Face Recognition Based on Embedded Systems

Hanna Björgvinsdottir, Robin Seibold

Master's thesis
2016:E17

LUND UNIVERSITY

Faculty of Engineering
Centre for Mathematical Sciences
Mathematics

CENTRUM SCIENTIARUM MATHEMATICARUM

Face Recognition Based on Embedded Systems

Robin Seibold
dat11rse@student.lu.se
Robinbobseibold@gmail.com

Hanna Björgvinsdóttir
dat11hbj@student.lu.se
hanna.bjorgvinsdottir@gmail.com

Centre for Mathematical Sciences, Lund University
Sölvegatan 18, P.O. Box 118, 221 00 Lund

Master's thesis work carried out at Axis Communications AB

Supervised by
Kalle Åström, kalle@maths.lth.se
Jiandan Chen, jiandan.chen@axis.com
Martin Ljungqvist, martin.ljungqvist@axis.com

Examiner
Niels Christian Overgaard, Niels_Christian.Overgaard@math.lth.se

June 1, 2016

Abstract

Machine learning in general, and artificial neural networks in particular, have gained a lot of attention in recent years. Using deep neural networks for classification tasks, such as face recognition, has proven more and more successful over time. The performance increase is partly due to more complex network architectures, and partly due to the use of larger datasets.

The increased complexity of networks has lead to an increase in parameters, which in turn results in slower training and inference, making it hard to deploy such models on limited hardware.

The main objective of this master's thesis is to train a convolutional neural network for face recognition, and deploy it on an embedded system, with the aim of real-time performance.

By using transfer-learning as a means to adjust a pre-trained model to fit new data, the time needed for the training phase is reduced. The resulting model achieves an accuracy of 91.66%, while distinguishing between 2,904 identities.

The model is then compressed by a method referred to as pruning, reducing the amount of parameters in the fully connected layers by a factor of 20, greatly reducing the memory footprint while remaining within $\sim 1\%$ of the original accuracy.

Finally, by combining the resulting neural network model with a custom built framework and a live video stream, real-time face recognition is achieved on an embedded device.

Acknowledgement

We would like to thank Axis Communications AB, for giving us the opportunity to do this master's thesis at Axis, and for providing all necessary utilities.

We would also like to thank our supervisors at Axis, Martin Ljungqvist and Jiandan Chen, for excellent guidance and support.

Big thanks to our supervisor Kalle Åström as well, for valuable comments and creative ideas.

Last, but not least, we would like to thank the brave souls who contributed to our dataset with images of themselves; Martin Ljungqvist, Jiandan Chen, Kalle Åström, Joakim Roubert, Jakob Beckerot, and Åke Södergård.

Contents

1 Introduction **1**
1.1 Main Objective . . . 1
1.2 Problem Description . . . 1
1.3 Related Work . . . 1
1.3.1 Face Recognition and Verification . . . 1
1.3.2 Transfer Learning . . . 2
1.3.3 Compression of Neural Networks . . . 2
1.3.4 Data Augmentation . . . 3
1.4 Utilities . . . 4
1.4.1 Hardware . . . 4
1.4.1.1 AXIS A8004-VE Network Video Door Station . . . 4
1.4.2 CUDA . . . 4
1.4.3 Caffe . . . 5
1.4.3.1 Caffe Model Zoo . . . 5

2 Methodology **6**
2.1 Neural Networks . . . 6
2.1.1 Forward Propagation . . . 6
2.1.1.1 Activation Functions . . . 6
2.1.2 Back Propagation . . . 7
2.1.2.1 Softmax Loss . . . 7
2.1.2.2 Stochastic Gradient Descent . . . 7
2.1.3 Layer Types . . . 7
2.1.3.1 Fully Connected Layer . . . 7
2.1.3.2 Convolutional Layer . . . 8
2.1.4 Max-Pooling . . . 9
2.1.5 Overfitting . . . 9
2.1.5.1 Dropout . . . 9
2.1.5.2 L_2 Regularization . . . 9
2.2 VGG16 . . . 10
2.2.1 VGG Face . . . 10
2.3 Transfer Learning . . . 11
2.4 Preprocessing . . . 12
2.5 Compressing Neural Networks . . . 12
2.5.1 Floating Point Precision Reduction . . . 13
2.5.2 Pruning . . . 14
2.5.2.1 Selection of Weights to Prune . . . 14
2.5.2.2 Weight Representation After Pruning . . . 15
2.6 Datasets . . . 16
2.6.1 Labeled Faces In The Wild . . . 17
2.6.2 Oxford VGG Face Dataset . . . 17
2.6.3 FaceScrub . . . 17
2.6.4 Additional Dataset . . . 17
2.7 Dataset Collection . . . 18
2.8 Data Analysis . . . 18
2.8.1 Data Quantity and Augmentation . . . 18
2.8.2 Input Image Size . . . 19
2.9 Prototype . . . 19
2.9.1 Video Capture . . . 19
2.9.2 Detection . . . 19
2.9.2.1 Feature-based Cascade Classifier . . . 20
2.9.2.2 Deformable Part Models . . . 20
2.9.3 Recognition . . . 20

3 Results and Discussion **21**
3.1 Transfer Learning 21
3.2 Neural Network Compression 23
3.2.1 Floating Point Precision Reduction 23
3.2.2 Pruning 24
3.2.3 Inference Time Comparison 25
3.3 Data Analysis 26
3.3.1 Data Quantity and Augmentation 26
3.3.2 Image Size 26
3.4 Impostors 27
3.5 Prototype 28

4 Conclusions **29**
4.1 Future Work 29

1 Introduction

Deep learning is a concept which has been, and still is, on the rise in many scientific fields, showing remarkable results in multiple areas, including object recognition. It is an approach to machine learning that to this day is being explored and improved continuously.

In recent years, improvement to neural networks used for object recognition and verification have largely been due to network expansions and an increase in training data. However, complex network architectures, such as AlexNet [Krizhevsky et al., 2012] and VGG-16 [Parkhi et al., 2015], although producing good accuracy, contain tens and hundreds of millions of parameters respectively, which can make deployment on embedded systems with limited compute power infeasible.

1.1 Main Objective

The aim of this master's thesis is to investigate the possibility of deploying a neural network for face recognition on an embedded platform, achieving good accuracy while simultaneously maintaining acceptable inference speed.

Different techniques for reducing the memory and time spent on both the inference and training part of neural networks are explored.

The objective is not to design a new neural network architecture, but rather to train an existing one on new datasets, and to find ways of compressing the resulting model.

The aim is also to develop a prototype, which uses the trained model to classify faces detected in a live camera feed.

1.2 Problem Description

As stated above, the main objective of this thesis is to deploy a neural network for face recognition on an embedded platform. The difference between face *verification* and face *recognition* is that verification is a $1 : 1$ comparison, while recognition is a $1 : N$ comparison. In other words, verification answers the question "Does image A depict the same identity as image B?", and recognition answers the question "Which identity in database D belongs to image A?".

In face recognition, finding an identity in the database can involve either *closed-set* or *open-set* identification. In the former, all identities are guaranteed to be in the database, and the mission is simply to decide which identity it is. In the latter, the identities are not guaranteed to be in the database. This introduces the problem of not only classifying the identities in the database, but also detecting impostors excluded from it. The face recognition performed in this thesis is of the kind that uses open-set identification.

Making a neural network perform well requires a lot of training, often involving tweaking of different parameters, and can therefor be extremely time consuming. Seeing as the time frame of this thesis is limited, the time spent on training the network is as well.

Another challenge concerning training is acquiring the amount of high quality training data needed to produce a good model.

When considering deployment of the trained model on an embedded system, three major factors must be taken into account; limited compute power, limited memory, and limited bandwidth. These factors place restrictions on the number of parameters in the network, as well as the size of the input.

Reducing the size of a model not only involves the problem of *how* to compress it, but also of how to preserve the accuracy. Both the size of the model stored on the hard drive, and the memory needed to perform a forward pass are of concern.

In addition to compression, ways to optimize the inference would be highly beneficial due to the limited compute power of the embedded system.

1.3 Related Work

Both face verification and recognition are problems that have been popular in computer vision and image analysis research for a long time. The realization that neural networks can be used for the task has made the subject even more popular.

The training phase of neural networks generally require a lot of time, compute power, and training data. Because of this, a fair amount of research has been conducted on the subject of reducing these factors.

1.3.1 Face Recognition and Verification

Two years ago, a now well known article called *DeepFace: Closing the Gap to Human-Level Performance in Face Verification* [Taigman et al., 2014] was released. The authors had successfully developed a face verification system that reached 97.35 % accuracy on the Labeled Faces in the Wild (LFW) [Huang et al., 2007] bench marking data set, following the unrestricted protocol. The (at the time) state of the art error rate was reduced by 27 %, almost reaching human-level performance. Their method consists of a set of preprocessing algorithms for face alignment, and a deep neural network for the verification. The preprocessing steps include both 2D and

3D alignments, based on a great amount of reference points.

A year later, researchers from Google published the paper *FaceNet: A Unified Embedding for Face Recognition and Clustering* [Schroff et al., 2015], introducing feature vectors of only 128 bytes per face. In addition to a convolutional neural network (CNN), embedded triplet loss was used to calculate these features, resulting in an accuracy of 99.63 % on the unrestricted LFW set. Besides the superior accuracy compared to DeepFace, FaceNet benefits from not requiring heavy preprocessing steps. Using only cropping of the faces, an accuracy of 98.87 % was achieved. The best reported accuracy was achieved using similarity transform alignment in addition to the cropping.

Shortly after the FaceNet publication, the Visual Geometry Group (VGG) at the University of Oxford published a paper called *Deep Face Recognition*, where a convolutional neural network was trained for face recognition [Parkhi et al., 2015]. After training the classifier, the last feature vector (next-to-last layer) was tuned for face verification, using embedded triplet loss. The final model achieved an accuracy of 98.95 % under the unrestricted setting of LFW, using less training data than both DeepFace and FaceNet. The authors explored the impact of using similarity transform alignment (like the one in FaceNet) during both the training and test phase, and found that the network only benefited from alignment during the test phase.

1.3.2 Transfer Learning

Convolutional neural networks are comprised of several layers, which can be seen as different feature extractors, trained to find the differentiating features of a given dataset. The generality of these features depend both on the type of data used, and the objective of the training. Studies have shown that the features get more and more specific the further into the network the layers are [Bengio, 2012]. The first layers normally react to general low-level features (such as edges, in the case of CNNs trained on images) while the later layers detect features of more and more complexity.

Because of the generality of the initial features, the first part of a CNN trained for one type of images can produce features that are of interest for other types of images as well.

Training large CNNs from scratch is time-consuming, and requires a lot of compute power, as well as large amounts of training data. One way of overcoming these obstacles is by taking advantage of the generality of features, and transferring them from a trained model (base model) to the target model [Azizpour et al., 2014].

There are two common methods of transferring features. One way is to simply copy the weights of the N layers which are thought to be of interest, 'freeze' them, and only train the later layers of the target model, which are initialized either randomly or from some known distribution. The other method is similar, only the copied weights are also trained, but with a lower learning rate. This process is often referred to as fine-tuning [Grundström et al., 2016].

The first method is usually preferred when the dataset of the target model is much smaller than that of the base model. In this case, the model will risk being overfitted if fine-tuning is performed. However, when the dataset of the target model is large, fine-tuning can improve the performance of the target model [Yosinski et al., 2014].

By transferring features, one can get the advantages of long training with large datasets, without having to perform it all oneself. Furthermore, when the first N layers of the target model are frozen, the number of parameters that require training are decreased, which leads to faster learning.

How many layers should be transferred depends on the architecture of the CNN, and the similarity of tasks for the base and target models. For example, if the base CNN is trained for classifying dog breeds, more layers could probably be transferred if the target CNN is to be trained for classifying cat breeds, than if its purpose is to recognize art styles.

1.3.3 Compression of Neural Networks

In recent years the architectures of neural networks have been getting more and more complex, and with the complexity the total number of parameters increases. Studying the more recent neural networks shows an almost exponential growth of both the parameters and the number of computations needed for forward and backward propagation.

The introduction of using parallel algorithms and using the supreme parallel computing powers of the graphical processing units (GPUs) has made neural networks feasible despite their size. Training a very deep neural network can still take days, even with the use of multiple GPUs, and the mere size can make it impossible to deploy the trained network on systems with limited hardware.

The interest for the topic of compressing neural networks both in size and computational time for both the training, storing, and deployment phase, has been growing alongside the neural network size. However, most of the focus has been on the training phase, while the focus in this master's thesis will mostly be on the deployment phase.

The authors of the paper *Compressing Deep Convolutional Networks using Vector Quantization*

[Gong et al., 2014], have developed successful methods that according to them can compress a deep neural network up to 24×, while only reducing the network accuracy with 1%. In the paper they describe different approaches for compression, including matrix factorization and different vector quantization approaches.

Another paper, mostly concerning offline compression of neural networks, is *Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding* [Han et al., 2015a]. This paper introduces three different approaches, which can be combined, resulting in a 35× and 49× offline compression for AlexNet and VGG-16 respectively.

While the previous paper is mostly focusing on the offline compression of a neural network, another article was also published [Han et al., 2015b], where the focus is on the compression approach referred to as pruning, which also can be used when the neural network is deployed. *Optimal Brain Surgeon and General Network Pruning* [Hassibi et al., 1993] is another, older, paper about pruning. In this paper the authors explored the result of pruning weights based on the second-order derivatives of the error function.

Another approach to neural network compression is to not focus on the neural network structure, or how the computations inside are handled, but instead to target the data types that represent the neural network. The usual data type that is used in neural networks today is the single precision floating point data type. This data type is represented by 32 bits, or 4 bytes, in computer memory, and has a precision of approximately 7 decimal digits.

The interesting thing with using a data type represented by less bits and with a lower precision for neural networks is that it can affect the speed and memory footprint for both phases of a neural network, that is, both the training phase and the deployment phase. In addition to this it also compresses the offline storage needed.

In the article *Improving the speed of neural networks on CPUs* [Vanhoucke et al., 2011] a neural network is implemented using fixed point data types, instead of floating point. For instance a solution is presented where the author uses the data type unsigned char for neuron activation values, and signed char for weights.

There also exists other approaches where a much more aggressive data type reduction has been made, one example is the paper *XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks* [Rastegari et al., 2016]. In this paper the authors present two different types of neural networks, one where the weights are represented as binary values, that is 0 or 1, and another solution where both the weights and the inputs to layers are represented in binary form.

In the article they structure the training phase according to the binary format, and by doing so they can keep a good accuracy, despite the low resolution of the weights and inputs.

1.3.4 Data Augmentation

Neural networks require a large quantity of data when training, in order to produce good results. Collecting such large datasets is not always feasible, due to lack of time or resources.

One way of getting around this issue is by expanding smaller datasets, using different transformations or other kinds of processing. In *ImageNet Classification with Deep Convolutional Neural Networks* [Krizhevsky et al., 2012], data augmentation is described as a way of preventing overfitting. After down-sampling all images, making the shortest side 256 pixels long, and cropping the center 256 × 256 patch, two augmentation techniques were used to expand the training data. First, random crops of size 224 × 224 were extracted from the 256 × 256 center crops of the original images. Both the crops and their horizontally flipped counterparts were used for training, in total expanding the training set by a factor of 2048. When testing, five crops and their horizontal reflections were fed through the network, and the average prediction used as final result. The crops were extracted from the four corners of the image, and the center.

The second approach used was principal component analysis (PCA). Multiples of the principal components were added to each image, changing the color and intensity of images each time they where used for training. This approach resulted in a top-1 error decrease of over 1% [Krizhevsky et al., 2012].

