



Henri Lunnikivi

# Model compression methods for convolutional neural networks

Faculty of Information Technology and Communication Sciences (ITC)  
Master's thesis  
November 2019

# Abstract

Henri Lunnikivi: Model compression methods for convolutional neural networks  
Master's thesis  
Tampere University  
Master's Degree Programme in Information Technology  
November 2019

---

Deep learning has been found to be an effective solution to many problems in the field of computer vision. Convolutional networks have been a particularly successful model for computer vision. Convolutional neural networks extract feature maps from an image, then use the feature maps to determine to which of the preset categories the image belongs. Convolutional neural networks can be trained on a powerful machine, and then deployed onto a target device for inference. Computing inference has become feasible on mobile phones and IoT edge devices. However, these devices come with constraints like reduced processing resources, smaller memory caches, decreased memory bandwidth. To make computing inference practical on these devices, the effectiveness of various model compression methods is evaluated quantitatively. Methods are evaluated by applying them on a simple convolutional neural network for optical vehicle classification. Convolutional layers are separated into component vectors for a reduction in inference time on CPU, GPU, and an embedded target. Fully connected layers are pruned and retuned in combination with regularization and dropout. Pruned layers are compressed using a sparse matrix format. All optimizations are tested on three platforms with varying capabilities. Separation of convolutional layers improves latency of the whole model by  $3.00\times$  on a CPU platform. Using a sparse format on a pruned model with a large fully connected layer improves latency of the whole model by  $2.01\times$  on desktop with a GPU and by  $1.82\times$  on the embedded platform. On average, pruning the model allows  $39.1\times$  reduction in total model size while causing a 1.67 %-point reduction in accuracy, when dropout is used to control overfitting. This allows for a vehicle classifier to fit in 180 kB of memory with reasonable reduction in accuracy.

**Keywords:** CNN, cross-platform, deep learning, dropout, GPGPU, Mali, master of science thesis, model compression, neural networks, OpenCL, pruning, Rust, separable convolutions, supervised learning.

The originality of this thesis has been checked using the Turnitin Originality Check service.

# Tiivistelmä

Henri Lunnikivi: Konvoluutioneuroverkkojen mallinpakkausmenetelmien arviointi  
Diplomityö  
Tampereen yliopisto  
Tietotekniikan DI-tutkinto-ohjelma  
Marraskuu 2019

---

Syväoppimisen on todettu olevan hyvä ratkaisu moniin konenäön ongelmiin. Konenäön sovelluksissa erityisen menestyksekkäästi käytetyt konvoluutioneuroverkot yksinkertaistavat laskentaa erottelemalla kuvasta korkean tason piirrekarttoja (engl. feature map). Konvoluutioneuroverkkoja voidaan kouluttaa tehokkaalla laitteistolla, josta mallin painokertoimet voidaan ottaa käyttöön kohdelaitteelle. Mallin käyttämisestä on tullut mahdollista matkapuhelimissa ja Internet-reunalaitteissa. Nämä laitteet ovat kuitenkin usein monella tavalla rajoittuneita: vähemmän laskentakapasiteettia, pienemmät välimuistit ja vähemmän erilaisten muistien välistä kaistanleveyttä. Jotta mallin käyttämisestä voitaisiin tehdä käytännöllisempää tällaisilla laitteilla, erilaisten mallinpakkausmenetelmien tehokkuutta arvioidaan kvantitatiivisesti. Työssä menetelmiä arvioidaan käyttäen niitä yksinkertaiseen konvoluutioneuroverkkoon, joka luokittelee ajoneuvoja optisesti. Konvoluutiokerrokset optimoidaan separoimalla ne pohjavektoreiksi, jotta mallin käyttäminen olisi laskennallisesti yksinkertaisempaa. Täysin kytkettyjen kerrosten vähiten merkityksellisiä painokertoimia karsitaan regularisoinnin (engl. regularization) ja dropout-menetelmien avulla. Karsitut kerrokset pakataan harvaan matriisimuotoon. Optimoituja verkkoja testataan pöytätietokoneen suorittimella, grafiikkasuorittimella ja tyypillisellä sulautetulla laskentajärjestelmällä. Konvoluutioiden separointi nopeuttaa koko mallin suoritusta 3-kertaisesti tavallisella suorittimella. Karsitut mallit ja harva matriisimuoto nopeuttavat mallin suoritusta 2,01-kertaisesti kun konvoluutiot voidaan ajaa grafiikkasuorittimella. Vastaava nopeutus sulautetulla laskentajärjestelmällä on 1,82-kertainen. Mallin karsiminen vähentää mallin kokonaismuistivaatimusta 39,1-kertaisesti, aiheuttaen noin 1,67 %-yksikön menetyksen tarkkuudessa, kun käytetään dropout-menetelmää ylisovittumisen (engl. overfitting) estämiseen. Tämä mahdollistaa ajoneuvoluokittelijan sovittamisen 180 kilotavuun muistia.

**Avainsanat:** diplomityö, dropout, GPGPU, konvoluutioneuroverkot, Mali, mallinpakkaus, neuroverkot, OpenCL, pruning, Rust, separoituvat konvoluutiot, syväoppiminen, valvottu oppiminen.

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck -ohjelmalla.

# Contents

1	Introduction . . . . .	1
1.1	Deep neural networks and image recognition . . . . .	2
1.2	Deep models with resource-constraints . . . . .	4
1.3	Separable convolutions and sparse layers . . . . .	5
2	Convolutional neural networks and model compression . . . . .	6
2.1	The structure of a convolutional neural network . . . . .	7
2.1.1	Fully connected layer . . . . .	8
2.1.2	Convolutional layer . . . . .	10
2.1.3	Rectifier linear unit . . . . .	11
2.1.4	Softmax . . . . .	13
2.2	Model training and the pruning process . . . . .	13
2.2.1	Calculating training loss . . . . .	14
2.2.2	Backpropagation . . . . .	15
2.2.3	Gradient descent . . . . .	17
2.2.4	Regularization . . . . .	17
2.3	Proposed optimizations . . . . .	19
2.3.1	Weights pruning . . . . .	19
2.3.2	Sparse models . . . . .	20
2.3.3	Separable convolutions . . . . .	20
3	Implementation of optimizations . . . . .	23
3.1	Use case: a convolutional network for car type recognition . . . . .	23
3.2	Implementation of training . . . . .	25
3.2.1	Pruning . . . . .	25
3.2.2	Assessment of model accuracy and overfitting . . . . .	26
3.3	Inference optimizations . . . . .	26
3.3.1	Sparse matrix multiplication . . . . .	27
3.3.2	Separable convolutions . . . . .	30
4	Measurements . . . . .	31
4.1	Data collection . . . . .	31
4.1.1	Threshold pruning . . . . .	33
4.1.2	L2 pruning . . . . .	33
4.1.3	L2 pruning with 50 % dropout . . . . .	35
4.1.4	Performance measurements . . . . .	36
4.2	Analysis of results . . . . .	39
4.2.1	Pruning for model size and latency reduction . . . . .	39

