio)$$

Where r is a rank. In figure 8 we show an example on the effects different ratio is fully connected layer of AgeNet using ratios: 0.2, 0.4, 0.6 and 0.8 in that order.

Each ratio gives a different unique graph, and in order to take the optimal ratio for a given ratio between accuracy and filesize we find the minimum of the corresponding graph. It

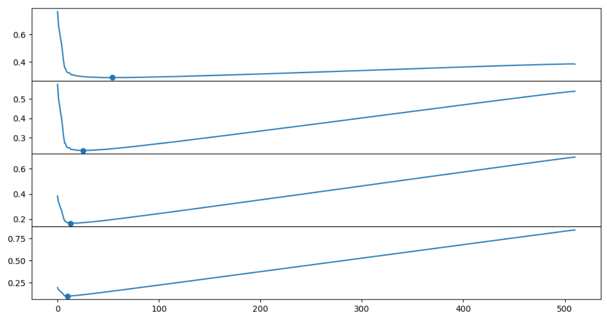

Figure 8: Effects on changing in ratio for the choice of the rank

is visible in the graphs that as the ratio increases, the rank decreases and a decrease in rank results in lower filesize and lower accuracy.

CPD Rank Optimization

As stated in the previous section, while the results of the SVD and CPD seem very similar, they do not compute the resulting matrices in the same way. While SVD calculates the new matrices at first and then truncates them according to a given rank, CPD creates the matrices with a rank r and tries to improve the accuracy of the new matrices in an iterative process. Each time you generate a solution for a CPD of a convolutional layer for a new rank, it has to calculate the CPD from scratch instead of just truncating a previous solution. This results in a lot of processing in trying to find the optimal rank. In figure 9 we show the difference between the CPD approximation of a convolutional layer and the original convolutional layer.

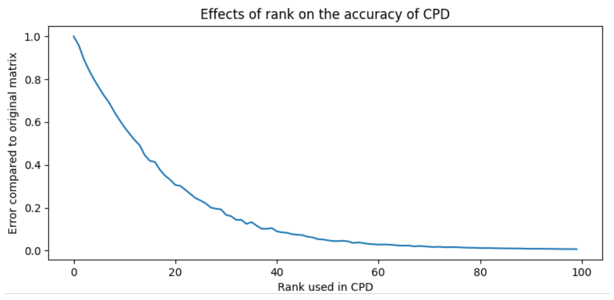

Figure 9: Effect of ranks 1 through 100 on the difference of the decomposition

This figure is only calculated up to 100 ranks, since it's too computationally expensive to calculate this up to the maximum amount of ranks for a given layer, that's why we need to find another method of computing the accuracy lost when applying CPD to a layer. As of this moment we could not find a method that improves the speed of CPD computations or a better way to see the effects of a rank on the accuracy. This is why we will only be compressing convolutional layers with a max rank below a threshold since the lower ranks can be calculated quickly. The threshold chosen in this paper relies on the application but we found that a threshold of 100 works well.

Pipeline

For the pipeline itself, the steps are very simple. We will show how it works using the pseudocode below.

Algorithm 1: Compression Pipeline

```
input: A ratio $x$ used for compression
       A network $net$ that we want to compress
Result: The compressed network $compressed\_net$
$compressed\_network \leftarrow create_empty_network()$;
for $layer \in net$ do
    if $layer$ is convolutional layer then
        $r \leftarrow$ optimal rank of $layer$ using ratio $x$;
        $layer_{cpd} \leftarrow$ CPD of $layer$ using rank $r$;
        add $layer_{cpd}$ to $compressed\_network$;
    else if $layer$ is fully connected layer then
        $r \leftarrow$ optimal rank of $layer$ using ratio $x$;
        $layer_{svd} \leftarrow$ SVD of $layer$ using rank $r$;
        add $layer_{svd}$ to $compressed\_network$;
    else
        add $layer$ to $compressed\_network$;
    end
end
```

The algorithm takes the two inputs from the user, original model and ratio, and uses these arguments to create a compressed model. It recreates a new compressed model based on the original model by compressing the convolutional and fully connected layers.