A similar method as the cropping above is described in *Return of the Devil in the Details: Delving Deep into Convolutional Nets* [Chatfield et al., 2014]. Five crops, and their horizontal reflections, were extracted from the corners and center of the training images, expanding the training set 10×. As opposed to the above, Chatfield et al. only downsampled the images before performing the augmentation, and so used the entire image as a basis, instead of the center crop. As a comparison, experiments were performed with augmenting the training set solely by reflection, expanding the set only 2×. The first method resulted in a mean average precision increase of around 3%, while the second showed less improvement.

The recently published paper *Do We Really Need to Collect Millions of Faces for Effective Face Recognition?* [Masi et al., 2016], describes a way of synthe-

		Embedded		
		Jetson TK1	**Jetson TX1**	**High End**
CPU	**Name**	ARM®Cortex-A15	ARM®Cortex-A57	Intel®Core i7-5820K
	Architecture	32-bit	64-bit	64-bit
GPU	**Architecture**	Kepler	Maxwell	Maxwell
	CUDA Cores	192	256	3072
	Memory Bandwidth	14.9 GB/s	25.6 GB/s	336.5 GB/s
	GFLOPs (FP32) Peak	365	512	6144
	GFLOPs (FP16) Peak	365	1024	-
	Compute Capability*	3.2	5.3	5.2
Memory	**RAM**	2 GB (shared)	4 GB (shared)	32 GB
	Graphics Memory			12 GB

Table 1.1: Hardware specifications. *measurement used by CUDA to divide hardware into groups based on different performance metrics.

sizing training data for face recognition, by manipulating the original dataset using different 3D models. The goal was to increase the appearance variability, by changing the pose, shape, and expression of the faces. Training a model with the augmented set of 2,400,000 images increased the accuracy from 95.31% to 98.06%, compared to a model trained on the original 495,000 images.

1.4 Utilities

The interest in neural networks has in recent times led to an increase in software and tools for studying and working with these. Below the most relevant software used in this master's thesis will be described, as well as the hardware used during development, and later the neural network deployment.

1.4.1 Hardware

The different phases of the neural network algorithm, and the different constraints these phases put on the hardware, leads to a big variety in hardware requirements.

The four main hardware setups used in this master's thesis can be split into three different categories; embedded hardware, mid range, and high end. Two of the four hardware setups are embedded systems with limited compute power, since the goal of this master's thesis was to deploy a neural network, trained for facial recognition, on such systems.

The two embedded systems used are the NVIDIA Tegra K1 and the NVIDIA Tegra X1. The Tegra series is a system on chip solution from NVIDIA, developed to be used for smartphones and the like. The deployment phase was always tested and evaluated on these devices.

All development and corresponding testing was done on mid range hardware, corresponding to a workstation PC, with an NVIDIA GeForce GTX 950 graphics card. Since the training phase of neural networks are the most time-, memory- and data-consuming, a special high end computer was used for the training phase.

The specifications for the embedded systems and the high end setup can be seen in Table 1.1. Most of the focus is on the different GPU's of these systems, and the corresponding graphical memory, since the GPU does most of the calculations, while the CPU acts more like a host. Since no performance evaluation was carried out for the mid range hardware, it is omitted from the table.

1.4.1.1 AXIS A8004-VE Network Video Door Station

In the prototype developed for this master's thesis, described in Section 2.9, an AXIS A8004-VE Network Video Door Station [Axis, 2015] is used to capture video. The specifications of the camera can be found in the referenced datasheet.

1.4.2 CUDA

Much of the software developed, and worked with, during this master's thesis included heavy computations. For that reason many of these computations were executed on the GPU using CUDA, which is a parallel computing platform and application programming interface (API) developed by NVIDIA.

CUDA also includes libraries such as cuBLAS, which is a CUDA implementation of the specification

Basic Linear Algebra Subprograms (BLAS), and cuSPARSE which is the counterpart of cuBLAS, but for sparse matrices. These libraries are optimized, and useful when working with neural networks, mostly because of the General Matrix Multiplication (GEMM) sub routine which calculates

$$\mathbf{C} = \alpha \cdot \mathrm{op}_1(\mathbf{A}) \cdot \mathrm{op}_2(\mathbf{B}) + \beta \cdot \mathbf{C}, \qquad (1.1)$$

where α and β are scalars, and $\mathrm{op}_x(\mathbf{X})$ can be a non-transpose operation, a transpose operation, or a conjugate transpose operation.

1.4.3 Caffe

Caffe is an open source framework for deep learning, developed by the Berkeley Vision and Learning Center (BVLC) [Jia et al., 2014]. The framework mainly contains C++ code, but it includes bindings for both Python and MATLAB.

All the GPU acceleration code is written using NVIDIA CUDA, and the corresponding CUDA libraries. There also exists an option to compile Caffe in symbiosis with NVIDIAs deep neural network library called cuDNN. cuDNN is essentially a library that has very optimized implementations for specific parts of neural networks, and using cuDNN makes some calculations included in the neural network extremely fast. The cuDNN version used in this master's thesis was 4.0.

1.4.3.1 Caffe Model Zoo

The BVLC provides what is called the Caffe Model Zoo, where organizations and individuals can share their trained neural network models. The Model Zoo contains a variety of network architectures, trained on a wide range of data.

2 Methodology

The neural network architecture chosen as a basis of this thesis is the VGG-16 architecture, referred to as architecture D in [Simonyan and Zisserman, 2014]. The reasons behind choosing VGG-16 are:

1. The network has been trained successfully for face classification.

2. VGG-16 is a deep architecture, containing both convolutional and fully connected layers, opening up for different compression approaches.

3. An architecture specification compatible with Caffe is publicly available.

4. A pre-trained version of the network is publicly available.

In the following sections general information regarding neural networks will be presented, along with VGG-16 specifics. This theory will be the basis for later sections where methods for neural network compression and training a neural network for new data are presented.

2.1 Neural Networks

In machine learning, an artificial neural network refers to a model designed after an abstraction of biological neural networks. As a consequence of this, the artificial neural network is capable of learning.

The smallest building block of an artificial neural network is the neuron. The specific implementation of the neuron varies between network architectures, but the essentials are that a neuron takes some input, and produces a corresponding output. A graphical representation of a neuron can be seen in Figure 2.1.

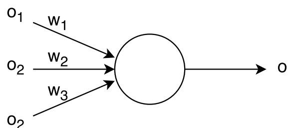


Figure 2.1: A neuron, the smallest building block of a neural network.

Neurons grouped together form a layer, and essentially an artificial neural network is the composition of multiple such layers. The actual application of the layers depends on many parameters, such as intermediate connections between the current layer and the next. Assigned to each connection is a parameter called weight, w, and each neuron is associated with a bias, b. These are the parameters that are trained to improve the behavior of the network.

A neural network has two main phases. The first one is called the training phase, and the second is called the test, or deployment, phase. Both of these phases are mentioned in Section 1. First the training phase is executed, where training data and the correct labels for that data, are given as input to the neural network. The neural network then produces an output based on the input data, which is compared to the label.

The error of the output is used as a basis for how the parameters in the neural network should change, such that the next time that same data is sent to the neural network the error will be less, or preferably non-existent. This procedure is often performed for large sets of data, and is very time consuming.

The deployment phase includes taking a trained neural network and using it on real data. In this phase, no parameters of the neural network are changed. An input is simply given to the network, and the produced output is presented.

In connection to both these phases, something called batch size is often mentioned. This simply refers to the multitude of data with separate labels that are given to the neural network. For instance, a batch size of N for a neural network with images as input, means that N images and N labels are passed to the network at once.

The following sections will describe the above theory more thoroughly.

2.1.1 Forward Propagation

Forward propagation is the process of sending input from the first layer to the second, from the second to the third, and so on, until the final output is given. A neuron in one layer may be connected to all neurons in the previous layer, or to a smaller section. The output from neuron i in layer l can be written as

$$\boldsymbol{o}_i^l = f_a(f(\boldsymbol{o}^{l-1})), \tag{2.1}$$

where $\boldsymbol{o}^{l-1}$ is a vector of all outputs from the previous layer connected to neuron i, f is a layer-specific function, and f_a is the so-called activation function.

The layer-specific function, f, is what specifies a layer's behavior. For example, in a convolutional layer f includes convolutions of $\boldsymbol{o}^{l-1}$, where the weights act as convolution kernels, and the biases are added to the result.

The activation function, f_a, introduces non-linearity to the network, and thereby allows for more complex models.

2.1.1.1 Activation Functions

There are three major activation functions mentioned widely in the literature. The first one is the sigmoid

function. Historically, it has been used to a great extent, but it has lost its appeal in recent years, partially due to its output not being centered around zero.

One of the functions replacing the sigmoid function is tanh, which essentially is a rescaled version of the sigmoid function, producing an output in the range $[-1, 1]$ instead of $[0, 1]$.

The second widely used function is called ReLU (Rectified Linear Unit). The ReLU function only outputs values $\geqslant 0$, as

$$f_a(x) = \max(0, x). \tag{2.2}$$

One of the advantages of using the ReLU function is that it can be implemented by simple thresholding, making the calculations much simpler than those of sigmoid and tanh.

Compared to sigmoid and tanh, the use of ReLU as activation function also tends to result in faster learning [Krizhevsky et al., 2012]. The ReLU function is used in both VGG-16 and AlexNet.

2.1.2 Back Propagation

The back propagation is the part of training where the actual learning happens. After the forward propagation is done, a loss function (sometimes referred to as cost function) is applied to the output from the last layer in the network. The loss function depends on the weights and biases of the network, and measures how bad the network performs, by comparing the expected output to the actual output.

The goal of the training is to minimize this metric. An example of a loss function is the softmax loss.

2.1.2.1 Softmax Loss

In VGG-16, and many other CNNs, the loss function used is softmax. Given an input matrix $\mathbf{X}$ of dimension $N \times K$, N being the batch size and K the number of identities (class labels, outputs), the softmax function is first applied to each sample in the batch. The softmax function transforms its input values to real values in the range $(0, 1)$, such that the sum is equal to 1. For element $\mathbf{X}_{n,k}$, the softmax function is defined as

$$\sigma(\mathbf{X}_{n,k}) = \frac{e^{\mathbf{X}_{n,k}}}{\sum_{j=1}^{K} e^{\mathbf{X}_{n,j}}}. \tag{2.3}$$

Next, the loss is calculated as

$$L = -\frac{1}{N} \sum_{n=1}^{N} \log \left(\frac{e^{\mathbf{X}_{n,l}}}{\sum_{j=1}^{K} e^{\mathbf{X}_{n,j}}} \right), \tag{2.4}$$

where $\mathbf{X}_{n,l}$ is the output corresponding to the correct identity.

2.1.2.2 Stochastic Gradient Descent

Stochastic gradient descent (SGD) is an approach to minimizing the loss function. Normal gradient descent works by updating the weights and the biases by a linear combination of the previous update and the negative average gradient of the weights or biases. The updating rule for weights at iteration $t + 1$ can be written as

$$\begin{aligned} \boldsymbol{v}_{t+1} &= \alpha \boldsymbol{v}_t + \eta \nabla L(\boldsymbol{w}_t), \\ \boldsymbol{w}_{t+1} &= \boldsymbol{w}_t + \boldsymbol{v}_{t+1} \end{aligned} \tag{2.5}$$

and similarly for biases. $\boldsymbol{v}_{t+1}$ is the update value for weights $\boldsymbol{w}_{t+1}$, and $\nabla L(\boldsymbol{w}_t)$ is the negative average gradient. α is a parameter called momentum, which controls how much the previous update value should affect the current. The learning rate, η, in turn controls the effect of the gradient.

The idea behind SGD is the same as for gradient descent, except that the gradient is calculated only for a random sub-sample (a batch), and the average is used as an approximation of the total average. By doing so, lots of costly calculations are saved, thereby speeding up the learning. However, a drawback of this method is that if the sub-sample is too small, the sub-sample average will not be a good enough estimate, causing fluctuation in the algorithm.

2.1.3 Layer Types

The two major layer types used when constructing CNNs are convolutional layers and fully connected layers. In Caffe, and other deep learning frameworks, other types of building blocks, such as pooling, are also referred to as layers. The difference between those and the two mentioned above is in essence two properties.

The first is that out of all the used layers, only the convolutional and fully connected layer have neurons, and therefor parameters like weights and biases. The second property, closely related to the previous property, is that only the parameters of the convolutional layers and the fully connected layers change during training – those are the layers that actually learn.

Since the deployed network in this master thesis is VGG-16, only layers relevant to that architecture will be presented and described below.

2.1.3.1 Fully Connected Layer

A fully connected layer connects all its neurons with every neuron in the previous layer. Both input and output are regarded as one-dimensional vectors. Figure 2.2 depicts a fully connected layer with three input neurons, and two output neurons.

The output, o', for a neuron in a fully connected layer is calculated as

$$o' = f_a\Big(\sum_{j=1}^{n}(w_j o_j) + b\Big), \tag{2.6}$$

where o_i is the output from the output neuron i, w_i is the weight between input neuron i and the output neuron, b is the bias for the output neuron, n is the number of input neurons, and f_a is the activation function for the output neuron.

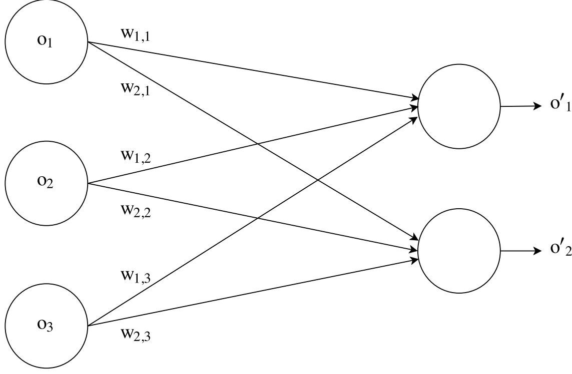


Figure 2.2: A fully connected layer, where every input neuron is connected to every output neuron.

Equation 2.6 can be rewritten as

$$o'_i = f_a\Big(\sum_{j=1}^{n}(w_{i,j} o_j) + b_i\Big) \qquad 1 \leqslant i \leqslant m, \tag{2.7}$$

to include the calculations for all m output neurons. This form of the equation enables the output calculations for a fully connected layer to be represented as the following matrix multiplication

$$\begin{bmatrix} o'_1 \\ o'_2 \\ \vdots \\ o'_m \end{bmatrix} = f_a\left(\begin{bmatrix} w_{1,1} & \cdots & w_{1,n} \\ w_{2,1} & \cdots & w_{2,n} \\ \vdots & \ddots & \vdots \\ w_{m,1} & \cdots & w_{m,n} \end{bmatrix}\begin{bmatrix} o_1 \\ o_2 \\ \vdots \\ o_n \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}\right). \tag{2.8}$$

Implementing the fully connected layers as in Equation 2.8 makes it possible to implement the forward propagation for such a layer with cuBLAS, and more specifically the GEMM function seen in Equation 1.1.

2.1.3.2 Convolutional Layer

For those neural networks whose training data consists of images, convolutional layers are very important building blocks. The structure of a convolutional layer differs a lot from the simple structure of a fully connected layer.

The first thing that differs is that both the input and output of a fully connected layer is often referred to as $m \times 1$ vectors. In convolutional layers both the input and output are of higher dimension, and are often referred to as three dimensional matrices, $m \times n \times c$, where c is the number of channels, m is the rows, and n is the columns.

Convolutional layers are built around something referred to as local connectivity. As opposed to fully connected layers, where every input neuron is connected to every output neuron, the output neurons of a convolutional layer are only connected to some of the input neurons. One purpose of having local connectivity is to make the neural network robust to the spatial structure of images.