4.2.2	Impact of dropout on pruning with L2 regularization . . . . .	40
4.2.3	Latency across devices . . . . .	42
4.2.4	Combining pruning and separable convolutions . . . . .	42
4.2.5	Reliability of results . . . . .	42
4.3	Summary of findings . . . . .	43
5	Conclusions . . . . .	44
5.1	Discussion . . . . .	44
5.2	Recommendation . . . . .	45
5.3	Future work . . . . .	45
	References . . . . .	48
	APPENDIX A. Results of model pruning . . . . .	49

# Abbreviations

Adam	Adaptive moment estimation algorithm for gradient descent
AI	Artificial intelligence
ASIC	Application specific integrated circuit
AVX	Advanced vector extensions
BLAS	Basic linear algebra subroutine
CNN	Convolutional neural network
CPU	Central processing unit
CRS	Compressed row storage
CSB	Compressed sparse blocks
DNN	Deep neural network
DRAM	Dynamic random-access memory
FMA	Fused multiply-add
FPGA	Field-programmable gate array
GAN	General adversarial network
GEMM	General matrix multiply
GPGPU	General-purpose computing on graphics processing units
GPP	General purpose processor
GPU	Graphics processing unit
MAC	Multiplier-accumulator
MLP	Multilayer perceptron
MSE	Mean squared error
ReLU	Rectifier linear unit
SGD	Stochastic gradient descent
SIMD	Single instruction, multiple data
SIMT	Single instruction, multiple threads
SRAM	Static random-access memory
SSE	Streaming SIMD extension
SVM	Support vector machine
TPU	Tensor Processing Unit
VCN	A convolutional neural network for vehicle classification [15] by Huttunen et al.
nnz	Non-zero
fN	N-bit floating-point number (eg. float f32, double f64)
iN	N-bit signed fixed-point integer (eg. signed char, short, int)
uN	N-bit unsigned fixed-point integer (eg. unsigned char)
$\mathbb{N}$	The set of all natural numbers $n \in [1, \infty[$

$\mathbb{R}$	The set of all real numbers. Often practically implemented as a 32-bit or a 64-bit floating point.
$\mathbb{Z}$	The set of all integers.
$\star$	Operator for cross-correlation.

# 1 Introduction

Deep learning is becoming pervasive as a solution to problems in the field of computer vision. In 2012, Cireřan et al. [6] reported reaching and exceeding of human level performance on state of the art computer vision benchmarks. Soon after, Krizhevsky’s *convolutional neural network* [20] classified the 1000-category ImageNet database of images with unprecedented accuracy. They achieved this level of accuracy by using a significantly more efficient training procedure that enabled them to fit a large model to the 1.2 million training examples of the huge ImageNet database. The best modern deep learning models, based on the convolutional neural network archetype, are able to create photorealistic images of people [18] or transform hand-drawn style guide images into synthesized photorealistic sceneries [26].

Research on convolutional neural networks and their derivatives has produced impressive results. The number of applications for convolutional neural networks and other deep learning models has increased steadily. The wealth of applicability is making deep learning pervasive, and there is a growing interest in how to deploy deep learning models on mobile phones, embedded devices, wearables, and in other applications of edge computing [1, 15].

As AI becomes pervasive, the requirements to supporting infrastructure increase as well. Pushing location of computing towards the end-user helps reduce total power consumption and increase privacy. However, deep learning and edge computing is a difficult combination. Resource constraints include: limited memory size and bandwidth, limited processing power and requirements for efficient usage of hardware. Simpler processors and *application specific integrated circuits* (ASICs) are often used instead of general purpose processors. In the past, running a deep learning model on an edge device has seemed like a tall order. In recent years, there has been plenty of research into compressing models. An older state-of-the-art deep model like AlexNet [20] used to have 60 million parameters, while recent cutting edge networks like MobileNetV2<sup>1</sup> [28] have only a little more than 2 million parameters for a similar or better accuracy. Another recent space-optimized variant, SqueezeNet, only has 1.2 million parameters, resulting in a < 500 kB model size [16].

For edge computing, reducing resource usage of deep learning models also results in less communication between devices, which results in less total power used and better bandwidth usage. Enabling a model to run on an edge device also mitigates the privacy problems inherent with the very common practice of server-offloading, where a picture taken on a cell phone might be sent to a server for processing. Moving

---

<sup>1</sup>open-sourced and available as part of TENSORFLOW-SLIM IMAGE CLASSIFICATION LIBRARY in the TENSORFLOW API



the processing onto a user’s phone or an intermediate edge device lessens problems caused by server-offloading. Additionally, it makes it easier for a user to be aware of where their data has gone.

The particular constraints of edge platforms create a need for particular techniques and processes. There is a need for models that are smaller, require less power, and that enable computation on the device instead of server-offloading. Enabling creation of models like this makes it possible to apply deep learning models on a wider range of platforms for cheaper and more securely than before.

The aim of this work is to evaluate the effects, results and platform applicability of some recent techniques in creating smaller and more efficient models. The main methods evaluated are cross-platform model compression and computational approximations.