Restriction Violations

For the final piece of the compression pipeline and before we look at the results we will revisit the restrictions in this paper and see why none of them were violated. The main restrictions of this paper were: Retraining, Resource Constraints, Partial Loading and No change in architecture.

There is not a single instance of retraining in this compression pipeline so this restriction is not violated.

While not explicitly stated, all of the methods can be ran on a device with constrained resources. Even though the CPD of a convolutional layer can be really taxing, we chose an upper bound for when checking for ranks in order to make it run with a limited amount of resources.

The compression pipeline does not remove the capability of model to be partially loaded as we do not change the architecture of the underlying model. In fact the compression pipeline helps the partial loading by turning large layers into multiple smaller ones, this increases the granularity of the layers in the compressed model.

4 Experimental Setup and Results

Testbed and Setup

The experimental setup for this research was an Ubuntu virtual machine running on a Windows 10 host. Since the results of this paper are not based on performance, the specifications of the computer running the tests are irrelevant.

The software framework for running the pipeline and validating the models is Caffe[5]. This is a neural network framework written in C++ which is a framework that is focused on speed and thus can be used on an edge device. The compression pipeline is written in Python 3.7 [8] and uses pycaffe, which is a python interface for Caffe, to read and save the models.

Tested Models

The convolutional neural network models tested in this paper are the GenderNet[7] and AgeNet [7]. These models are used in the Caffe format and are thus compatible with the pipeline. The two models, GenderNet and AgeNet, are very similar in architecture as they both have the same layers in the same order. Three convolutional layers followed by three fully connected layers. The main difference between these two models is the ammount of different classifications of each model. GenderNet only checks if the person in an image is male or female, so just two classifications. Instead AgeNet gives an age range, in this case the model uses eight different age ranges, so eight classifications.

The setup for generating the results of the compression pipeline for a given model with different ratio's is done by first creating 21 different ratio's. In our case these are just all ratios from 0 to 1 with 0.05 step increments. Then we run the pipeline on the given model using all of these ratio's one by one. This gives us 21 different compressed models each based on a different ratio.

We do the same using a naive approach for compression. This uses the same techniques as our compression pipeline but has a simpler and more linear way to calculate the optimal rank. This pipelien will instead just take bases the rank directly on the ratio without looking at the accuracy of filesize. This method for calculating the rank of a layer is defined by:

$$rank = (1 - ratio) * max\_rank$$

We also apply this naive pipeline, referred to as "linear", to the model with the same ratio's. This gives us 42 compressed models per model using two different compression pipelines per ratio.

Test Set

The test set that was used for the validation of the models is the "AdienceFaces" dataset [2]. This is a test set containing labelled images with age and gender data. This is also the dataset that was used in training the GenderNet and AgeNet models.

All model comparisons are calculated by testing the accuracy of each model using the same test set. This means that we take a random subset of 1000 labelled images and use it to infer every model and see if the result of that model is the same as the label of the image. We use the same random subset for each model to remove any variations in accuracy created by validating each model with different image subsets.

Results

Before going into the results of the pipeline we have to look at the baseline for the tested methods. These baselines can be seen in table 1 and help to show the effects of compression.

Models	Accuracy (%)	File Size (MB)
AgeNet	59.4	47.6
GenderNet	84.9	47.8

Table 1: Baseline values for the different tested models

Effects of ratio on compression

Here we will show the different effects of ratio for each compression pipeline on each model. In order to keep the table small, we will pick out three evenly spaced ratio's in this table. We'll pick the extremes 0 and 1 and include the halfway point 0.5. This table is here for me to explain and visualize the trade-off between accuracy and file size. After the table we'll take a look at the final results in graphs.