The weights are defined as K three dimensional matrices, $m_f \times n_f \times c$, often called filters. Each output neuron has a group of input neurons associated with it, which are referred to as that output neuron's receptive field. The neurons in the receptive field are convolved with all filters, producing the pre-activation function output of the neuron. This procedure is done for all output neurons, finally creating the full pre-activation output of the layer.

The number of output neurons of a convolutional layer depends on four hyper parameters. The first one is called **depth**. This parameter essentially defines how many output neurons that are connected to the same receptive field, and is the same as the number of filters. The next two parameters are **horizontal stride**, s_w, and **vertical stride**, s_h. These parameters define how close intermediate receptive fields are, horizontally and vertically. The last parameter is called **zero-padding**, and defines how many zeros should be added to the borders of the input. An illustration of a convolutional layer can be seen in Figure 2.3.

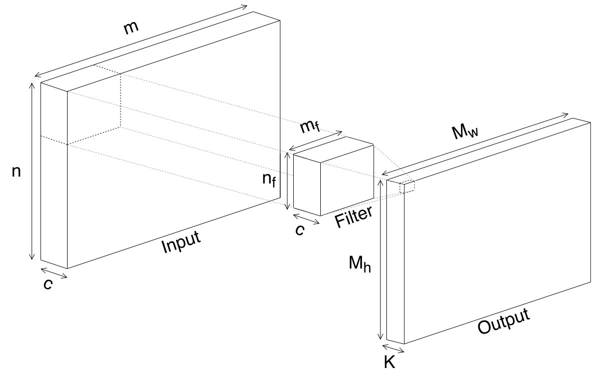


Figure 2.3: An illustration of a convolutional layer.

Just like the fully connected layers, the convolutional layers can be reshaped so that they are expressed as matrix multiplications, implemented with GEMM. The procedure is not as straight forward as for the fully connected layers, and is often used in order to optimize the speed of convolutional layers. This approach is not implemented in all deep learning frameworks, and is in this case specific for Caffe, without utilizing the cuDNN library, although a similar approach is used in cuDNN [Chetlur et al., 2014].

Given an input of size $m \times n \times c$ that is to be convolved with K filters of size $m_f \times n_f \times c$ with a stride of s_w and s_h, the first step is to represent the three dimensional input in two dimensions. This is done by splitting the input into M three-dimensional blocks, which are of the same size as the filters. Each block is then stretched out into a vector of size $1 \times m_f n_f c$, using a technique often referred to as image to column (im2col). The resulting input is now a $M \times m_f n_f c$ matrix.

The number of so called blocks, or M, depends on the previously mentioned parameters vertical stride, s_h, horizontal stride, s_w, and zero padding, p, and can be calculated as

$$M = M_w \cdot M_h = \tag{2.9}$$
$$= \Big(\frac{(n + 2 \cdot p) - n_f}{s_w} + 1\Big) \cdot \Big(\frac{(m + 2 \cdot p) - m_f}{s_h} + 1\Big).$$

If the stride in any direction is less than the corresponding filter size, this will lead to input data being duplicated, which eventually makes this technique a trade off, where speed is won at the cost of memory, thanks to the optimized implementations of matrix multiplication.

The filters are represented in a similar fashion. Every filter is streched out into a vector of size $m_f n_f c \times 1$ with im2col. These vectors combined form a matrix of size $m_f n_f c \times K$, which is the resulting weight tensor of the layer. The forward propagation can then be expressed as

$$\mathbf{O} = \mathbf{B} \cdot \mathbf{F} = \tag{2.10}$$

$$\begin{bmatrix} b_1^1 & \cdots & b_{m_f n_f c}^1 \\ b_1^2 & \cdots & b_{m_f n_f c}^2 \\ \vdots & \ddots & \vdots \\ b_1^M & \cdots & b_{m_f n_f c}^M \end{bmatrix} \begin{bmatrix} f_1^1 & \cdots & f_1^K \\ f_2^1 & \cdots & f_2^K \\ \vdots & \ddots & \vdots \\ f_{m_f n_f c}^1 & \cdots & f_{m_f n_f c}^K \end{bmatrix} \tag{2.11}$$

where b_j^i is element j in the im2col reshaped block i, and f_l^k is element l in the im2col reshaped filter k.

The resulting matrix $\mathbf{O}$ is reshaped back to the dimension $M_h \times M_w \times K$, before the biases are added. The activation function is then applied, producing the final layer output.

2.1.4 Max-Pooling

Max-pooling is a form of down-sampling, and is often used between a set of convolutional layers to reduce the amount of parameters.

The max-pooling is performed separately for each channel. Each output value corresponds to the maximum value of a $k_w \times k_h$ grid of input values. The size of the output depends both on the grid size, k_w and k_h, and the stride, s_w and s_h. The stride is a measure of the distance between adjacent grids.

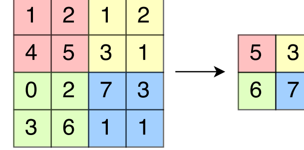


Figure 2.4: Illustration of input (left) and output (right) of the max-pooling function, with $k_w = k_h = 2$ and $s_w = s_h = 2$.

2.1.5 Overfitting

A risk when training neural networks is that they might get overfitted. Overfitting (or overtraining) is the term used when the network no longer learns general features, but instead focuses more and more on features specific to the training set. This means that although the model fits the training data very well, it does not generalize to other data. There are several methods of preventing this from happening, two of which are described below.

2.1.5.1 Dropout

One technique for reducing overfitting of a network is called dropout. Neurons are randomly – but with a pre-defined probability, called dropout rate – temporarily deleted. This forces the remaining neurons not to rely too much on the missing ones, and thus making the network more robust. In each batch, a different set of neurons are deleted.

2.1.5.2 L$_2$ Regularization

Another method of reducing overfitting is L$_2$ regularization (also called weight decay), which adds a regularization term to the loss function

$$L = L_0 + \frac{\lambda}{2N} \sum_w w^2, \tag{2.12}$$

where L_0 is the original loss function, and the regularization term is the sum of the squared weights, scaled by a regularization parameter, λ, over two times the batch size, N.

Adding the regularization term to the loss function leads to smaller weights being preferred over larger ones. This in turn reduces overfitting of the model.

2.2 VGG16

The base neural network architecture used in this thesis is VGG-16, as described in [Parkhi et al., 2015]. The network contains 13 convolutional layers and three fully connected layers. An illustration of the architecture is shown in Figure 2.5. Different colors correspond to different layer types, and for each layer in the network the layer name and output size is listed.

In both dropout layers, a dropout rate of 0.5 is used.

All convolutional layers in VGG-16 share the same kernel height and width, $m_f = n_f = 3$, the same stride, $s_w = s_h = 1$, and the same zero padding, $p_w = p_h = 1$.

The pooling layers also share the same format, with a grid size of 2×2, and a stride of $s_w = s_h = 2$.

The activation function used in VGG-16 is the ReLU function shown in Equation 2.2, and the loss function used is the softmax loss, as shown in Equation 2.4.

2.2.1 VGG Face

The Oxford Visual Geometry Group (VGG) provides, via the Caffe Model Zoo, a trained model of the VGG-16 network, called VGG Face [Parkhi et al., 2016]. VGG Face is trained to classify 2,622 identities, and has been trained on 2,6 million face images. The dataset used for training is described in more detail in Section 2.6.2.

η	α	λ	r_d	batch size
0.01	0.9	0.0005	0.5	64

Table 2.1: Values of the learning rate (η), momentum (α), regularization parameter (λ), dropout rate (r_d), and batch size in VGG Face.

During the training of VGG Face, the risk of overfitting was reduced by using both dropout and L_2 regularization. Dropout was performed before the last two fully connected layers, using a dropout rate of 0.5. For the L_2 regularization, $\lambda = 5 \cdot 10^{-4}$ was used as regularization parameter.

Table 2.1 shows all hyper-parameter values used. Twice during the training, when the accuracy stopped increasing, the learning rate shown in Table 2.1 was decreased by a factor of 10 [Parkhi et al., 2016].

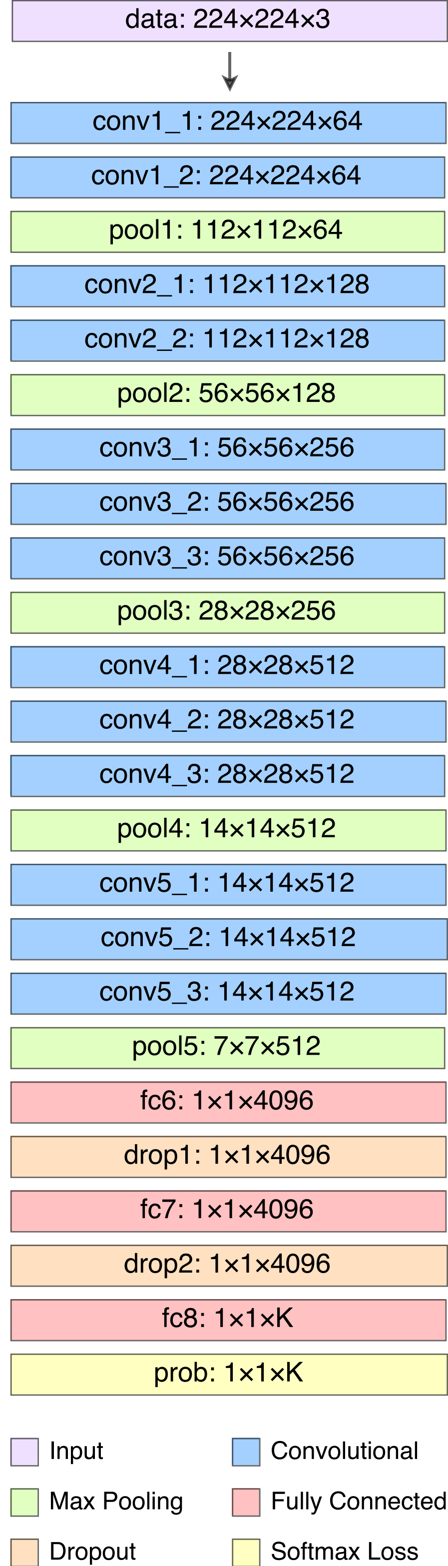


Figure 2.5: Illustration of the VGG-16 architecture. In the two bottom-most layers, K refers to the number of labels.

2.3 Transfer Learning

Having limited time and hardware, training a CNN from scratch was not an option for this Master's Thesis. Instead, features were transferred from the pre-trained VGG Face model, and only the last layers of the target model were trained.

In order to decide which layers to transfer, the different layers of VGG Face were analyzed. As discussed in Section 1.3.2, the first layers of CNNs contain the more simple, general features. Looking at Figure 2.6, visualizing the weights of the first convolutional layer in VGG Face, one can reach the conclusion that these filters react to simple corners and edges, supporting the previous claim.

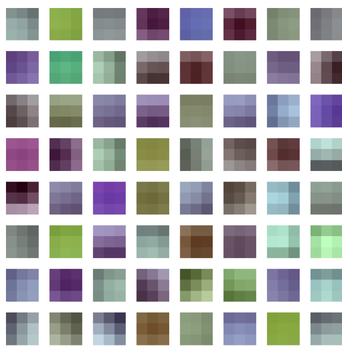


Figure 2.6: Illustration of the weights in the first convolutional layer of VGG Face.

In Figure 2.7, illustrating some of the outputs from the convolutional layers, it can be seen that the output gets more and more abstract the further into the network the layer is placed.

Intuitively, the majority of layers in VGG Face can be kept as-is, since the model has been trained with the purpose of classifying identities (and that ambition remains). However, since the last fully connected layer of VGG Face contains 2,622 outputs, each corresponding to one identity, that layer has to be adjusted in size and retrained to fit the new identities. A decision was made to use the weights from all convolutional layers of VGG Face as is. Three main training setups were explored:

Freezing the first two fully connected layers, fc6 and fc7, as well, training only the last layer, fc8, initialized with a Gaussian distribution with mean $\mu = 0$ and standard deviation $\sigma = 0.01$.

Fine-tuning the first two fully connected layers, fc6 and fc7, with a low learning rate, and training fc8 from scratch, initialized with a Gaussian distribution with $\mu = 0$ and $\sigma = 0.01$.

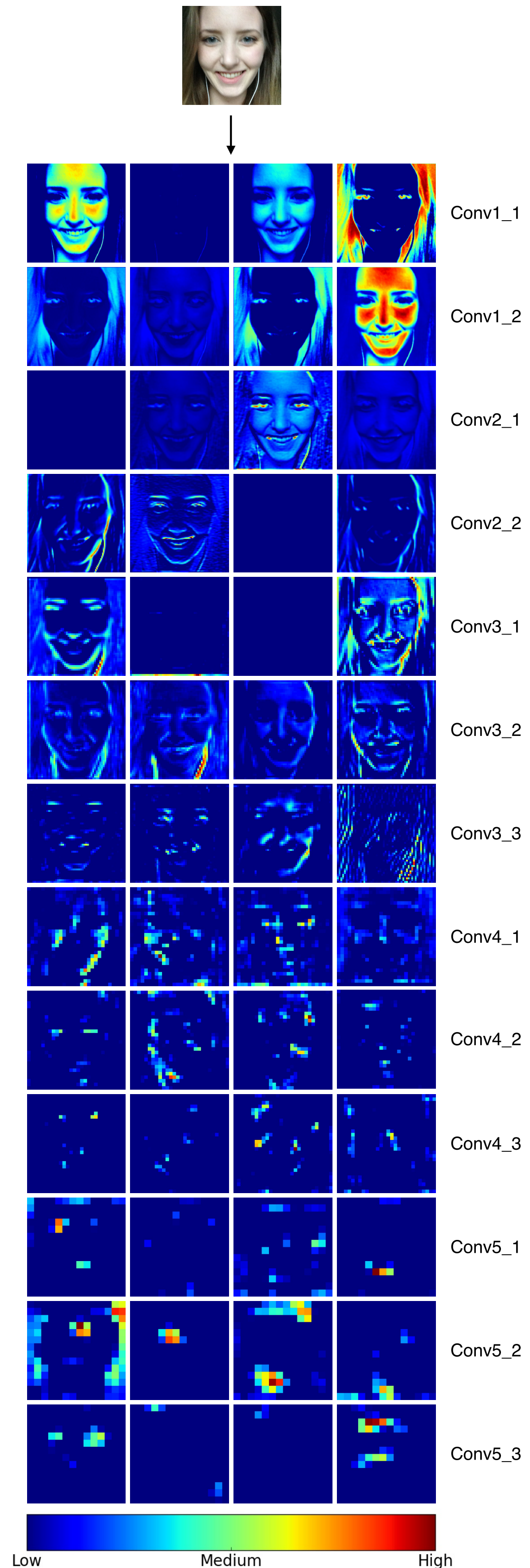


Figure 2.7: A visualization for (some of) the output from the convolutional layers of VGG Face. Each row of images corresponds to one layer.

Training all fully connected layers from scratch, initialized with a Gaussian distribution with $\mu = 0$ and $\sigma = 0.01$.

The models trained using the above configurations will be referred to as the *frozen model*, the *fine-tuned model*, and the *fc trained model*, respectively.

2.4 Preprocessing

Before feeding input to neural networks, some processing is often performed on the data. When it comes to images, the preprocessing generally involves resizing and mean-subtraction.

The layers in neural networks are designed to receive data of a certain size, which is why image resizing is often required. VGG-16 accepts RGB images of size 224×224, so when creating the datasets used for training and testing, resizing is performed on images whose sizes differ from 224×224.

The reason for performing mean-subtraction is to (on average) center the pixel values around zero, by subtracting the mean of all training images.

There are two ways of generating the image mean; channel-wise, and pixel-wise. For the channel-wise mean, the mean of each channel across all images is calculated. In this case, the mean consists of (in the case of three-channel images) only three values, leading to the same value being subtracted from all pixels in a channel. The pixel-wise mean however is calculated separately for each pixel and channel. Figure 2.8 shows examples of both channel-wise mean and pixel-wise mean.