## 1.1 Deep neural networks and image recognition

The neural network is a basic model structure of deep learning. A neural network comprises interconnected weights, or parameters, that transform the inputs in a series of layers. A properly trained network with enough input data can infer classes or attributes from unseen inputs. An instance of a simple deep neural network (DNN) is the *multilayer perceptron* (MLP). A MLP takes selected features as a vector of inputs. Inputs are then multiplied in chained fully connected hidden layers and a nonlinearity for each layer. The role of fully connected layers is to apply a weight to each of the inputs, and the role of the nonlinearities is to simulate an activation of a neuron. Fully connected layers are often implemented as plain matrix multiplications, while there are multiple popular implementations of different activation functions. To summarize, the inputs to the network are progressively multiplied with matrices and then activated in a predefined structure of layers.

The many parameters of a network can be algorithmically stabilized such that the resultant model can correctly classify input data it has never seen before. This is called *training* of weights. Training can be done by ”showing” the network carefully selected and correctly labeled input data, and adjusting the weights towards the least error with a gradient descent algorithm. In addition to the parameters, a network is defined by its hyperparameters. The typical hyperparameters of a network are the number of layers and the cardinality of each layer. The first layer starts the transformation of the input data, and the last layer provides a representation of the desired outputs.

The creation and operation of a neural network can thus be coarsely divided into two phases: *training* and *inference*. The training phase involves the use of a massive set of training data to estimate the correct parameters for the network. Training until the network stops learning (ie. *convergence*) can often take hours, days

or weeks depending on the size of the model and the availability of computational resources. In inference, the deployed network is making predictions on data it has never seen before, and the training data set needs not be present on the inference device. Inference is a real-time application. Real-time applications are often latency sensitive and constrained by available resources on the real-world platforms that need to be doing the inference. While training is mostly done on a high performance computer, inference might be run on a small edge device. Examples of devices that might be running inference would be an end user’s cell phone, or an integrated system like a security camera.

Hardware manufacturers have risen to the occasion by developing various hardware solutions to accelerating computations for machine learning, both for server-side and edge computing. At Google I/O, in 2016, Google announced its Tensor Processing Unit (TPU), which had already been in use in their data centers. A TPU resembles a GPU in its ability to do high volumes computation power efficiently but lacks rasterization and texture mapping capabilities. In 2018, Google announced the Edge TPU - a power efficient ASIC for machine learning in edge computing. All of the optimizations measured on a GPU in this thesis should also translate to comparable TPUs with similar effects on performance, possibly with further reductions in power consumption. Other notable machine learning manufacturers include Intel Movidius<sup>2</sup> which produces special hardware for computer vision, and Huawei with their Kirin<sup>3</sup>-family of SoC chipsets, which are oriented for mobile AI.

The convolutional neural network (CNN) is a special case of a deep neural network, with layers of adaptive feature extractors before the fully connected layers. CNNs have held the title of being the best adaptive image recognizer since 2011, when Ciresan et al. used CUDA to accelerate the training of a general, fully parameterizable convolutional network on a GPU platform. That implementation used maxpooling and gradient descent on a GPU to achieve best published results on object classification benchmarks of that time (NORB, CIFAR10, MNIST). [5]

The bar for the state-of-the-art of convolutional neural networks was again raised in 2012 by Alex Krizhevsky’s AlexNet [20]. His work enabled quick training of CNNs on very large datasets. This was achieved by speeding up network training via swapping the commonly used nonlinearity function  $f(x) = \tanh(x)$  for a nonsaturating nonlinearity called ReLU, and then by training the network using a GPU. ReLU is defined simply as  $R(x) = \max(0, x)$  and it was first applied in signal processing by Hahnloser et al. in 2000 [9]. ReLU and its gradient are easy to compute on a GPU. This technique has been found to consistently reduce model training times by at least a factor of 2 – 6×. Additionally, they effectively reduced model overfitting by

---

<sup>2</sup><https://www.movidius.com/>

<sup>3</sup><https://consumer.huawei.com/en/campaign/kirin980/>

use of a technique called *dropout*, where half of the model parameters are randomly selected for temporary deactivation during training.

## 1.2 Deep models with resource-constraints

Deep learning models have come a long way in ease of computing. Creating and training a model with sufficient predictive power used to be difficult, but is now practical. While the process of training an effective computational model is still a task better fit for specialized, high-capacity hardware, the inferring models have become small. The size reduction of models has proven it possible [21] to fit a model on an embedded device, with constraints on memory bandwidth and processing power. In 2016, Iandola et al. introduced SqueezeNet [16], that reaches AlexNet level accuracy on ImageNet with  $50\times$  less parameters. SqueezeNet can also be compressed into the size of 0.5 MB, which is  $510\times$  smaller than AlexNet. Running these networks on smaller devices, ASICs and FPGAs is now viable.

Reducing model size has many advantages. Memory access on small devices is generally energy intensive. Han et al. indicate that in a neural network, the energy use per operation is most dependent on the type of memory access:  $5\text{ pJ}$  for on-chip SRAM and  $640\text{ pJ}$  on off-chip DRAM [11]. There is a 100-fold difference in using a register w.r.t. using the main memory. Reducing the size of the network is generally effective in reducing the total energy cost of the network by moving the computation closer to the processor.

Another advantage of small model size is when a model is trained in a distributed environment, where servers need to communicate less [16]. Lane and Bhattacharya discuss that many of the current models running on smaller devices also offload the inference to an external server [1]. This causes a privacy issue where unnecessary data, such as the pictures that a cell phone user takes, leak further away from the device where they are needed. Not requiring cloud offloading for inference alleviates these privacy concerns.

Smaller model size implies that less data need to move between circuits and between devices. For the current processor architectures, this means that a smaller network layer might fit into a different kind of a memory or a smaller cache that is closer to the computation device doing the work. Moving less data between devices is also obviously beneficial to reduce network load and delays in real-time systems.

There has been continuing research interest in reducing the size of deep models. A recent study [10] by Han, Mao and Dally has been able to reduce the memory footprint of AlexNet by  $35\times$  and the memory footprint of the VGG-16 by  $49\times$ , both with only a very slight reduction in accuracy. This was achieved via pruning of low-relevance weights, trained quantization and Huffman coding.