Ratio	Model	Pipeline	Accuracy	File Size
0	AgeNet	Our	59.4%	47.6 MB
		Linear	59.4%	47.6 MB
	GenderNet	Our	84.9%	47.8 MB
		Linear	84.9%	47.8 MB
0.5	AgeNet	Our	35.0%	7.8 MB
		Linear	45.6%	26.8 MB
	GenderNet	Our	85.0%	7.0 MB
		Linear	84.9%	27.0 MB
1	AgeNet	Our	13.2%	6.1 MB
		Linear	11.3%	6.1 MB
	GenderNet	Our	46.1%	6.1 MB
		Linear	49.3%	6.1 MB

Table 2: Effects of ratio on both our pipeline and a naive approach

As expected a ratio of 0 doesn't change the model from the baseline as this value of the ratio is fully focused on accuracy. It is usually the case that compression comes at a cost of information and with less information the model should become less accurate, that's why the model does not compress with a ratio of 0. However if you look at the data using our pipeline for GenderNet using ratio 0.5, you can actually see a small accuracy increase, this could be attributed to noise reduction in the model due to the compression. It's interesting to see but this could be attributed to bias in the data set and will be elaborated on in the responsible research section.

Now we'll take a look at 1. A ratio of 1 minimizes filesize with no regard to accuracy. As shown in the table, the accuracy for both models have gone down to roughly the same as a random guess. As GenderNet has two choices, a correct random guess would happen 50% of the time, on the other hand AgeNet has eight choices, in this case a correct random guess would happen 12.5% of the time. This seems to correlate with the accuracy of the models that were minimizing filesize, which is very low but basically useless since the accuracy is as good as a random guesss.

Finally we will look at a ratio of 0.5, this ratio shows different things, in the case of GenderNet our pipeline clearly outperforms a linear compression method by having a filesize which is over 4 times as small with comparable accuracy. However when looking at AgeNet we may have some conflicting ideas, the filesize is a lot smaller but so is the accuracy. This is why our pipeline is configurable with a ratio. If users think the accuracy is too low, they can decrease the ratio such that the accuracy will increase at the cost of file size.

Resulting figures

Considering our previous findings it's interesting to see the continuous effects of the ratio on models, to see this we apply our testset on all 42 models per model as specified in the "Tested Models" section. We will take a look at three different graphs per model, each model will show the effects of a specific rank on a metric of that model, on both our pipeline and the linear pipeline. The metrics we will take a look at are: Accuracy, File Size and Compression Factor. Accuracy and File Size are self explanatory. Compression factor basically shows the quality of a compression. The calculation for the compression factor cf of a compressed model with accuracy acc and file size fs, given the accuracy acc_{orig} and file size fs_{orig} of the uncompressed model is like so:

$$cf = \frac{acc/acc_{orig}}{fs/fs_{orig}}$$

This leads compressed models with the same accuracy as the original but half the filesize to have a compression factor of 2. And the original model having a compression factor of 1.

AgeNet

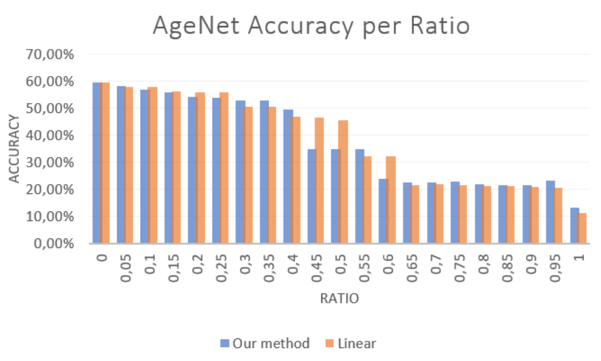

Figure 10: The accuracy of both pipelines using different ratios

In figure 10 the results don't seem spectacular, the accuracy of the models from our pipeline are very close to the accuracy of the models with the same ratio from the linear pipeline. Sometimes the accuracy of the linear pipeline is actually higher than ours. This slightly disappointing result bring us to our next figure.

In figure 11 we can finally see where our linear pipeline got it's name from. Apart from that we can luckily see that our pipeline is significantly outperforming the linear pipeline at every ratio except for 1 and 0, which results in the pipelines compressing the models the same way, so this should not be a surprise. Evidence of this is seen in table 2.