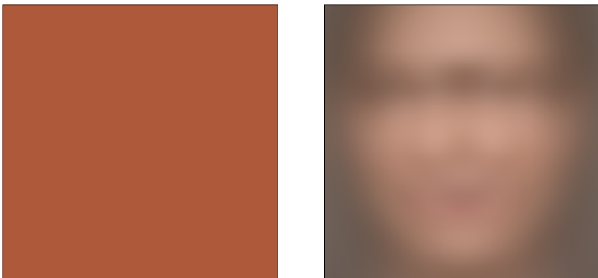


Figure 2.8: Channel-wise mean (left) used in VGG Face, and pixel-wise mean (right) of the curated combinatory dataset described in Section 2.6.

During transfer learning, it is not apparent whether the mean of the original training data or the new training data should be used. Since the original dataset is often larger, it can be argued that it is advantageous to use its mean. On the other hand, since the network is trained to fit new data, it seems reasonable that the mean being subtracted should originate from the new data. Since no evidence was found that one approach is better than the other, it was decided that a pixel-wise mean would be calculated from the dataset currently used for training.

2.5 Compressing Neural Networks

When it comes to deep learning, and neural networks, the term compression can refer to different variables. Below some of these are specified and described.

Number of parameters for the network. This includes all the weights and biases, and is closely related to both the offline storage, and the memory footprint.

Offline storage refers to the actual storage needed to represent the parameters of the neural network when saved on the hard drive. For this master's thesis all trained neural networks are stored in a binary file format.

Memory footprint is the memory needed when the neural network is deployed, and running. Both the main memory (RAM) and the graphics memory are included in this variable.

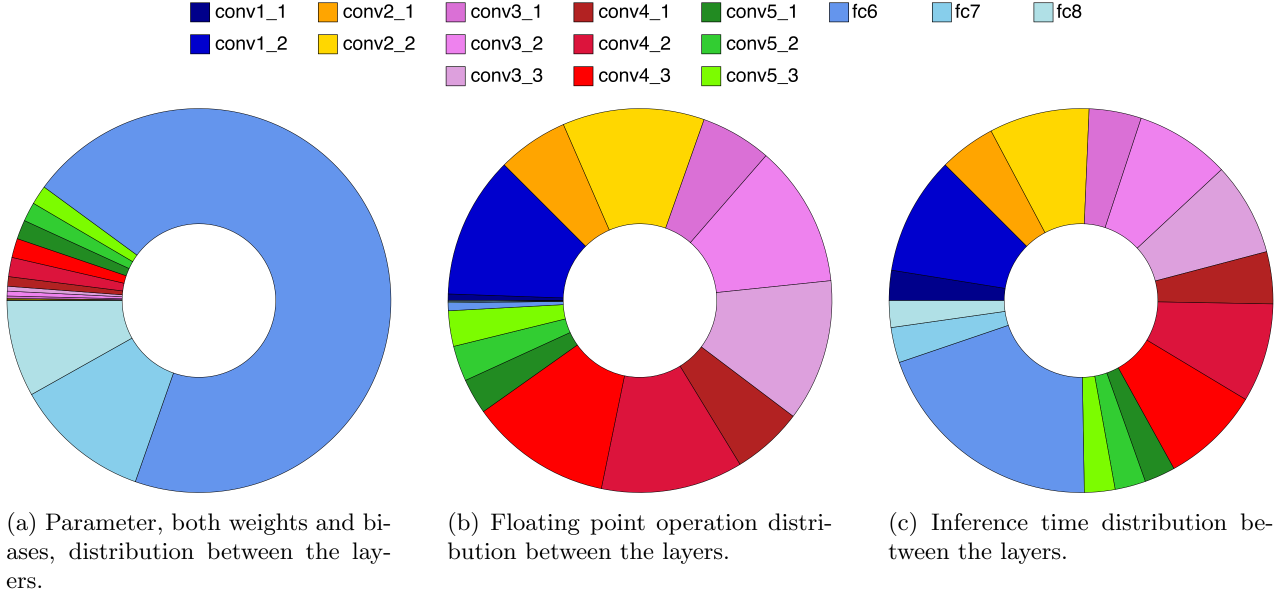
Floating point operations is the number of operations concerning floating point numbers that are executed during different phases of the neural network. For this master's thesis all additions, subtractions and multiplications involving floating point numbers are counted as one floating point operation each.

Time complexity refers to the run time of different phases of the neural network. The time complexity is related to both the memory footprint and the number of floating point operations. In this master's thesis the time complexity will be presented as real time, as opposed to using the big O notation ($\mathcal{O}$).

All these variables are of concern when working with neural networks, especially on embedded systems. The different ways that the layers are structured makes them susceptible to different compression algorithms.

An analysis of the VGG-16 network during inference shows that the different layers are very different when it comes to affecting the different variables. As can be seen in Figure 2.9a and 2.9b most of the parameters, around 90%, of the VGG-16 architecture are located in the fully connected layers, while around 99% of the floating point operations are executed in the convolutional layers.

This is a pattern that can be seen in many of the popular deep neural network architectures. From this a conclusion can be drawn that for compressing the



(a) Parameter, both weights and biases, distribution between the layers.

(b) Floating point operation distribution between the layers.

(c) Inference time distribution between the layers.

Figure 2.9: Illustration of how the 148 million parameters (a), 30.9 billion floating point operations (b), and the time (c), are distributed among the different layers of the fine tuned VGG-16 architecture. Both the number of floating point operations and the time are for one inference, using a batch size of one. (**Best viewed in color**)

number of floating point operations of a neural network, one should focus on the convolutional layers. On the other hand, for compressing the number of parameters, and eventually the memory needed for both storing and deploying the model, one should focus on the fully connected layers.

The time distribution between the layers for one inference, which can be seen in Figure 2.9c, is more evenly spread, and is a combination of both parameters, because of the overhead of memory transfer, and floating point operations.

2.5.1 Floating Point Precision Reduction

Section 1.3.3 mentions previous attempts to increase the speed, and decrease the memory, of a neural network by representing the parameters in the network with small data types, instead of the common 32 bit floating point data type.

Instead of using integers, or single bits, to represent the parameters in the neural network, another data type is used in this master's thesis, namely the data type called half.

The floating point data type used for the experiments conducted in this thesis are defined by the IEEE Standard for Floating-Point Arithmetic (IEEE 754). This standard specifies a float as one bit defining the sign, 8 bits defining the exponent, and 24 bits (23 explicitly stored) defining the significand precision. The format is illustrated in Figure 2.10a.

In general, calculating the decimal value given a float in bits is done as per the following equation

$$value_{10} = (-1)^{b_{31}} \cdot 2^{e-127} \cdot \left(1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}\right), \quad (2.13)$$

where e is the decimal value of the bits b_{23} - b_{30}.

The data type called half, also using the IEEE 754 definition, is represented by one bit defining the sign, 5 bits defining the exponent, and 11 bits (10 explicitly stored) defining the significand precision. The equivalent of Equation 2.13, but for the data type half is

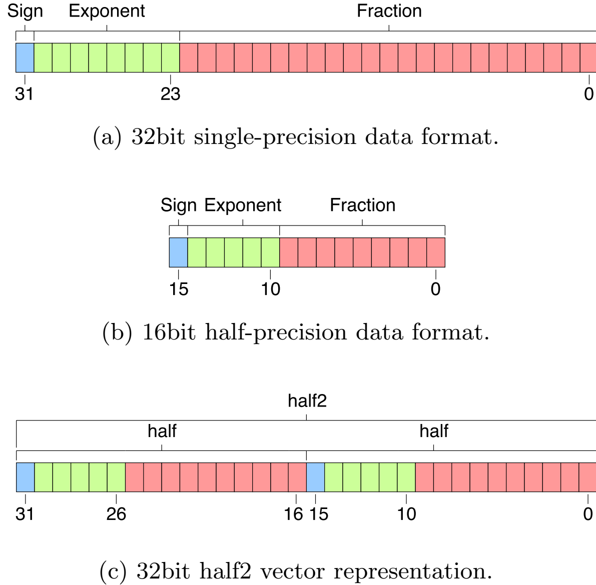
$$value_{10} = (-1)^{b_{15}} \cdot 2^{e-15} \cdot \left(1 + \sum_{i=1}^{10} b_{10-i} 2^{-i}\right). \quad (2.14)$$

From the previous equations and Figure 2.10 it can be concluded that some precision will be lost using the half data type instead of float.

In theory, changing to a data type represented by half as many bits as the original data type, might imply double the computation speed, and half the memory consumption. In practice however, when working with such basic arithmetic types, the improvements depend on the hardware support and data type implementation.

Out of the different hardware setups featured in Section 1.4.1, only the GPU architecture used in the Tegra X1 can fully utilize the power of using the half data type for computations [NVIDIA, 2015].

It is still possible to work with the half type on the other hardware setups, the data will be stored as half precision floating points numbers, and even



(a) 32bit single-precision data format.

(b) 16bit half-precision data format.

(c) 32bit half2 vector representation.

Figure 2.10: Images illustrating the different floating point data formats, where Figure (a) is the original **float** format, (b) is the **half** format, and (c) is the vector format for two **half**s, also referred to as **half2**.

transferred to the GPU as such, which will of course impact the bottleneck of the bandwidth, but when the computations are executed the halfs will be transformed into 32 bits and operated on as such.

The difference for the Tegra X1 system is that it supports computations with the half type by storing two halfs in a half2 vector as seen in Figure 2.10c, and sending that vector to a dedicated 32 bit CUDA core for calculations. This does however only work when the operations are the same for both the halfs.

For the experiments in this master's thesis concerning FP16, all training, and the following offline storing was done using 32 bit floats. Later during the deployment phase, specifically for the Tegra X1, the floats were converted into halfs.

2.5.2 Pruning

In Section 1.3.3 two published papers, with different approaches to pruning were presented. The actual idea of neural network pruning is related to biological neural networks, much like the whole concept of neural networks, where the number of synapses in the human brain increases at an early age, and later decreases to finally converge [Walsh, 2013].

The first thing done when exploring the actual implementation of pruning, was to split the problem into two separate sub problems; the first being to find which weights should be pruned, and the second being how to represent the weight tensor after the pruning had been performed.

2.5.2.1 Selection of Weights to Prune

The basic idea is that the output of a neuron in the previous layer, multiplied with a weight that has a value of zero, will not have any impact on the corresponding neuron in the current layer, and the weight can therefor be seen as a dead weight, and can be safely removed. However, it is very unlikely that a trained neural network will have any dead weights.

Instead dead weights are defined depending on other premises, and can vary for different pruning approaches. For instance, a weight and neuron product will have less of an impact on the neuron in the next layer the closer the weight is to zero. The same pattern can be applied to weights in relation to the other weights in that layer. The weights might not be distributed around zero, but around an arbitrary number.

In accordance with the previous statements, three different pruning approaches are explored. The first one takes no regards to different weight distributions in the layer, the second one prunes weights relative to the full layer weight distribution, and the last one prunes weights in regard to weight distribution for individual neurons.

In the definition of all these approaches a pruning parameter, γ, is introduced, and the weight matrix will be denoted as $\mathbf{W}$. Following this notation and Equation 2.8 for fully connected layers, means that $\mathbf{W}_{ij}$ is the weight from neuron j in the previous layer, to neuron i in the current.

For convolutional layers $\mathbf{W}$ represents the matrix $\mathbf{F}$ in Equation 2.10. This means that $\mathbf{W}_{ij}$ will be element i in the im2col reshaped filter j.

The first approach simply follows the theory that the closer to zero a weight value is, the less of an impact it has, and can therefor be removed. This can be written as

$$\mathbf{W}_{ij} = \begin{cases} 0, & \text{if } |\mathbf{W}_{ij}| \leqslant \gamma \\ \mathbf{W}_{ij}, & \text{otherwise} \end{cases}, \tag{2.15}$$

which will in fact remove the weights that affect the next layer, and in theory the end accuracy, the least.

The second approach involves pruning the weights that affect the next layer the least, in relation to the distribution of the weights, as follows

$$\mathbf{W}_{ij} = \begin{cases} 0, & \text{if } |\mathbf{W}_{ij}| \leqslant \gamma \cdot \sigma(\mathbf{W}) + \mu(\mathbf{W}) \\ \mathbf{W}_{ij}, & \text{otherwise} \end{cases}, \tag{2.16}$$

where $\sigma(\mathbf{W})$ is the standard deviation of the weight matrix $\mathbf{W}$, and $\mu(\mathbf{W})$ is the mean of $\mathbf{W}$.

The last approach differs between the fully connected layer and the convolutional layers. In the case of fully connected layers it is similar to the previous

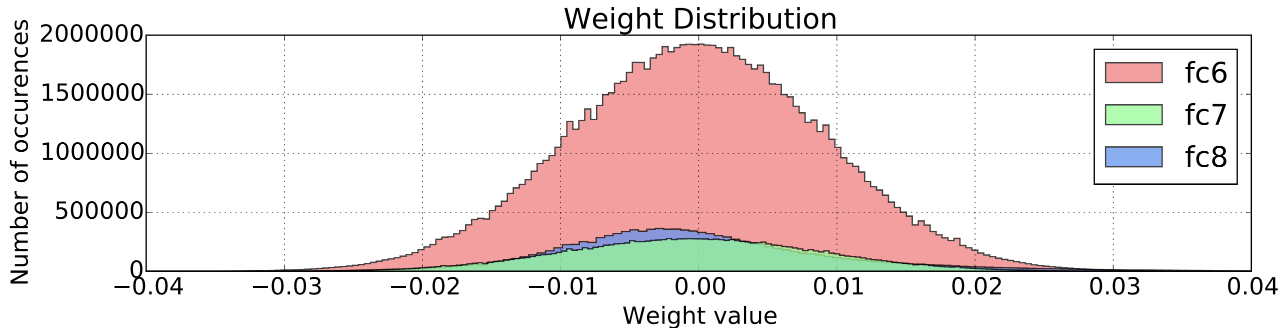


Figure 2.11: Weight distribution for the different layers in the VGG Face.

approach, and can be written as

$$\mathbf{W}_{ij} = \begin{cases} 0, & \text{if } |\mathbf{W}_{ij}| \leqslant \gamma \cdot \sigma(\mathbf{W}_i) + \mu(\mathbf{W}_i) \\ \mathbf{W}_{ij}, & \text{otherwise} \end{cases}. \tag{2.17}$$

In this equation $\sigma(\mathbf{W}_i)$ is the standard deviation of row i in $\mathbf{W}$, which essentially means the standard deviation of all the weights from the previous layer to neuron i in the current, and the same applies to μ.

For convolutional layers the threshold is based on the standard deviation and mean of each output neuron's receptive field. Using the matrix representation in Equation 2.10 then results in the following equation

$$\mathbf{W}_{ij} = \begin{cases} 0, & \text{if } |\mathbf{W}_{ij}| \leqslant \gamma \cdot \sigma(\mathbf{W}_j) + \mu(\mathbf{W}_j) \\ \mathbf{W}_{ij}, & \text{otherwise} \end{cases}. \tag{2.18}$$

Figure 2.11 illustrates the weight distribution for the fully connected layers in the original VGG Face model. As can be seen the weights for both fc6 and fc7 are distributed around zero. For fc8 however, the weight distribution has a slight offset, and is instead centered around 0.003.

After a neural network has been pruned, it can be retrained. This is often done to tune the parameters that are left, making them compensate for the weights that were removed. There are mainly two approaches to training a pruned network.

The first one is to simply prune the network once, quite aggressively, and then retrain it. The other way is referred to as iterative pruning. As the name implies, a second training phase is added. The pruning is then executed iteratively during the second training phase, using a smaller pruning parameter.