In a recent study [1] by Lane and Bhattacharya, they developed an approach called

SparseSep to reduce the memory footprint of a CNN consisting of both a notable amount of convolutional layers and fully connected layers. In SparseSep, the idea is to compress the fully connected layers via use of sparse matrices and the convolution layers via separation of the kernels into vectors. Both techniques involve insertion of a new layer to get a reduction in total number of computations. The techniques produce a minor loss in accuracy, but they claim to use principled methods that avoid specializing the original model to recognize a smaller set of activities/contexts. They were able to reduce the memory footprint of some commonly used deep neural networks by  $11.3\times$  on average and get the networks to run  $13.3\times$  faster on platforms like Qualcomm Snapdragon 400, ARM Cortex M0 and M3, and Nvidia Tegra K1.

### 1.3 Separable convolutions and sparse layers

Effective compression of convolutional neural networks requires strategies that are able to reduce the size of both convolutional and fully connected layers. Model pruning is a method of removing low impact parameters and has been previously used effectively in model compression [10, 34], though it has been shown to be mostly effective on the fully connected layers, and less so on convolutional layers.

Combining the model pruning approach with an orthogonal approach for reducing the size of the convolutional layers is an effective way to gain good total model compression. An effective way to reduce the cost of the convolutional layers has been introduced by Jaderberg et al. [17]. They exploit the redundancy between different filters by decomposing the convolution into a horizontal and a vertical convolutional vector.

In this study, the effects of several recent inference optimization techniques are investigated in the context of a small CNN [15]. This CNN classifies images into four classes of vehicles. The network will be introduced in detail in Chapter 3. The performance characteristics of the compressed layers are empirically mapped for three different devices: a desktop computer with a graphical processor, a multithreading Intel CPU and a representative embedded computer.

## 2 Convolutional neural networks and model compression

Making a small MLP capable of making predictions on images is nigh impossible due to the large size of image files. Multiplying the pixels of an image taken on a multi-megapixel camera with a weights matrix costs unfeasibly high amounts of computing power. Fortunately, much smaller images, like a  $128 \times 128$  pixels RGB image, usually contain enough information for a computer to make predictions on. However, this is still too large to enable training a MLP and another approach is required.

Prior to training, MLP has no presumption of any correlation between the parameters of the input layer. In a real world image, the close-together pixels are strongly correlated. A more macroscopic feature of the image might have significance regardless of its position in the image. Given enough training data, a MLP may learn to account for this, but it is computationally very costly. The other option is to build these types of correlations into the model via use of techniques called local receptive fields, weight sharing and subsampling (Chapter 2). This is what makes up a convolutional neural network (CNN). [2, pp. 267-269]

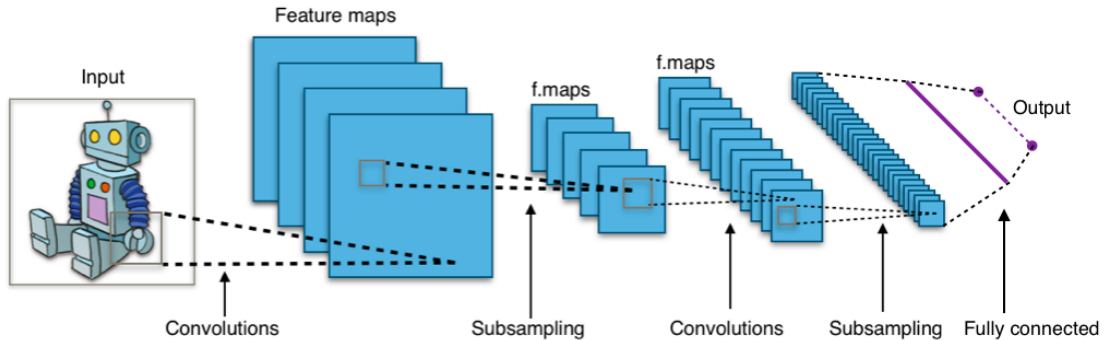
Convolutional neural networks solve the problem of extracting macroscopic features from the image instead of treating each input pixel as a feature. This has allowed models to have much less parameters, e.g. one  $3 \times 3$  kernel of parameters per feature instead of one for each combination of one pixel and one hidden layer node.

The convolutional neural network (CNN) is a DNN that is widely applied in image recognition tasks [2, p. 267], and has in fact been the state-of-the-art model architecture of image recognition since the implementation by Cireşan et al. [5]. It differs from a MLP in the inclusion of one or more convolutional layers at the start of the network. To understand the purpose of including the convolutional layers, the difference to a MLP is explained next.

Prior to training, a DNN does not assume any correlation between the parameters of the input layer, though it may learn it with enough input data. A convolutional neural network uses the intrinsic properties of images to extract relevant information before the fully connected layers. This is achieved by leveraging the fact that the close-by pixels of a real-world photo are more related to each other than distant pixels. Convolutional layers extract local features into feature maps using local receptive fields and weight sharing (2.1.2 Convolutional layer). Stacking more than one convolutional layer makes it possible to extract higher-order features. Convolutional layers are often paired with subsampling layers that effectively reduce the amount of

data in the pipeline and make the network less sensitive to small input translations. [2, pp. 267-269]

An illustration of a convolutional neural network is presented in Figure 2.1.



**Figure 2.1** An illustration of a CNN [32].

As seen in Figure 2.1, features get extracted with convolutions from an image into feature maps. These feature maps get subsampled into smaller feature maps. From those smaller feature maps, higher order features are extracted and again subsampled. The final layer of feature maps gets fed into one or more fully connected layers that produce the final output.

## 2.1 The structure of a convolutional neural network

Convolutional neural networks are essentially a stack of linear algebra, organized into layers. This section describes the different layer types and the non-linear activation functions used for post-processing layer outputs.

A convolutional neural network predicts the class of an input image from a number of pre-defined categories. It does so by passing the input image through a series of specifically weighted layer transformations in the form of convolutions and matrix multiplications. Transforming an input with a well-selected set of weights produces the right class label for a given input image. Obtaining well-selected weights for the layer transformations is called *training* of the model. The particulars of training are further explained in Section 2.2.