Finally we reach figure 12 which is arguably the most important graph. This graph shows that our pipeline again significantly outperforms the linear pipeline. This compression

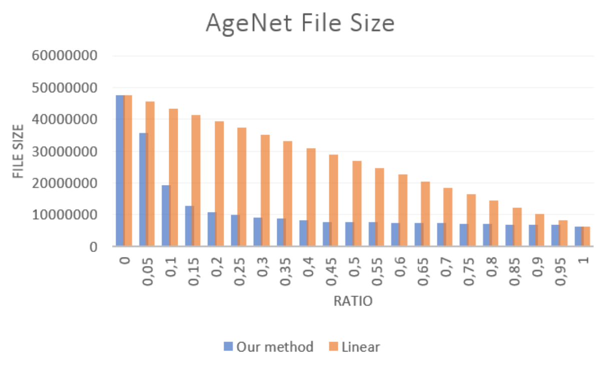

Figure 11: The file size of both pipelines using different ratios

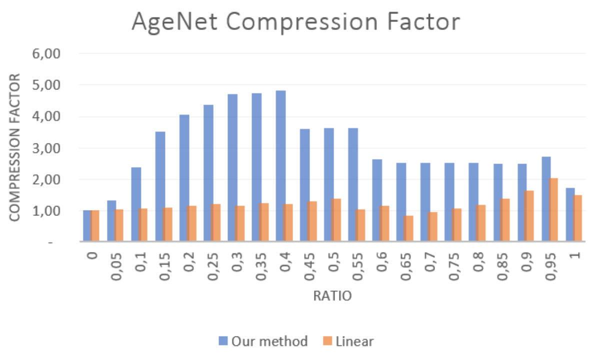

Figure 12: The compression factor of both pipelines using different ratios

factor shows that there is never a moment where the linear pipeline would be a better option that our pipeline, except at 0 and 1. This means that you will never get a lower accuracy when using our pipeline compared to a model from the linear pipeline if they have the same size, at least in the case for AgeNet.

GenderNet

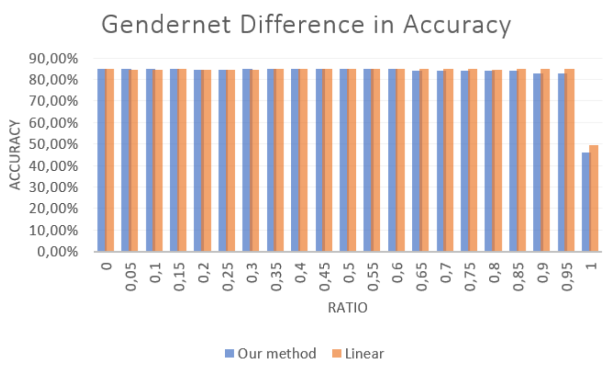

Figure 13: The accuracy of both pipelines using different ratios

In figure 13 the results are interesting to say the least. Again, the accuracy of the models from our pipeline are very close to the accuracy of the models with the same ratio from the linear pipeline.

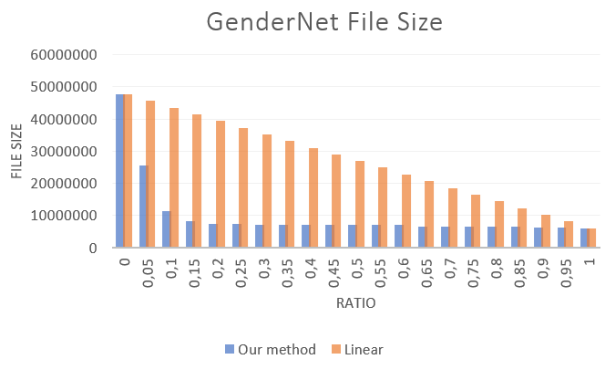

Figure 14: The file size of both pipelines using different ratios

In figure 14 we can see once again that our pipeline is significantly outperforming the linear pipeline at every ratio except for 1 and 0.