For this master's thesis only the first approach of retraining was explored. The retraining for layers was implemented by representing the pruned weight tensor, $\mathbf{W}$, as a dense matrix were all the pruned weights had a value of 0.0. When updating the weights during back propagation, only the un-pruned weights were updated.

Implementing the training of pruned layers in this way does not lead to any compression during the training phase, but rather the opposite. In the next section a weight representation after pruning will be presented, that does compress the neural network during deployment.

2.5.2.2 Weight Representation After Pruning

Recall from Equations 2.8 and 2.10 that the forward propagation of a fully connected layer, as well as a convolutional layer, can be written as a matrix multiplication. This is also the case for when the layers have been pruned, the only difference being that the previously dense matrix $\mathbf{W}$ now contains multiple zeros, also referred to as dead weights.

As with pruning, there exists different approaches for representing the weight matrix post-pruning. Section 1.3.3 mentions articles concerning this topic. Implementing pruning featuring a weight matrix mask has been briefly explored for this master's thesis. However, since the main objective of the pruning was reducing the memory footprint of the neural network, both offline and when deployed, and secondly improving computation speed, masking was not a suitable alternative.

The chosen implementation instead features the use of sparse matrices, and more specifically the compressed sparse row (CSR) formatted sparse matrix. A sparse matrix, $\mathbf{A}$, in this format is represented by one vector, $\mathbf{A}_{val}$, containing the non-zero values, another vector, $\mathbf{A}_{row}$, where the first element is zero and the exceeding elements are the accumulated number of non zero values per row, and a third vector, $\mathbf{A}_{col}$, containing the column indices of the non-zero matrix elements.

The following $m \times n$ sparse matrix in dense format

$$\mathbf{A} = \begin{bmatrix} a & 0.0 & 0.0 & b & 0.0 \\ c & 0.0 & d & 0.0 & 0.0 \\ 0.0 & e & f & 0.0 & g \\ h & 0.0 & 0.0 & i & 0.0 \end{bmatrix}, \tag{2.19}$$

where $m = 4$ and $n = 5$, will therefor in CSR format, using zero-based indexing, be represented as the three vectors

$$\mathbf{A}_{val} = \begin{bmatrix} a & b & c & d & e & f & g & h & i \end{bmatrix}, \tag{2.20}$$

$$\mathbf{A}_{row} = \begin{bmatrix} 0 & 2 & 4 & 7 & 9 \end{bmatrix}, \tag{2.21}$$

$$\mathbf{A}_{col} = \begin{bmatrix} 0 & 3 & 0 & 2 & 1 & 2 & 4 & 0 & 3 \end{bmatrix}. \tag{2.22}$$

Both the vectors $\mathbf{A}_{val}$ and $\mathbf{A}_{col}$ will always have a size equal to the number of non-zero elements in matrix $\mathbf{A}$, while $\mathbf{A}_{row}$ will have $m + 1$ elements.

The overhead, both time and memory related, of using sparse matrices is quite noticeable, especially when working with unstructured sparse matrices, as in the case of pruning.

A constraint on the pruning is added as a consequence of the sparse matrix overhead. A minimum number, α, of weights need to be pruned in order for the pruning to actually have a positive effect on the memory footprint and execution speed of the neural network.

The experiments in this Master's thesis were conducted using 32 bit integers for the values in A_{row} and A_{col}, and all floating point numbers are represented in single precision for pruned layers. This means that

$$\alpha > \frac{m \cdot n}{2} - (m + 1), \tag{2.23}$$

where α is the number of weights pruned, m and n are the number of weight matrix rows and columns correspondingly, needs to be true for the pruning to decrease the memory footprint.

Calculating the least number of weights that need to be pruned for a speed increase is a little more complicated, since it depends on more parameters. The speed increase does however correlate to the memory decrease, since it is very time consuming in GPU programming to read data from global memory. Figure 2.12 illustrates the overhead for both memory and speed.

The data for Figure 2.12 was collected by doing multiple sparse matrix and vector multiplications, with a varying number of elements zeroed, and comparing the time and memory footprint with a dense matrix and vector multiplication. As can be seen in the figure, the time correlates to the memory, and the

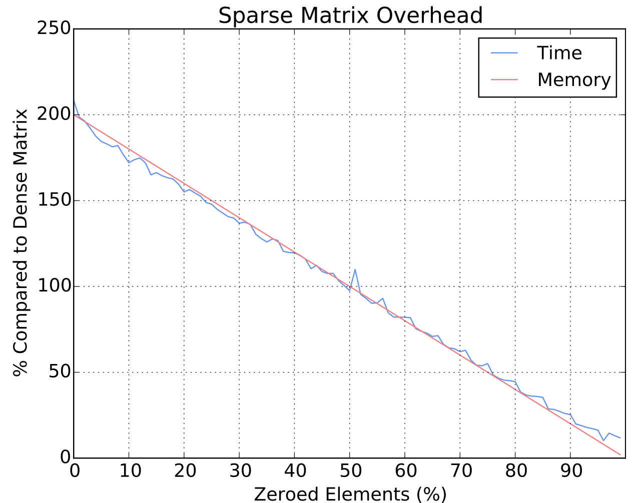


Figure 2.12: Illustration of the CSR overhead. The graph shows the computation time and memory footprint of a sparse matrix and vector multiplication in relation to a dense matrix and vector multiplication.

overhead makes anything above 50 % zeroes infeasible.

The pruned weight matrix can also be represented in CSR format when storing the model offline, and this approach can therefor also lower the offline memory needed for the model.

To conclude, using the sparse matrix approach and one of the pruning approaches mentioned in Equations 2.15, 2.16, and 2.17, given that a little more than 50 % of the weights in a layer can be pruned, a decrease in both time and memory for the deployed neural network can be achieved.

2.6 Datasets

The quality of neural networks greatly depends on the data they are trained on. One of the easiest (and in some ways also the hardest) ways of improving the performance of a neural network is to train it on large quantities of data, with large amounts of samples per class.

The reason for it being mentioned as an easy way of improving the performance is that no parameters need to be manually tweaked, the only thing that is done is adding more data. However, collecting this data can be extremely time-consuming.

As collection of large datasets is out of the scope for this thesis, pre-existing sources of data were relied upon. All datasets used consisted of images in so called uncontrolled settings.

During the transfer learning described in Section 2.3, and the other experiments, different combinations of the following datasets where used. Infor-

Name	Datasets	Identities	Images
combinatory	VGG, FaceScrub*, Additional	2,968	2,287,022
curated combinatory	VGG curated, FaceScrub*, Additional	2904	908,041
extended FaceScrub	FaceScrub, Additional	346	77,523

Table 2.2: Different dataset combinations used during training. *only identities that do not overlap those of VGG Face.

mation regarding the different combinations can be found in Table 2.2.

2.6.1 Labeled Faces In The Wild

Labeled Faces in the Wild (LFW) is a dataset widely used for training and testing face verification algorithms. It contains images of people in so-called uncontrolled settings, meaning that variables such as light, pose, and expression differ among the images [Huang et al., 2007]. In the dataset 1,680 identities have two or more images associated with them, but only 12 identities contain more than 50 images. In this sense, the dataset is not optimal when training a neural network for face recognition.

2.6.2 Oxford VGG Face Dataset

The dataset used for training the VGG Face model described in Section 2.2.1 was collected from the internet, using a face detector, and filtered both automatically and manually. The dataset contains 2,622 identities, with 1,000 images per identity, making a total of 2,622,000 images.

The images are collected from multiple sources, and vary in both quality and content. The dataset can therefor, like LFW, be viewed as a collection of images captured in uncontrolled settings.

Wishing to perform transfer learning of VGG Face on this dataset, in addition to others, the dataset needed to be obtained. A file containing links to all images, along with coordinates from a face detector, were used to download and crop the images. However, a large amount of the links were broken, resulting in less images than the original dataset. In total, 2,209,218 images were downloaded, belonging to the 2,622 identities, with an average of 843 images per identity.

In addition to the problem with the broken links, it appears that the coordinates from the face detector were not always correct. Also, in some cases an image supposed to depict one person clearly depicts someone else. It is not known whether the last two issues are due to the images having been replaced on the web since downloaded by the Visual Geometry Group, or if all these 'faulty' images were actually used in training.

A subset of the links were marked as 'curated' by the Visual Geometry Group. For this curated subset, additional effort was put into manual filtering, making it approximately 95% pure. The downloaded part of the curated set contained 651,423 images, depicting 2,558 identities.

When the VGG Face model was originally trained, both the curated and the non-curated datasets were used. Surprisingly, the non-curated training set produced better results than the curated one [Parkhi et al., 2015]. The authors claim two reasons for this; that a larger dataset is preferable over a smaller one, even if it contains more noise, and that good images are removed along with the bad during curation, reducing the number of pure training images.

2.6.3 FaceScrub

The FaceScrub dataset has over time been adjusted, and in its current form it comprises of 106,863 images for a total of 530 identities. 265 of those identities are men, with a total of 55,306 images, while 265 are women with a total of 51,557 images.

The dataset was collected through crawling the internet, and using a data-driven approach to clean the enormous amount of downloaded images [Ng and Winkler, 2014].

For this thesis, only part of the FaceScrub dataset was acquired, with a total of 77,323 images and 529 identities. Out of these identities, 343 overlap the identities in the VGG Face dataset. When the two sets were combined for use in training, only the non-overlapping identities from FaceScrub were used.

2.6.4 Additional Dataset

Following the procedures in Section 2.7, a dataset of Axis employees was collected. When conducting the experiments described in this thesis, the dataset contained only three identities, and a total of 481 face images.

Later, when the final model was trained, solely for the purpose of the demo, five more identities, and 316

images, had been added to the set.

2.7 Dataset Collection

In the interest of fully showcasing the demo described in Section 2.9, a dataset of Axis employee faces was collected. As a first step, the authors collected 100 preexisting photos of themselves. Next, the employees of the department, and all thesis workers at Axis, were asked to contribute with 50 images each.

The images were cropped according to the bounding boxes found by the face detector described in Section 2.9.2.2, and added to the dataset used for fine-tuning VGG Face.

2.8 Data Analysis

Collecting training data is often a long and tedious process, and it can be challenging to get a hold of the data. For this reason, it is of interest to learn how much data is required to train a neural network, so that it results in a well performing model. However, there is no gold standard for how much data is needed. Not only is it affected by the quality of the data, but it varies from one network to another.

2.8.1 Data Quantity and Augmentation

With the purpose of investigating how many images per identity are needed in order to fine-tune the VGG Face model such that it produces an acceptable accuracy, experiments were performed.

The dataset used consisted of 454 identities, from both FaceScrub and the additional dataset, which all contained 100 images per identity or more. The test set consisted of 20 images from each identity, all selected randomly. The size of the training sets varied, from 5 images per identity to 80. First, 80 images from each identity were selected randomly for training, and the smaller training sets were all subsets of these.

As discussed in Section 1.3.4, data augmentation can be used to improve the performance of neural networks, by artificially expanding the training data, as well as to reduce overfitting.

As a way of extending the experiment, all image sets were augmented, expanding each set by 500%. Each image was augmented using five different techniques:

Adding Gaussian noise to the image. The noise consists of random values from a Gaussian distribution, with mean $\mu = 0$ and standard deviation $\sigma = 80$, constrained to the value range of the original image.

n_i^o	n_i^a	**iter./image**	**iter. total**
5	0	60	13620
5	25	10	13620
10	0	60	27240
10	50	10	27240
20	0	60	54480
20	100	10	54480
40	0	60	108960
40	200	10	108960
80	0	60	217920
80	400	10	217920

Table 2.4: Training setups used for the data quantity experiments in Section 2.8.1. n_i^o refers to the number of original images per identity, and n_i^a refers to the number of additional images used per identity, produced by augmentation. Iter. refers to training iterations.

Adding Gaussian blur to the image, with a kernel size of $k_w = k_h = 3$, and standard deviation $\sigma = 0.8$.

Adding salt and pepper noise to the image. A matrix of the same size as the input image, containing uniformly distributed random numbers in the range $[0, 255)$ is generated. At the positions where the random matrix has values < 20, the corresponding values of the image are set to 0, and at the positions where the random matrix contains value > 235, the corresponding pixels of the image are set to 255.

Mirroring the image – simply flipping the image horizontally.

Applying adaptive histogram equalization to the L channel of the image, after converting it to the Lab color space.

The channel is divided into a grid of 8×8 tiles, after which histogram equalization is applied to each block. If a histogram bin is above the contrast limit, in this case 4, the pixels above the limit are redistributed to other bins, uniformly, before applying the equalization. This is done in order to prevent noise amplification.

Lastly, bilinear interpolation is applied to remove artifacts between tiles, and the image is converted back into RGB color space.

Figures 2.13b to 2.13f show the results of the different augmentation techniques, given the image in Figure 2.13a as input.

η	$\eta^*_{conv1_1} - \eta^*_{conv5_3}$	η^*_{fc6}	η^*_{fc7}	η^*_{fc8}	α	λ	r_d	Batch size
0.001	0.0	0.1	0.1	1.0	0.9	0.0005	0.5	10

Table 2.3: Values of base learning rate (η), layer-specific learning rate multipliers (η^*_{layer}), momentum (α), regularization parameter (λ), dropout rate (r_d), and batch size, used in the experiments of Section 2.8. Multiplying the base learning rate with a layer-specific learning rate multiplier results in that layer's learning rate.

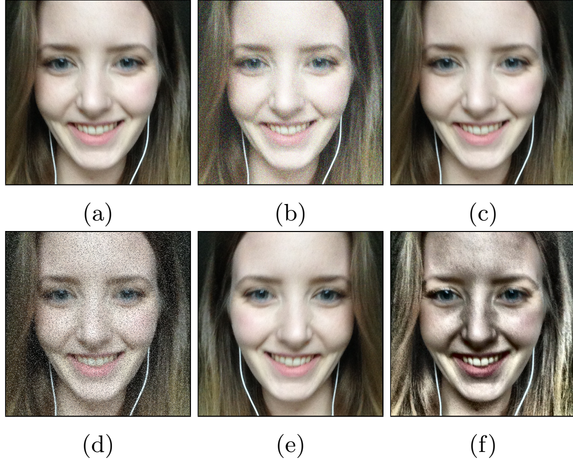


Figure 2.13: Resulting images from the different augmentation techniques; (a) original, (b) Gaussian noise, (c) Gaussian blur, (d) salt and pepper noise, (e) mirroring, and (f) adaptive histogram equalization.

The motive behind adding Gaussian blur to the images was to force the network to focus on more high level features. Opposite to this, the purpose of the adaptive histogram equalization was to enhance the fine-grained features.

Both the Gaussian noise and the salt and pepper noise can be compared to dropout in neural network training. Random pixel values are altered, forcing network weights not to rely too much on the neighboring weights.

The training data setups used for this experiment are listed in Table 2.4, and the parameter settings common for all setups can be found in Table 2.3.

2.8.2 Input Image Size

As mentioned in Section 2.4, images are re-sized before being passed as input to VGG-16. The network accepts RGB images of size 224 × 224, which means that all images smaller than that have to be expanded, and all that are larger have to be reduced in size.

An interest was taken in how the size of test images affects the accuracy of a trained model, and in pursuance of the answer an experiment was conducted.

The VGG Face model, transfer learned on the extended FaceScrub dataset, listed in Table 2.2, with all convolutional layers frozen, and all fully connected layers trained from scratch, was used together with the 622 images larger than 224 × 224 from the validation set as a baseline. The accuracy for the original version of these test images was 99.1961%.

The 622 validation images were downsampled to various fixed sizes, the largest being 224 × 224, and used for testing the model. The resulting accuracy was then noted, and compared to that of the original images.