When a CNN makes a class prediction (i.e. inference), inputs are transformed through a series of layers. Each layer has a hyper-dimensional weights matrix - i.e. a tensor that the input gets multiplied with. The structure of a layer transformation generally matches the following template:

1. an input tensor, ...
2. multiplied or convolved with a *weights* tensor, ...
3. with a function for post-processing, ...

4. producing an output tensor.

The exact shape and function of each of the parts varies between different layers. Weights might function like a filter in a convolutional layer, or as a simple weighting matrix for a set of input neurons in a fully connected layer. In post-processing, the output may be aggregated to a reduced size, and/or an activation function may be applied to transform each element in the output to otherwise enhance the capabilities of the network. Finally, the outputs are propagated to the next hidden layer, or the final output probabilities.

The linear algebra needed for inference includes cross-correlation for convolution, matrix multiplication for fully connected layers and an activation function for the nonlinearities. In convolutional layers, features are extracted from the input image using local receptive fields (i.e. filters) that multiply-add or convolve each pixel in the input. This produces a feature map that is subsampled to a smaller feature map that makes it more invariant to small translations in input data. Reduced-size feature maps from subsampling are fed into fully connected layers. In a multiclass image classification application, the last layer would usually be a fully connected layer, with a softmax nonlinear function to convert numbers into action probabilities matching with each output class. [2, p. 267-269]

The simplest part of a CNN, a commonality with the MLP, is the fully connected layer.

### 2.1.1 Fully connected layer

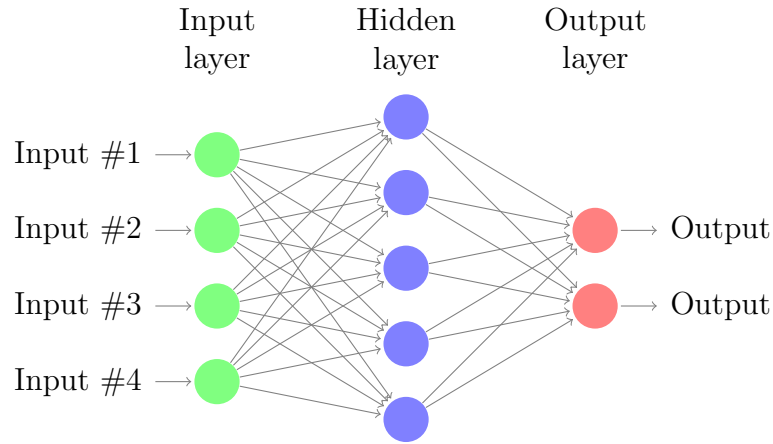
The fully connected layer is the basic building block and the defining factor of most contemporary neural networks. A model consisting only of fully connected layers and activations is called a multilayer perceptron or a feed-forward neural network. In their 2006 book, Bishop et al. described it as having been proven to be the network type of greatest practical value in pattern recognition [2, p. 226]. The modern convolutional neural network such as the one used by Krizhevsky et al. [20] still uses this layer type to correlate image features and outputs.

A fully connected layer, put simply, is a matrix multiplication between well-selected parameters and values of input neurons, producing the values of output neurons  $\vec{z}_i$  with a bias weight  $\vec{b}$  sometimes added to each output neuron.<sup>1</sup> These outputs become *activations* after being activated by the activation function. The vector formed by the activations of neurons in one layer is denoted with  $\vec{a}_i$ . Obviously, what follows is that the activations of neurons in any preceding layer are called  $\vec{a}_{i-1}$ , and

---

<sup>1</sup>Mathematical descriptions about multilayer perceptrons and backpropagation in this chapter are adapted from <http://neuralnetworksanddeeplearning.com/> [23]

the neurons in the following layer are called  $\vec{a}_{l+1}$ . The structure of a fully connected layer in the context of a simple neural network is illustrated in Figure 2.2.



**Figure 2.2** An illustration of a simple feed-forward neural network with one hidden layer. [8]

In a fully connected layer, every neuron is connected to every neuron in the previous layer multiplied by a weight. Mathematically, this is matrix multiplication between the activations of the previous layer  $\vec{a}_{l-1} \in \mathbb{R}^{M \times 1}$ , where  $M$  is the number of neurons in the previous layer, and the weights matrix  $\mathbf{W}_l \in \mathbb{R}^{N \times M}$ , where  $N$  is the number of neurons in the current layer. Additionally, there can be one bias variable  $b_l$  summed to each output neuron, though the network described later in this thesis manages without. This calculation produces output  $z_l = \mathbf{W}_l \vec{a}_{l-1} + b_l \in \mathbb{R}^{N \times 1}$ . Finally, an activation function is applied to the output. The rectifier function (denoted by  $R$ ) is used in all layers except the last one. The rectifier function  $R$  is better explained later in Subsection 2.1.3. The activations of a given layer are defined in terms of the activations in the previous layer as follows:

$$\vec{a}_l = R(\mathbf{W}_l \vec{a}_{l-1} + \vec{b}_l). \quad (1.1)$$

The network described later in this thesis (Chapter 3) contains no biases, so the expression for the fully connected layer simplifies to:

$$\vec{a}_l = R(\mathbf{W}_l \vec{a}_{l-1}). \quad (1.2)$$

In a MLP, these fully connected layers are chained, and the result of the final layer  $\vec{a}_L$  can be calculated recursively based on Equation 1.2. Executing this part of the algorithm on a computer is called a *forward pass*. A forward pass is required for both the first training pass, and later when the model is deployed, during inference. The matrix multiplications of the fully connected layers can be efficiently computed on most kinds of contemporary hardware by using highly optimized general matrix



multiplication implementations (GEMM), while applying the rectifier function is very efficient on all kinds of processors.

### 2.1.2 Convolutional layer

Fully connected layers and non-linearities can be used to represent the solution to any problem space, though this is often computationally expensive and redundant. The common solution to this problem in the domain of image recognition is to first extract features from the input data using convolutional layers.