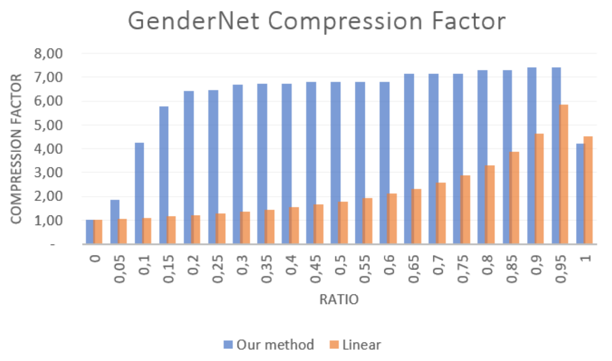

Figure 15: The compression factor of both pipelines using different ratios

Then finally we reach figure 15. This graph shows that our pipeline again significantly outperforms the linear pipeline. All the same things apply to GenderNet just like AgeNet, for both model you will never get a lower accuracy when using our pipeline compared to using linear pipeline if the resuling compressed models have the same size.

5 Responsible Research

We tried to explain the methods used in this paper thoroughly enough for others to replicate. Nevertheless, we will release the source code at some time in the future such that anyone can use the compression pipeline and validate the results for themselves.

6 Conclusions and Future Work

In this paper, we presented a compression pipeline that is configurable by the user to answer the research question:

What is the best method of compressing neural network models, provided the compression has to be done in an environment with resource constraints.

Given the constraints and scope of the research which were:

- Retraining: The results of my research should be a compression pipeline that does not depend on retraining, since this will be inconvenient/impossible on edge devices.
- Resource Constraints: The compression pipeline should work with resource constraints. This is the main reason why there is no retraining.
- Partial Loading: The model resulting from the compression pipeline should be able to be partially loaded.
- No change in architecture: The output model from the compression pipeline needs to be a Caffe model having the same supported layers as the input model.

The answer to this question is to compress fully connected and convolutional layers using a technique called singular value decomposition and canonical polyadic decomposition respectively. These techniques do not break the constraints set in the paper, and provide results based on the users chosen trade-off between file-size and accuracy.

Future Work

In this paper we already stated some interesting methods in neural network compression that we did not implement in our compression pipeline usually because of the ”No Change in Architecture” restriction. There can be a lot of improvements however if we disregard this restriction. If you would allow the compression pipeline to use a modified version of Caffe, or even a whole different framework, the pipeline could use additional layer compression techniques.

As for improvements on the current technique, there are a lot of restrictions on the compression of convolutional layers in our compression pipeline. The largest part of this is the problem with calculating the optimal rank for CPD using any filesize/accuracy ratio, as this is very computationally expensive. There could be further work into finding an optimal rank faster or using a different method for compressing convolutional layers

Another issue is that in it’s current state, our pipeline likely requires multiple runs with different ratios in order to find a model that is roughly the file size you want.

References

[1] Guobin Chen, Wongun Choi, Xiang Yu, Tony X. Han, and Manmohan Krishna Chandraker. Learning efficient object detection models with knowledge distillation. In *NIPS*, 2017.

[2] Eran Eidinger, Roee Enbar, and Tal Hassner. Age and gender estimation of unfiltered faces. *Information Forensics and Security, IEEE Transactions on*, 9:2170–2179, 12 2014.

[3] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network, 2015.

[4] Frank L. Hitchcock. The expression of a tensor or a polyadic as a sum of products. *Journal of Mathematics and Physics*, 6(1-4):164–189, 1927.

[5] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[6] V. Klema and A. Laub. The singular value decomposition: Its computation and some applications. *IEEE Transactions on Automatic Control*, 25(2):164–176, 1980.

[7] Gil Levi and Tal Hassncer. Age and gender classification using convolutional neural networks. *2015 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2015.

[8] Python. https://www.python.org/downloads/.

[9] Xukan Ran, Haoliang Chen, Zhenming Liu, and Jiasi Chen. Delivering deep learning to mobile devices via offloading. In *Proceedings of the Workshop on Virtual Reality and Augmented Reality Network*, VR/AR Network ’17, page 42–47, New York, NY, USA, 2017. Association for Computing Machinery.

[10] Shobhit Srivastava, Shubhadeep Guha, Alex Jenkins, Research Analyst, and Counterpoint Research. Apple to drive native ai adoption in smartphones, Oct 2017.

[11] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks, 2016.

[12] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. Incremental network quantization: Towards lossless cnns with low-precision weights, 2017.