2.9 Prototype

The prototype built for this thesis is a camera demo written in C++. It consists of three major parts; video capture, detection, and recognition. The user specifies models to use for both detection and classification, which means that the demo can perform other tasks than face recognition.

2.9.1 Video Capture

The camera demo can be run in either camera mode, or test mode. In the former, images are fetched from a video camera specified by the user. The camera used during the development of the system is an AXIS A8004-VE Network Video Door Station.

A threaded approach was used to implement the frame fetching from the camera. One thread continuously fetches images from the camera, only storing the latest one. In this way, when images are to be processed for detection and recognition on the main thread, the latest image is already stored locally, and can be processed instantly.

When running the camera demo in test mode, all images placed in a test image folder are loaded and processed, one at a time.

2.9.2 Detection

Whether in camera mode or test mode, the user specifies whether to perform detection on the images or not. There are currently two (face) detection algorithms to choose from. If one or more detections are made in an image, the crops containing the detections are forwarded to the recognition part of the program.

2.9.2.1 Feature-based Cascade Classifier

One of the detection algorithms available is the OpenCV implementation of Viola-Jones detection. The Viola-Jones detection uses three simple types of features, often referred to as Haar features, illustrated in Figure 2.14. The features are obtained by subtracting the sum of pixel values under the white rectangles from the sum of pixel values under the black rectangles, resulting in a single value [Viola and Jones, 2001].

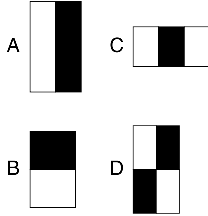


Figure 2.14: Examples of two-rectangle (A and B), three-rectange (C), and four-rectangle (D) Haar features.

Before calculating the features, Viola and Jones used so-called integral images as intermediate representations of the images, resulting in substantially faster feature calculations.

Instead of calculating all possible features for an image, a subset of them are chosen to form the classifier, using a version of AdaBoost, making the classification even faster [Viola and Jones, 2001]. Another method used to speed up the detection is using a cascade of classifiers. The early classifiers, which are simple and fast, are used to quickly filter out the image regions which do not contain the object of interest, while the more complex classifiers are only applied to regions which have a higher probability of containing the class object. Only regions which pass all classifiers are then considered positive regions.

2.9.2.2 Deformable Part Models

The other available detection algorithm is an implementation of the paper Exact Acceleration of Linear Object Detectors [Dubout and Fleuret, 2012], taking deformable part models (DPM) as input. The code is freely available at [Idiap, 2012] under the GNU General Public License Version 3 [Free Software Foundation, 2007].

DPM takes into account intra-class variability, by using class models made up of different components. The components are made from HOG features (Histogram of Oriented Gradients descriptors). When calculating HOG features, an image is divided into small cells. In each cell, the histogram of gradient directions are accumulated over the pixels. The gradients of an image region I are computed by performing convolution with the vertical and horizontal discrete derivative filters, as

$$\nabla I = (I_x, I_y) = (I * D_x, I * D_y), \tag{2.24}$$

where

$$D_x = [-1, 0, 1] \quad \text{and} \quad D_y = [-1, 0, 1]^T. \tag{2.25}$$

The magnitude of the gradient is computed as

$$\|\nabla I\| = \sqrt{I_x^2 + I_y^2} \tag{2.26}$$

and its orientation is

$$\Theta = \arctan \frac{I_y}{I_x}. \tag{2.27}$$

In the implementation used, these calculations are sped up by the use of Fourier transforms. Then, after contrast normalizing all cells, the resulting histograms together form the final descriptor [Dalal and Triggs, 2005].

2.9.3 Recognition

The recognition part of the camera demo is coupled with Caffe. The user specifies which network to use, which model to load weights from, and which file to read labels from. When the demo is first started, all parameters are loaded into memory, and remain there throughout the entire execution. It is up to the user whether Caffe should run in GPU or CPU mode.

The user has two options when it comes to the batch size used in the forward propagation. Either one detected object is forward propagated at a time (batch size=1), or all detected objects in an image are forwarded in one batch (batch size=number of detected objects).

When the classification is done, the five labels with the highest scores are printed to the standard output, along with their scores, and the image is displayed using OpenCV. In case detection is used, a frame is placed around each detection, and the top prediction label is printed below it, along with the output score. Additional information printed on each frame is the number of frames per second (fps), classification time, and detection time.

3 Results and Discussion

All the results presented in this section are from experiments using the retrained VGG-16 architecture, unless stated otherwise. The validation set used for tests concerning the total accuracy of the network consists of 34,925 images of 2,904 identities, from the curated combinatory dataset listed in Table 2.2.

For all training setups, the learning rates (shown in Table 2.3 and Table 3.2) were lowered with a factor of 10 after a third of the training was complete, and again after two thirds.

3.1 Transfer Learning

Wanting to use as much training data as possible, the VGG Face dataset and Facescrub were merged, along with the additional dataset, using only the non-overlapping identities from FaceScrub. The VGG Face dataset was used both in its original form, and the curated. These datasets are referred to as the combinatory dataset, and the curated combinatory dataset (see Table 2.2).

As mentioned in Section 2.6.2, the curated VGG Face dataset did not produce as good results as the non-curated, when the VGG Face model was originally trained by the Oxford Visual Geometry Group [Parkhi et al., 2015]. In order to quickly decide whether the combinatory dataset, or the curated combinatory, should be used for training the final model, a comparison was made.

The datasets were used one at a time to transfer learn the VGG Face model. The hyper-parameters listed in Table 3.1 were used in both cases. To make a fair comparison, both models were trained the same amount of iterations *per image*, as opposed to the same total amount of iterations.

$\boldsymbol{\alpha}$	$\boldsymbol{\lambda}$	$\boldsymbol{r_d}$	**Batch size**
0.9	0.0005	0.5	32

Table 3.1: Values of momentum (α), regularization parameter (λ), dropout rate (r_d), and batch size, used while transfer learning.

The final and best accuracy of the curated combinatory model was 90.34%, while the model trained with the combinatory dataset achieved only 78.05% accuracy. The models were trained for a total of 238,770 and 600,000 iterations respectively.

These results differ somewhat from [Parkhi et al., 2015]. A possible explanation for this could be that while the un-curated VGG Face dataset worked well for training the original VGG Face model, it contains too much noise to transfer learn a well performing classifier.

Another reason could be that the downloaded VGG Face dataset actually contained more noise than the original, due to displaced links. Since the un-curated dataset contained more noise than the curated set to begin with, the added noise from displaced links could have made the proportion of noisy images too large to outperform the smaller – but cleaner – dataset.

Due to the results of this experiment, the dataset used for comparing the different transfer learning setups was the curated combinatory dataset.

The curated combinatory dataset was randomly split into three parts; training, test, and validation. The training set contained 80% of all images, the test set was made up of 16% of the images, and the validation contained 4% of the total number of images. Care was taken to make sure all sets contained images of all identities.

The test set was used during training. At predefined intervals, tests were performed using all images in the test set, to see how the accuracy of the models progressed. The division of the curated combinatory dataset into training, test, and validation sets was performed in order to double check that the hyper-parameters and training setups were not tailored for the test set, by testing the models with the validation set, after completed training. As shown in Figure 3.1, the validation accuracy and the test accuracy are analogous.

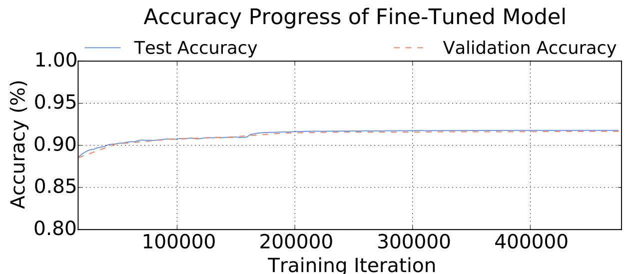


Figure 3.1: Test and validation accuracy of the fine-tuned model, during different stages of training.

As mentioned in Section 2.3, the transfer learning was performed using three different setups. The parameters common for all setups are shown in Table 3.1.

Common for all setups was also that the weights from all convolutional layers were transferred from VGG Face, and kept as is. Table 3.2 shows the learning rates of the remaining layers for the different setups.

When discussing the resulting models, the *frozen model* will refer to the result from the first setup, that is transferring and freezing fc6 and fc7, and train-

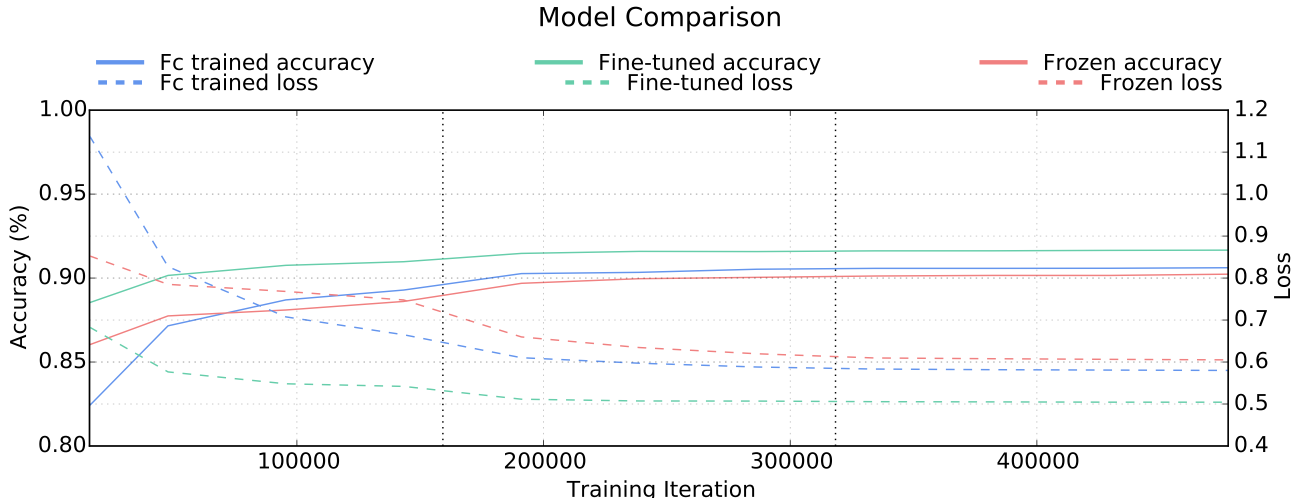


Figure 3.2: Accuracy and loss of the fc trained model, fine-tuned model, and the frozen model, shown at different training iterations. The two black dotted vertical lines mark a 90% decrease of the learning rate, starting at $\eta = 0.001$.

	η	η^*_{fc6}	η^*_{fc7}	η^*_{fc8}
frozen	0.001	0.0	0.0	1.0
fine-tuned	0.001	0.1	0.1	1.0
fc trained	0.001	0.1	0.1	1.0

Table 3.2: Values of base learning rate (η) and layer-specific learning rate multipliers (η^*_{layer}) for the frozen, fine-tuned, and fc trained setups. Multiplying the base learning rate with a layer-specific learning rate multiplier results in that layer's learning rate.

ing fc8 from scratch. The model resulting from the second setup, where fc6 and fc7 are transferred and fine-tuned, will be called the *fine-tuned model*. Finally, the model for which all fully connected layers are trained from scratch, will be referred to as the *fc trained model*.

In order to compare the different models, the validation set was used to test the performance of models saved during different stages of the training. Figure 3.2 includes plots of the accuracy and loss of all three models.

Throughout the entire training, the best accuracy is achieved by the fine-tuned model, with a peak accuracy of 91,66%. The loss is also the lowest for the fine-tuned model, at all times.

The frozen model starts out with a better accuracy and loss than the fc trained model, but gets outperformed by the fc trained model after about 90,000 iterations, when it comes to both accuracy and loss.

In the frozen model, both fc6 and fc7 were initialized with weights from the VGG Face model, while in the fc trained model they were initialized from a Gaussian distribution.

The results make it clear that the best option is to initialize fc6 and fc7 with weights from the VGG Face model, and that they should not be frozen, but instead allowed further training.

After approximately a third of the total training time, the base learning rate, shown in Table 3.2, is multiplied by 0.1. The learning rate is again decreased after a sixth of the training time. Starting at $\eta = 0.001$, this means that the final learning rate is only 10^{-5}. As seen in Figure 3.2, the accuracies of all models saturate somewhat after 300,000 iterations, which is around the time when the learning rate is decreased for the second time.

The reason for decreasing the learning rate is to allow for smaller and smaller adjustments of the weights and biases. Generally, the learning rate is lowered at times when the accuracy stops increasing, hoping to be able to increase it by making smaller modifications. Judging from the saturating accuracy in Figure 3.2, the models could have benefited from having the first learning rate decrease occur at a later time. Not knowing beforehand when the accuracy would stop increasing, a choice was made to lower the learning rate after one and two thirds of the total number of training iterations.

Having said that, no optimization of the hyper-parameters was attempted in this thesis. Instead, they were kept fixed for all set-ups, allowing for comparison of transfer-learning approaches. With some effort put into tweaking the hyper-parameters, it is likely that better accuracy could be achieved.

Figure 3.3 shows the per-batch loss of the fine-

tuned model during training, along with the average loss. The initial loss is 9.98956, but it decreases rapidly. From the average loss it is clear that the loss keeps decreasing throughout the training.

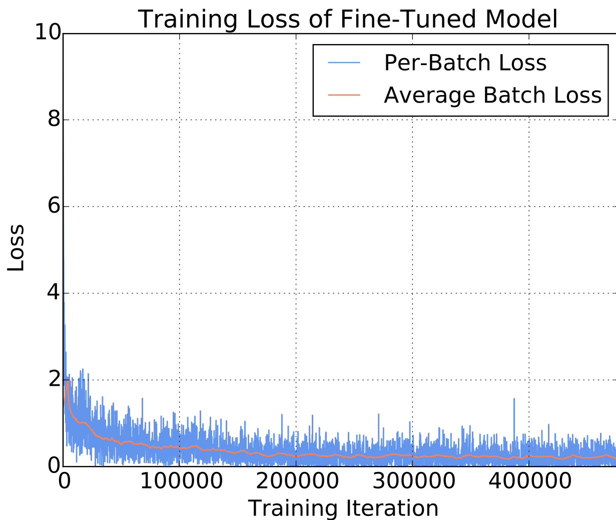


Figure 3.3: Per-batch loss, and average loss over 80 batches, of the fine-tuned model training.

The large fluctuation of the loss, as seen in Figure 3.3, could indicate that the batch size is too small. If a larger batch size had been used, more samples would be averaged over in the SGD, producing a better estimate of the full training set average. Since all models shared the same hyper-parameters, it is fair to assume that the same reasoning holds for the fc trained model as well as the frozen model. The reason for using a batch size of 32, and not a larger one, was limited graphics memory in the high end computer used for training.

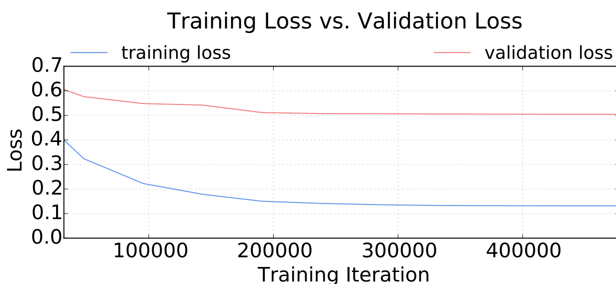


Figure 3.4: Loss of the fine-tuned model, at different stages of training, for both the validation set and a subset of the training set.

To get an idea about the amount of overfitting in the models, a subset of the training images were used for testing the fine-tuned model. The subset contained 37,631 images, roughly 13 images per identity. Figure 3.4 shows the loss of both the training subset and the validation set, throughout different stages of training.