The convolutional layer is a solution to the inability of the fully connected layer to correlate spatial locality, with the benefit of drastic reduction in total computation and number of weights required. Correlating spatial locality is achieved via use of local receptive fields, weight sharing and subsampling. These techniques substantially reduce the number of parameters in the network when compared to a fully connected layer. [2, p. 269]

A convolutional layer processes an image into a set of  $F$  feature maps by *convolving* the image with  $F$  weights kernels. Each feature map is produced by convolving the respective weights kernel with each overlapping, kernel-sized neighborhood of input pixels. Each of the kernels essentially detects one pattern or feature within the input image in all positions of the image. Multiple chained convolutional layers detect higher-order features. [2]

In the network under optimization (explained later in Section 3.1), there is a downsampling layer after each convolution. The subsampling layer takes as input all nonoverlapping rectangles of the feature map and picks the highest value as output. A  $2 \times 2$  max pooling layer would then produce a half-sized version of the feature map. Chaining convolution layers with subsampling layers increases the degree of invariance of features to minor input transformations [2].

The algorithm for a convolutional layer is, in principle, similar to a filter in an image processing program. In the algorithm for a monochrome image, a *weighted, constant-sized, 2D filter* is multiplied over each input pixel neighborhood. Then the products are summed, which produces a single output pixel. This repeats for every input pixel, producing a 2D set of pixels known as the *feature map*. This process repeats for a number of times for a number of different filter weights, thus producing a number of different feature maps. The multicolored RGB version of this algorithm changes only in that the filters are 3-dimensional, where the last dimension is the color. Each 3D neighborhood still produces one singular output pixel per feature map because the products for each color are finally summed.

Given a specific hard-coded kernel and convolving it over an image, a variety of effects can be produced, as seen in Figure 2.3. In *model training*, this effect is turned around to train the weights of the convolutions such that they produce maps

of trainable features.

Convolution in machine learning is almost always implemented as *cross-correlation*. Using the notation introduced in Subsection 2.1.1, 2-D cross-correlation between a monochrome input image of  $(i, j) \in W \times H$  and the  $(u, v) \in K \times K$  kernel of one feature map is expressed as:

$$\vec{z}_{l,i,j} = \mathbf{W}_l \star \vec{a}_{l-1} \triangleq \sum_{u=-k}^k \sum_{v=-k}^k \mathbf{W}_{l,u,v} \vec{a}_{l-1,i+u,j+v}.$$

Here,  $\star$  is used to denote cross-correlation and  $k = \lfloor \frac{K}{2} \rfloor$  is the radius of the kernel. The inputs to a convolution usually have multiple channels  $C$  (eg. in RGB,  $C = 3$ ), and multiple features are detected using  $F$  filters. Adding in  $c \in \mathbb{N}^C$  channels and  $f \in \mathbb{N}^F$  filters and feature maps, the final equation for 3-D cross-correlation is defined as

$$\vec{z}_{l,f,i,j} = \mathbf{W}_l \star \vec{a}_{l-1} \triangleq \sum_{c=1}^C \sum_{u=-k}^k \sum_{v=-k}^k \mathbf{W}_{l,f,u,v} \vec{a}_{l-1,i+u,j+v}. \quad (2.2)$$

A plain convolution innately produces a slightly smaller feature map in relation to the original image by cropping a little bit of the edges of the image. This intuitively happens because the sliding multiplication window over the original image has no values to multiply for the outermost pixels of the feature map, as can be seen in Figure 2.4. Often, it would be useful to have the output feature map be equal in size to the input image. This issue can be remedied by adding zeroes as *edge padding* to the input image for the kernel to convolve over.

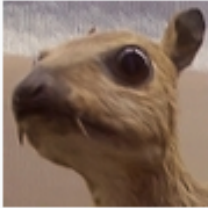



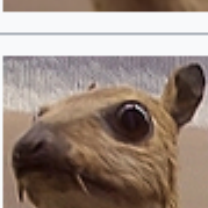
Mathematically, a convolution is still a linear transformation and as such, can be computed using a matrix multiplication algorithm, provided that the input image is first reordered. Chellapilla et al. were the first in 2006 to use a technique which first unrolls the convolution, then uses a basic linear algebra subroutine (BLAS) to efficiently compute the convolution as a matrix multiplication [4]. This unrolling operation is often referred to as *im2col*, while the opposite operation is called *col2im*.

### 2.1.3 Rectifier linear unit

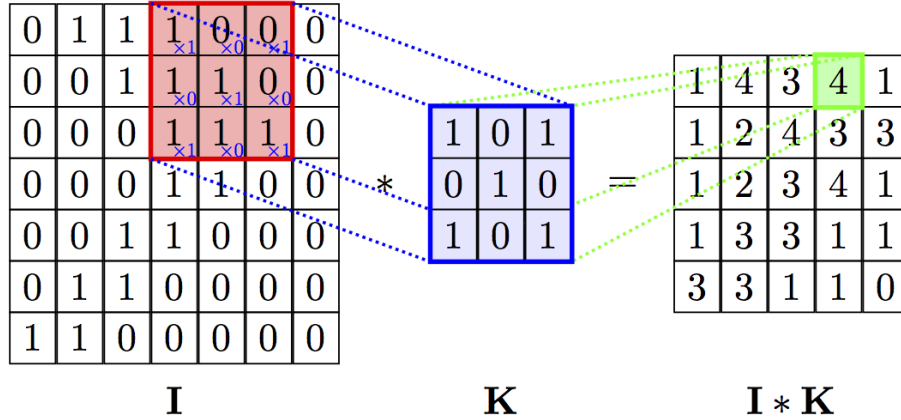
Outputs of all layers are activated with an activation function to introduce nonlinearity to the network. The rectifier linear unit (ReLU) is a simple nonlinear activation that turns negative values into zeros. The behavior of the ReLU is described as follows:

$$R(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0. \end{cases} \quad (2.3)$$

According to Bishop et al., the choice of activation function depends on the

Operation	Kernel $\omega$	Image result $g(x,y)$
<b>Identity</b>	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
<b>Edge detection</b>	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
<b>Sharpen</b>	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
<b>Gaussian blur 5 × 5</b> (approximation)	$\frac{1}{256} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$	
<b>Unsharp masking 5 × 5</b> Based on Gaussian blur with amount as 1 and threshold as 0 (with no image mask)	$\frac{-1}{256} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & -476 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$	

*Figure 2.3 Effects of different filter kernels. [33]*



*Figure 2.4 Convolution. [27]*

nature of the data and the assumed distribution of target data [2, p. 227]. However, in practice, ReLU was found by Krizhevsky et al. [20] to reduce model training times by a factor of 2 – 6×, while still allowing the parameters of a model to converge during training such that the network accuracy remains very high.