The loss of the training subset is lower than that of the validation set, which is to be expected since the training set was used to optimize the weights and biases of the network. The loss of both sets decreases until the very end, which is a positive result. Had the validation loss started to increase while the training loss kept on decreasing, it would have been an indication that the model suffered from overfitting after that point in time.

3.2 Neural Network Compression

A static approach was taken for the experiments concerning the half data type. This means that no additional training phase was executed, instead a model trained with float was used as the basis.

Concerning pruning, experiments for both a static approach, as well as a one time retraining were carried out. Unfortunately, pruning of the convolutional layers was not explored. The reasons for this are time, and the assumption that the amount of weights that can be pruned with only a one time retraining, [Han et al., 2015b], will not be enough to outperform the cuDNN implementation of convolutional layers. Instead all focus concerning pruning is on the fully connected layers.

3.2.1 Floating Point Precision Reduction

The first experiment carried out was concerning using the 16 bit half, also referred to as FP16, instead of the 32 bit single, for the floating point representation of the neural network. This was done by converting the 32 bit model into 16 bit for the testing phase. An initial loss of accuracy for the testing set was to be expected, since the conversion implies a loss of decimal precision.

The first experiment was conducted using the high end setup, listed in Table 1.1. As mentioned in Section 2.5.1, the hardware, or more specifically the GPU architecture, of that setup does not natively support computations with the half type. This means that additional conversions, between half and float, were performed during the inference.

First the model and its parameters are converted from float to half for the whole network. For each individual layer-specific computation the parameters are read in the GPU as half, then converted to float for the computations, and after the computations the floats are converted back to halfs. Although memory is saved by representing the parameters as halfs, and the memory reads are made cheaper, no time is won in the computations.

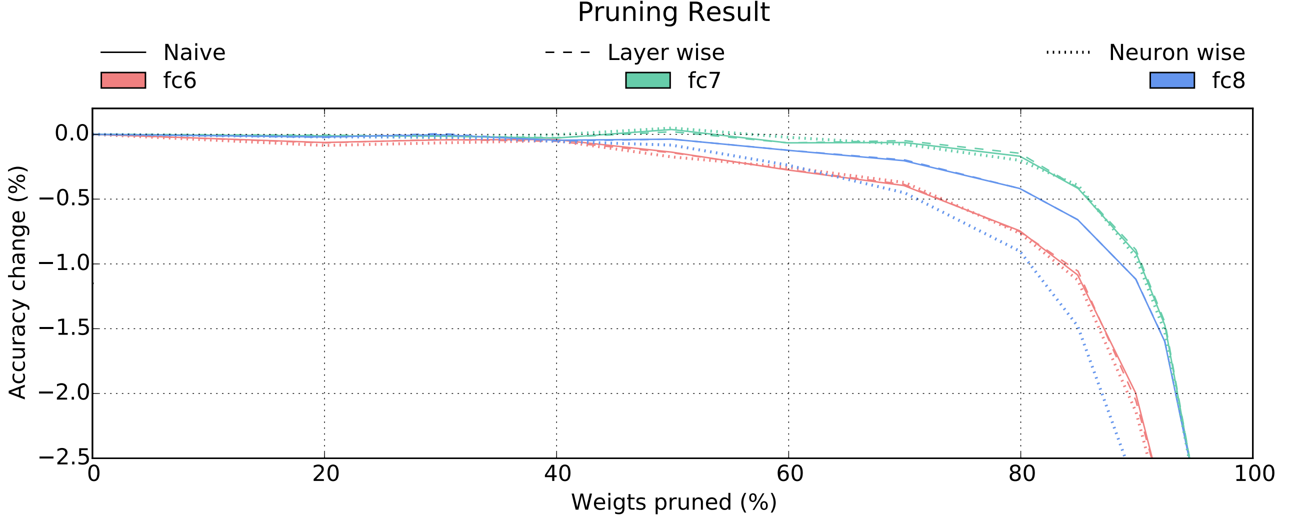


Figure 3.5: Pruning result for the fully connected layers of the fine-tuned model.

The experiment resulted in an accuracy loss of 47.15 percentage. The reduced memory footprint of the application does not make up for the accuracy loss, and deploying models trained with floats, on GPUs that do not support the half data type, is therefor unfeasible.

The second experiment concerning the half was executed on the Tegra X1, which has a GPU that does support half computations. This means that all parameters in the neural network are only converted once. Performing the same test as above only resulted in a 11.25 percentage accuracy loss, down to 79.36% accuracy, while both the memory footprint and the inference time got reduced by around half when compared to not using any optimization.

To further analyze the loss of converting the float to half, an error e was calculated as

$$e = e_c + e_{fp} = \sum_{i=1}^{M} |w_i^{32} - w_i^{16}| + \sum_{i=1}^{N} |b_i^{32} - b_i^{16}|, \quad (3.1)$$

where M and N are the total number of weights and biases correspondingly, in the neural network, w_x^{32} and w_x^{16} are weight x represented in 32 bit and 16 bit correspondingly, and the same for the biases b. The variable e_{fp} is the accumulated rounding error from representing the numbers in the floating point format, and e_c is the accumulated error of converting float numbers to halfs.

Using Equation 3.1 resulted in an error of

$$e = 185.85013236.$$

3.2.2 Pruning

The goal of the first experiment concerning pruning was to find out which of the different pruning approaches has the least negative impact on the total accuracy of the net, and how robust the different fully connected layers are to pruning.

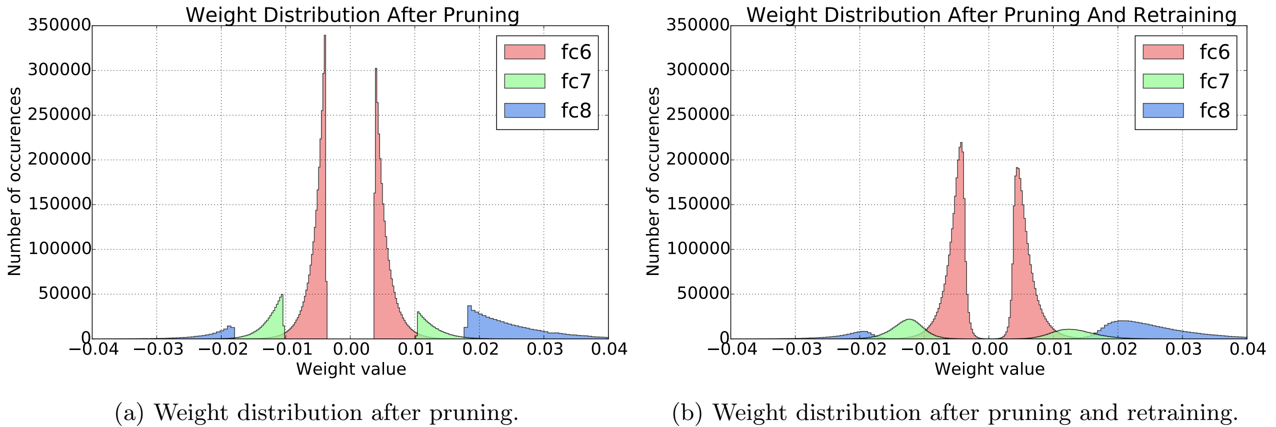
To explore this, every fully connected layer was pruned individually, using the three different pruning approaches separately, iteratively tuning the pruning parameter, γ. This experiment was also done to find out how many parameters to prune for the best possible time and memory decrease, while keeping the accuracy on an acceptable level.

The pruning parameter was tweaked such that a fixed percentage of the weights were left for every iteration. The first measurement was taken when 80% of the weights remained, and the last measurement was taken when 1% of the weights were left. The percentage of weights left was decreased each iteration, by a delta which also decreased over time, so that more measurements were performed the less weights there were left in the network.

The result of this experiment for all the fully connected layers can be seen in Figure 3.5. The y-axis of the graph was cut to better visualize the difference between the independent layers, and the three different pruning approaches per layer. The accuracy decreases rapidly after 90% of the weights are removed for each layer, and when around 2.5% of the weights are left the accuracy for all layers and methods has dropped 10%.

A lot of interesting things can be derived from this experiment. The first thing to notice is that all fully connected layers are robust to pruning, and almost 90% of the weights in every layer can be removed, by any pruning approach, only reducing the accuracy by at most ~2.5% per layer.

Out of these three layers, fc7 is the one most robust



(a) Weight distribution after pruning.

(b) Weight distribution after pruning and retraining.

Figure 3.6: Illustration of how retraining after pruning the fine-tuned model affects the distribution of the weights in the fully connected layers.

to pruning. Opposite to this is layer fc6, where the accuracy starts decreasing quite early in comparison. In practice fc6 is the layer which one would like to prune the most, as seen in the parameter distribution in Figure 2.9a. The last layer in the VGG-16 architecture, fc8, is more robust than fc6, but less robust than fc7, which is quite surprising since those weights are directly effecting the – pre-activation function and pre-softmax loss – output of the full neural network.

The different pruning approaches explored produce similar results, except for the neuron-wise approach for layer fc8, which diverges quite heavily from the result for both the naive and the layer-wise approach for the same layer.

Another thing to notice is that removing 50% of the weights in layer fc7 actually improves the accuracy for the test set used, independently of the pruning approach.

Pruning quite aggressively results in a substantial drop in accuracy. To account for this an experiment was made where 95% of the weights in the three fully connected layers were layer-wise pruned for the fine-tuned model, which had the best validation accuracy. This made the accuracy on the validation set drop to 83.38%. The pruned model was then retrained, so that the loss could be minimized for the weights that were left. Figure 3.6 shows the weight distribution for the three fully connected layers both after pruning, and after the additional training phase.

As can be seen in Figure 3.6a the removed weights for all layers are around zero. This is because the mean, μ, for the weights in these layers were

$$\mu_{fc6} = -0.000039,$$

$$\mu_{fc7} = -0.000588,$$

$$\mu_{fc8} = -0.000006.$$

Figure 3.6b shows how the weights are more evenly distributed after retraining. The retraining was run for around $70,000$ iterations and the accuracy once again reached 91%.

The resulting model was then deployed on the Tegra X1, and the different metrics in Section 2.5 were measured for both the unpruned model and the pruned one. The result can be seen in Table 3.3.

	Original	**Pruned**
Number of parameters	146 M	21 M
Offline storage	584.6 MB	100 MB
Memory footprint	1309 MB	402 MB
FLOPs	30.9 B	30.7 B
Inference time	143 ms	115 ms

Table 3.3: Values of the different metrics presented in Section 2.5, before and after pruning, as described in Section 3.2.2.

3.2.3 Inference Time Comparison

As a last experiment all the different compression algorithms were compared for the different hardware setups. The result can be seen in Figure 3.7. It should be noted that for Tegra K1 only the pruned model result is presented. This is because the memory of the Tegra K1 would not allow for one inference using the original model, and that cuDNN v4.0 was not compatible with the Tegra K1 at the time of testing. It should also be noted that all results are for the fine-tuned model, using a batch size of one, and that the pruned model used is the one described in Section 3.2.2, with only 5% of the weights left in all fully connected layers.

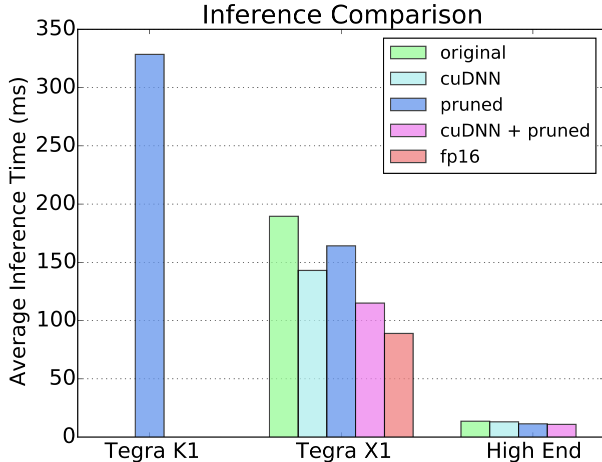


Figure 3.7: Inference time for the different compression algorithms, on the different hardware setups.

3.3 Data Analysis

In the experiments concerning data analysis, the expanded FaceScrub dataset, described in Table 2.2, was used for training and validation.

3.3.1 Data Quantity and Augmentation

Figure 3.8 shows the results of the experiment described in Section 2.8.1. The models which have been trained on the same amount of original images (those that are horizontally aligned in the graph) have trained for the same total amount of iterations. In other words, the models which contain augmented images have trained for 10 iterations per image, and the models trained on only original images have trained for 60 iterations per image.

According to Figure 3.8 augmentation clearly has a positive effect on the accuracy. Another interesting observation is that the gain in accuracy for the augmented datasets decreases the more original images are used. It seems reasonable to assume that the same trend would hold for superior augmentation techniques, although the initial gain would be greater.

The reason for using the simple augmentation techniques described in Section 2.8.1 was mainly lack of time. As discussed in [Krizhevsky et al., 2012], simple mirroring does little to affect the accuracy, and had more time been set aside for this experiment, then the other augmentation techniques described in the paper could have been tested as well.

3.3.2 Image Size

In order to explore how robust the network is to different image resolutions, the experiment in Section 2.8.2 was performed. Figure 3.10 shows the accuracy

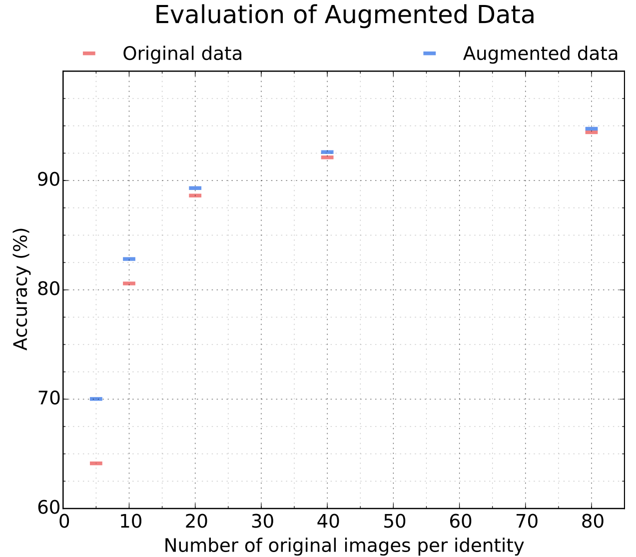


Figure 3.8: Results from the data quantity and augmentation experiments described in Section 2.8.1. The models trained on the same amount of original images have trained for the same amount of iterations.

drop for the different image sizes. The input size of the network is 224×224.

A hypothesis was made that the accuracy would decrease as soon as the test images were smaller than the input size of the network, and that the decrease would be somewhat linear. However, the accuracy drop shown in Figure 3.10 is not linear, and the accuracy is steady until the test images contain less than 70×70 pixels. After that, the accuracy starts to decrease, and it drops dramatically when the number of pixels in the test images is below 5% of the number of input pixels, i.e. when the images are smaller than 49×49 pixels.

Seeing as the results differed greatly from the hypothesis, this lead to an investigation of image size distribution in the different datasets used for training.

Figure 3.9 shows the distribution of image sizes for the downloaded part of the VGG Face dataset, which the model was originally trained on, and the distribution of the FaceScrub dataset, used in the transfer learning.

A great share of the VGG Face images are smaller than 224×224, especially compared to FaceScrub. Only 9.4% of the images in the VGG Face dataset are larger than or equal to 224×224, while 34.7% of the FaceScrub images are. In addition, no images in FaceScrub are smaller than 100×100 pixels.

The late drop in accuracy in the input size experiment could be due to the fact that all convolutional layers in the model have been trained on the

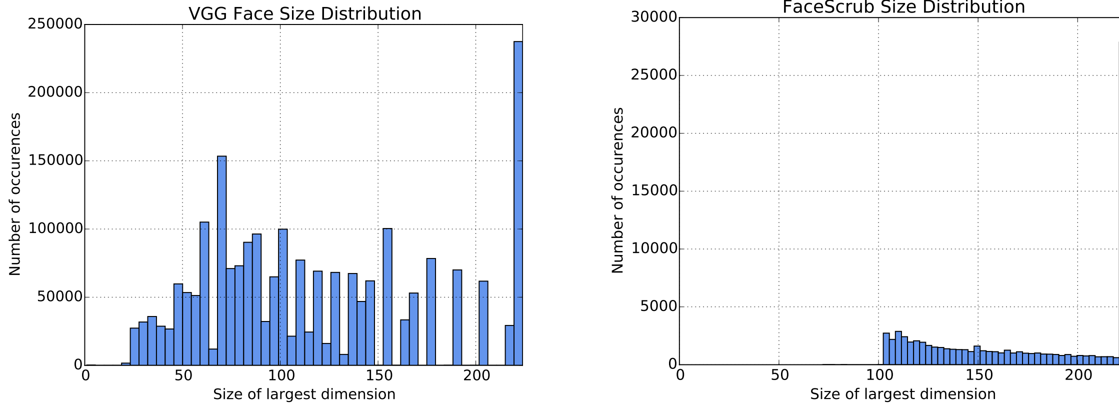


Figure 3.9: Image size distribution of the downloaded part of the VGG Face dataset, and the FaceScrub dataset. Images larger than 224×224 are included in the right-most bar.

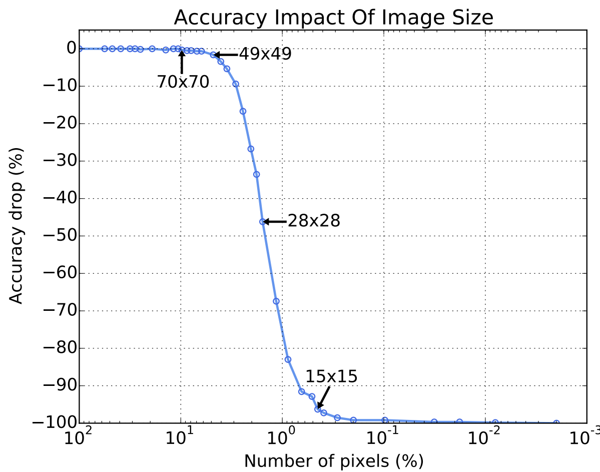


Figure 3.10: Illustration of the impact of image size relative the input size of VGG-16, using 622 test images on a model fine-tuned on FaceScrub. The original images resulted in an accuracy of 99.1961%.

VGG Face dataset, including the many small images, so that the network has learned the coarser features needed to classify them.

The model trained for this experiment shared the same training setup as the fc trained model, discussed in Section 3.1, except that the number of iterations per training image was almost 50% higher. The model achieved an accuracy of 95.66% (for the full validation set), while the fc trained model achieved only 90.61% accuracy. The reason for this gap is likely due to the difference in training iterations, as well as the difference in number of labels; the extended FaceScrub dataset contains only 346 identities, while the curated combinatory set contains 2904 identities.

3.4 Impostors

A difference was noticed between the fc trained model and the model trained on the extended FaceScrub dataset, described in Section 3.3.2, when testing them both in the camera demo. The model trained on the extended FaceScrub dataset seemed to produce higher probabilities for identities not in the database (impostors).

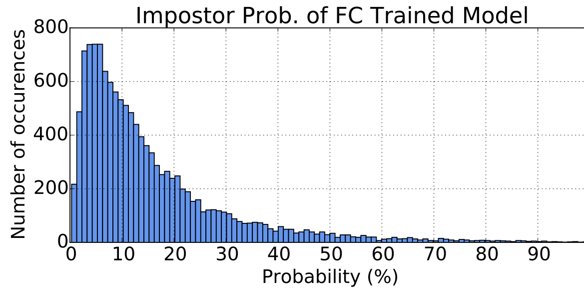
To investigate this further, an experiment was performed on the two models. The subset of LFW images with identities overlapping neither the extended FaceScrub dataset nor the curated combinatory dataset was used to analyze the probabilities of impostors.

The LFW subset contained 5,604 identities, and a total of 12,728 images. These images were all passed to both the fc trained model, and the model trained on the extended FaceScrub dataset, and the probability of the highest scoring label was saved. Figure 3.11 shows the distribution of probabilities for the different models.

As seen in the figure, the fc trained model generally generates lower classification probabilities for the impostors. This supports the observation made when comparing the two models in the camera demo, and indicates that the fc trained model is more fit to handle impostors.

Furthermore, Figure 3.12 shows a comparison of the highest impostor classification probabilities for the two models. In the fc trained model, only 0.62% of the images get classified with a probability greater or equal to 80%, and only 0.19% of the images get a probability greater than or equal to 90%. For the model trained on the extended FaceScrub dataset, the same measurements gave 7.76% and 3.87% respectively.

Using these results, one can place a threshold on the classification probability, excluding impostors by only accepting classifications above a certain value.



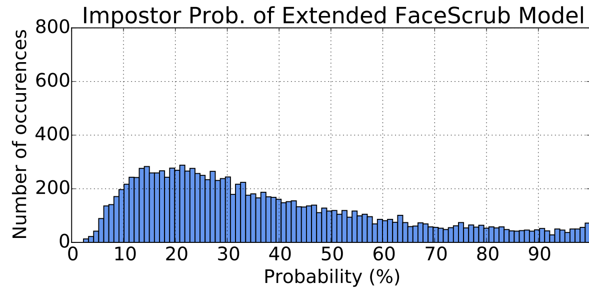


Figure 3.11: Distribution of the probability of the highest scoring label for all impostor images.

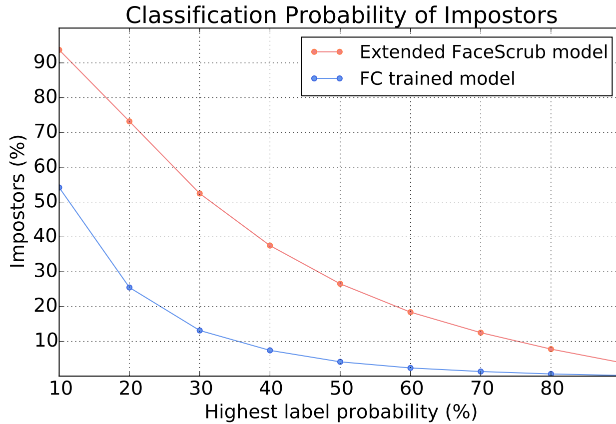


Figure 3.12: Comparison of the highest probability of assigned labels for impostor images, using the fc trained model and the model trained on the extended FaceScrub dataset.

More extensive testing could be performed to get a more reliable threshold, using a larger dataset, and making sure that all ethnicities are represented in the set.

3.5 Prototype

Overall, the prototype works very well. The total time needed for one classification is however higher than the inference time presented in Figure 3.7, since there is additional overhead in the camera demo. The overhead includes sending the images over the network, detecting and cropping faces, scaling the face images to match the input size of the network, and finally, after classification, drawing and displaying the result on the screen.

The most time-consuming of the steps above is the detection. Since face detection is out of scope for this master's thesis, no attempt was made to speed it up.

4 Conclusions

The aim of this master's thesis was to explore the possibility of deploying an algorithm for face recognition on an embedded system, while aiming for real-time performance and good accuracy.

In order to speed up the training phase, the parameters from a pre-trained model of VGG-16, called VGG Face, were transferred and re-trained using three different approaches:

Freezing the first two fully connected layers, fc6 and fc7, and training only the last layer, fc8, from scratch.

Fine-tuning the first two fully connected layers, fc6 and fc7, and training fc8 from scratch.

Training all fully connected layers from scratch.

Common for all approaches was that the weights of all convolutional layers were transferred and kept as-is.

The approach reaching the best accuracy was the fine-tuned model, in which parameters from all layers except the last were transferred from the VGG Face model, and the first two fully connected layers were fine-tuned while the third was trained from scratch.

Differentiating between 3,904 identities, the fine-tuned model achieved an accuracy of 91.66%. Another model, trained on a smaller dataset, containing only 346 identities, was able to achieve an accuracy of 96.66%. This model did however not perform as well when it came to detecting impostors – that is, the model assigned labels with high probabilities to the impostors.

As for compression algorithms both the precision reduction and the pruning approach yielded a speed increase and memory footprint decrease. Although precision reduction lead to a 11.25 percentage drop in accuracy, both the speed increase and memory decrease were enormous, around 38% and 50% correspondingly, while using cuDNN.

When it comes to pruning, only the fully connected layers – where most of the parameters are – were targeted. This gave a good result when it came to offline and online memory use, not as good of a compression as [Han et al., 2015b] presented, but on the other hand they targeted all the layers in the network, together with iterative pruning embedded in the training phase. However, the approach presented in this thesis, utilizing sparse matrices, resulted in a speed increase. It can also be noted that at first the pruning showed an accuracy drop, but this drop was minimized with retraining.

4.1 Future Work

In order to improve the accuracy of the trained CNN models, the hyper-parameters should be tuned, which was not done in this thesis. The only hyper-parameter varied between compared training setups was the learning rate of the different layers in the fine-tuned model, the frozen model, and the fc trained model. The majority of hyper-parameters used in this thesis were based on those of the VGG Face model, and are most likely not optimal.

As for the data augmentation, more complex augmentation techniques could be attempted, as mentioned in Section 3.3.1. Besides trying the augmentation techniques in [Krizhevsky et al., 2012] other than mirroring, it would be of interest to try even more advanced augmentation techniques, for example the one described in [Masi et al., 2016], using 3D models to change pose, shape, and expression of faces.

Another possibility, if constructing a dataset from scratch, would be to create 3D models of all subjects, and use those to synthesize images taken from different angles, in different light conditions. Manipulating the shape of the model would then also be a possibility, not only changing the expression of the face, but adding facial hair and other attributes which may vary over time, resulting in a synthesized dataset with high intra-class variability.

Besides trying out different types of augmentation techniques, it would be interesting to compare the effect of the different techniques, applying them one by one.

The biggest improvement when it comes to the compression algorithms explored would be to use the half floating point data type not only in the inference phase, but also in the training phase. This would make the parameters of the neural network minimize the cost function, using less precision, and when deployed the rounding error would be non-existent, since no conversion would be made.

The pruning can be further developed by also targeting the convolutional layers, perhaps with another approach than sparse matrices. The use of iterative pruning in the training phase can also be used to extend the number of pruned weights, and at the same time minimize the accuracy loss.

References

[Axis, 2015] Axis (2015). *AXIS A8004-VE Network Video Door Station*. Axis Communications AB.

[Azizpour et al., 2014] Azizpour, H., Razavian, A. S., Sullivan, J., Maki, A., and Carlsson, S. (2014). From generic to specific deep representations for visual recognition. *CoRR*, abs/1406.5774.

[Bengio, 2012] Bengio, Y. (2012). Deep learning of representations for unsupervised and transfer learning.

[Chatfield et al., 2014] Chatfield, K., Simonyan, K., Vedaldi, A., and Zisserman, A. (2014). Return of the devil in the details: Delving deep into convolutional nets. *CoRR*, abs/1405.3531.

[Chetlur et al., 2014] Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E. (2014). cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759.

[Dalal and Triggs, 2005] Dalal, N. and Triggs, B. (2005). Histograms of oriented gradients for human detection. In *International Conference on Computer Vision & Pattern Recognition*, volume 2, pages 886–893. IEEE Computer Society.

[Dubout and Fleuret, 2012] Dubout, C. and Fleuret, F. (2012). Exact acceleration of linear object detectors. In Fitzgibbon, A. W., Lazebnik, S., Perona, P., Sato, Y., and Schmid, C., editors, *ECCV (3)*, volume 7574 of *Lecture Notes in Computer Science*, pages 301–311. Springer.

[Free Software Foundation, 2007] Free Software Foundation, I. (2007). Gnu general public license.

[Gong et al., 2014] Gong, Y., Liu, L., Yang, M., and Bourdev, L. D. (2014). Compressing deep convolutional networks using vector quantization. *CoRR*, abs/1412.6115.

[Grundström et al., 2016] Grundström, J., Chen, J., Ljungqvist, M., and Åström, K. (2016). Transferring and compressing convolutional neural networks for face representations. *International Conference on Image Analysis and Recognition.*

[Han et al., 2015a] Han, S., Mao, H., and Dally, W. J. (2015a). Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *CoRR*, abs/1510.00149.

[Han et al., 2015b] Han, S., Pool, J., Tran, J., and Dally, W. J. (2015b). Learning both weights and connections for efficient neural networks. *CoRR*, abs/1506.02626.

[Hassibi et al., 1993] Hassibi, B., Stork, D. G., and Wolff, G. J. (1993). Optimal brain surgeon and general network pruning. *Neural Networks, IEEE International Conference on*, 1:293–299.

[Huang et al., 2007] Huang, G. B., Ramesh, M., Berg, T., and Learned-Miller, E. (2007). Labeled faces in the wild: A database for studying face recognition in unconstrained environments. Technical Report 07-49, University of Massachusetts, Amherst.

[Idiap, 2012] Idiap (2012). Exact acceleration of linear object detectors.

[Jia et al., 2014] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*.

[Krizhevsky et al., 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*.

[Masi et al., 2016] Masi, I., Tran, A. T., Leksut, J. T., Hassner, T., and Medioni, G. G. (2016). Do we really need to collect millions of faces for effective face recognition? *CoRR*, abs/1603.07057.

[Ng and Winkler, 2014] Ng, H.-W. and Winkler, S. (2014). A data-driven approach to cleaning large face datasets. In *IEEE International Conference on Image Processing (ICIP)*.

[NVIDIA, 2015] NVIDIA (2015). Gpu-based deep learning inference: A performance and power analysis. Technical report.

[Parkhi et al., 2015] Parkhi, O. M., Vedaldi, A., and Zisserman, A. (2015). Deep face recognition. In *British Machine Vision Conference*.

[Parkhi et al., 2016] Parkhi, O. M., Vedaldi, A., and Zisserman, A. (2015 (accessed March 23, 2016)). *VGG Face Descriptor*.

[Rastegari et al., 2016] Rastegari, M., Ordonez, V., Redmon, J., and Farhadi, A. (2016). Xnor-net: Imagenet classification using binary convolutional neural networks. *ArXiv e-prints*.

[Schroff et al., 2015] Schroff, F., Kalenichenko, D., and Philbin, J. (2015). Facenet: A unified embedding for face recognition and clustering. *CoRR*, abs/1503.03832.

[Simonyan and Zisserman, 2014] Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556.

[Taigman et al., 2014] Taigman, Y., Yang, M., Ranzato, M., and Wolf, L. (2014). Deepface: Closing the gap to human-level performance in face verification. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[Vanhoucke et al., 2011] Vanhoucke, V., Senior, A., and Mao, M. Z. (2011). Improving the speed of neural networks on cpus. In *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*.

[Viola and Jones, 2001] Viola, P. and Jones, M. (2001). Rapid object detection using a boosted cascade of simple features. pages 511–518.

[Walsh, 2013] Walsh, C. A. (2013). Peter Huttenlocher (1931-2013). *Nature*, 502:172–172.

[Yosinski et al., 2014] Yosinski, J., Clune, J., Bengio, Y., and Lipson, H. (2014). How transferable are features in deep neural networks? *CoRR*, abs/1411.1792.

Master's Theses in Mathematical Sciences 2016:E17
ISSN 1404-6342

LUTFMA-3293-2016

Mathematics
Centre for Mathematical Sciences
Lund University
Box 118, SE-221 00 Lund, Sweden

http://www.maths.lth.se/