### 2.1.4 Softmax

In the last layer of a network, there is a softmax activation function. The purpose of the softmax activation is to convert values into action probabilities [30]. This is done by switching the elements of the activation vector from linear to logarithmic scale and updating each element to be a fraction of the total sum of values. The action probability  $\sigma(\vec{x})_j : \mathbb{R}^K \rightarrow \mathbb{R}^K$  for each output class  $j \in K$  is defined as follows,

$$\sigma(\vec{x})_j = \frac{\exp \vec{x}_j}{\sum_{k=1}^K \exp \vec{x}_k}, \quad (2.4)$$

where  $\vec{x} \in \mathbb{R}^K$  is the vector of output neurons before activation. The softmax activation function produces the output probabilities, or class predictions, of the network.

## 2.2 Model training and the pruning process

This section describes the process of model training. Model training refers to adjusting the values of weights towards the direction of least error with a gradient descent algorithm until a sufficiently good model is obtained. The complexity comes from determining how to calculate error effectively, and how to find the minimum for the error effectively using gradient descent. A description of determining the error for each layer for each input (2.2.1 Calculating training loss, 2.2.2 Backpropagation) is followed by a description of the gradient descent method (2.2.3 Gradient descent).

Finally, well-known, modern methods are introduced for improving the ability of the model to generalize (2.2.4 Regularization). Coincidentally, these regularization methods are also relevant to creating smaller and more efficient models. At the end of 2.2.3 Gradient descent, a summary of equations for the implementation of a simple deep neural network is provided.

## 2.2.1 Calculating training loss

### Quadratic cost

The carrying principle of supervised learning is that the weights of the model are shifted

iteratively towards the direction of least error w.r.t. ground truth, until the model is sufficiently accurate. This iterative process is governed by two main variables: the cost function and the learning rate. The cost function determines how divergence from the right answer is penalized, and the learning rate determines how fast the weights should be adjusted based on that. A learning rate that is too low causes the model to converge very slowly, while an overly high of a learning rate may cause the model to either "bounce" around a local minimum, or to diverge. A diverging training process is known as a gradient explosion.

A modified mean squared error (MSE), or the quadratic cost function is written as

$$\begin{aligned} C &= \frac{1}{2n} \sum_{i=1}^n (\vec{y}(\vec{x}_i) - \vec{a}_L(\vec{x}_i))^2 \\ &= \frac{1}{2n} \sum_{i=1}^n (\vec{y} - \vec{a}_L)^2, \end{aligned} \tag{2.5}$$

where  $n$  is the number of training samples  $\vec{x}_i$ .  $\vec{a}_L(\vec{x}_i)$  are the activations in the output layer (i.e. prediction). The ground truth  $\vec{y}(\vec{x}_i)$  are the vectors with known class labels, a.k.a. the right answers. The output activations  $\vec{a}_L(\vec{x}_i)$  can be calculated simply as one forward pass over the network, as was previously discussed at the end of Subsection 2.1.1.

Throughout this work, both the activations and the ground truth labels are expressed and implemented as one-hot encoding. This means that a true label of a domain with 4 separate classes could be expressed as a vector of 4 floating point probabilities  $\vec{y} = \langle 0, 0, 1, 0 \rangle$ .

The quadratic cost function is easy to understand. The original formula for the MSE is multiplied here by an extra  $\frac{1}{2}$  to make the expression reduce when differentiated with respect to weights. This operation does not affect the resultant

behavior of the network.

The practical interpretation of this is that to train the weights of a model, the network needs to be provided with training samples  $\vec{x}_i$ . Then the weights must be shifted effectively, such that the outputs  $\vec{a}_L(\vec{x}_i)$  move in the direction of least error with respect to ground truth. Mathematically, this can be interpreted as the derivative of the cost function w.r.t. the outputs:

$$\nabla_{\vec{a}}C = \frac{\partial}{\partial \vec{a}} \left\{ \frac{1}{2n} \sum_{i=1}^n (\vec{y} - \vec{a}_L)^2 \right\}. \quad (2.6)$$

This differential is easily obtained with the chain-rule  $(f \circ g)' = (f' \circ g)g'$ :

$$\begin{aligned} \nabla_{\vec{a}}C &= \frac{1}{2n} \sum_{i=1}^n \frac{\partial}{\partial \vec{a}} (\vec{y} - \vec{a})^2 \\ &= \frac{1}{2n} \sum_{i=1}^n 2(\vec{y} - \vec{a})(-1) \\ &= \frac{1}{n} \sum_{i=1}^n \vec{a} - \vec{y} \\ \nabla_{\vec{a}}C &= \vec{a} - \vec{y}. \end{aligned} \quad (2.7)$$

As is apparent, the gradient of the error can be calculated by subtracting the value of the true label from the outputs. The extra multiplier  $\frac{1}{2}$  causes the final expression of the derivative to simplify.

However, in multiclass classification problems with a softmax output activation, the cross entropy loss function is used to represent the distance between the target distribution and the view of the model of the distribution. Cross entropy cost function is explained in subsection 2.2.2.

## 2.2.2 Backpropagation

Once the cost function is decided, the error for each layer needs to be determined and differentiated w.r.t. the weights in each layer. An effective method of doing this is calculating the error in the final layer, and then backpropagating that error backwards through the network, reusing the intermediate derivatives obtained in the calculation of the previous layer. A description of the backpropagation process for determining the error in the rest of the layers follows.

## Calculating error

The error for the final layer is calculated first. Remembering that in a fully connected layer, the output before activation is  $\vec{z}_L = \mathbf{W}_L \times \vec{a}_{L-1}$ , the error for the final layer in a network of fully connected layers, can be defined as

$$\vec{\delta}_L = \nabla_{\vec{a}} C \odot \nabla \sigma(\vec{z}_L). \quad (2.8)$$

In Equation 2.8 and onwards,  $\odot$  signifies an elementwise vector product between the gradient of the cost function and the gradient of the softmax outputs  $\nabla \sigma(\vec{z}_L)$  of the previous layer.

For quadratic cost, the error in the final layer  $\vec{\delta}_L$  can be calculated based on Equation 2.8 by differentiating the cost function presented before in Equation 2.5, producing  $\nabla_{\vec{a}} C = \vec{a}_L - \vec{y}$ . The error in the final layer becomes:

$$\vec{\delta}_L = (\vec{a}_L - \vec{y}) \odot \nabla \sigma(\vec{z}_L). \quad (2.9)$$

This error  $\vec{\delta}_L$  is easily calculable. Here, the difference  $\vec{a}_L - \vec{y}$  is simply output activations subtracted by ground truth labels. Obtaining the gradient for the softmax function  $\nabla \sigma$  is not necessary. This is because the gradient can be simplified out of the expression via swapping the MSE cost function for the cross entropy cost function, explained later in this subsection.

The error in all other layers except the final one can be found by backpropagation [23]:

$$\vec{\delta}_l = (\vec{\delta}_{l+1} \times \mathbf{W}_{l+1}^\top) \odot \nabla R(\vec{z}_l), \quad (2.10)$$

where  $\nabla R$  is the gradient of the rectifier function.

## Cross entropy cost

If MSE is used as the loss function, initial learning is slow [23]. This is fairly intuitive from Equation 2.9 on page 16 for error in the final layer: the gradients of the softmax function tend to be a poor reflection of the perfect step sizes to find the next best estimate in backpropagation. In particular, when the outputs of the model are far from the ground truth, the gradient of the softmax function tends to be very flat, causing a slowdown in learning. This problem can be ameliorated by using cross entropy as the loss function [23]:

$$C = - \sum_{i=1}^n \vec{a}_L(\vec{x}_i) \log \vec{y}(\vec{x}_i), \quad (2.11)$$

where  $\vec{y}$  is ground truth, and  $\vec{a}$  is model output. Using one-hot encoding, the

magnitude of the vectors  $\vec{a}$  and  $\vec{y}$  is the same as the number of classes. Implementation of cross entropy cost and its derivative for vectors in one-hot encoding are provided in TENSORFLOW. Those implementations are used in the model under optimization.

### 2.2.3 Gradient descent

With the knowledge of how much each weight impacts the resultant cost  $\frac{\partial C}{\partial \mathbf{W}_i}$ , the weights should now be adjusted effectively towards the direction of least error. The classic approach to this optimization problem is called stochastic gradient descent (SGD). In SGD, each weight is updated each iteration  $i$  based on a learning rate  $\alpha$ :

$$\mathbf{W}_{i+1} = \mathbf{W}_i - \alpha \frac{\partial C}{\partial \mathbf{W}_i}. \quad (2.12)$$

Running this training iteration on a computer usually takes a lot of time. Iteration continues until it is interrupted, usually when the model converges.

In modern machine learning applications, SGD has largely been replaced by more recent and effective methods of gradient descent, such as the adaptive moment estimation method (Adam) by Kingma et al. [19]. The basic principle of adjusting the weights towards the least error still holds.

**Table 2.1** Equations for training a simple CNN.

Id	Equation	Eq.-#
<i>1. feedforward</i>		
node computations	$\vec{z}_l = \begin{cases} \vec{a}_{l-1} \times \mathbf{W}_l & , \text{ for fully connected} \\ \mathbf{W}_l \star \vec{a}_{l-1} & , \text{ for convolution} \end{cases}$	1.2, 2.2
activations	$\vec{a}_l = \vec{R}(\vec{z}_l), l \in [1, L - 1]$	1.2
final layer activations	$\vec{a}_L = \frac{\exp \vec{z}_{L-1}}{\sum \exp \vec{z}_{L-1}}$	2.4
<i>2. backpropagation</i>		
output error	$\vec{\delta}_L = \nabla_a C = \vec{a}_L - \vec{y}$	2.7
error	$\vec{\delta}_l = (\vec{\delta}_{l+1} \times \mathbf{W}_{l+1}^\top) \odot \nabla R(\vec{z}_l)$ $\frac{\partial C}{\partial \mathbf{W}_i} = \vec{a}_{l-1}^\top \times \vec{\delta}_i$	2.10
<i>3. gradient descent</i>		
weights update (SGD)	$\mathbf{W}_{i+1} = \mathbf{W}_i - \alpha \frac{\partial C}{\partial \mathbf{W}_i}$	2.12

Table 2.1 describes all the equations required to train a neural network consisting of fully connected layers and convolutional layers.

### 2.2.4 Regularization

When a network is trained on a set of training data, it can perform worse with images that it has never seen before. This is called *overfitting* to training data. One of the most effective ways of disrupting a network's tendency to overfit is to



prevent parameter co-adaptation with a technique called dropout. Dropout refers to deactivating random neurons for one forward pass during training, which prevents them from participating in backpropagation. This computationally inexpensive method causes the neurons of the network to learn to make more general "observations" on the data, independent of the presence of particular neurons in the surrounding layers. [13]

Another very common method of input regularization is called data augmentation. Data augmentation means expanding the initial set of labelled images by adding slightly transformed duplicates. Some such transformations include flipping, rotating, shifting and blurring.

In supervised learning, data is not infinite as it cannot be generated. As such, the same input data may be shown to the network multiple times over the course of a single training session. Each *set of iterations* over which the network sees the whole input data set is called an *epoch*.

Dropout is generally effective in increasing the ability of a single neuron to produce more generally useful output. In practice, this results in better ability of the resultant model to generalize, improving accuracy on test set. Dropout has been found by Krizhevsky et al. to approximately double the time that it takes their model to converge [20]. The resultant models are still large, and it has been found that further reductions in size can be achieved by eliminating a large number of parameters in the resultant network.

While the inputs of a network are determined by the data, and the outputs by the use case, the structure and number of hidden units can be adjusted to get the best network performance. The number of hidden units non-trivially influences the network's propensity for overfitting. A good practical solution is to choose a model with a high performance on a *validation* set. The validation set is a set of images chosen to be left out of the training set. [2, p. 256]

In this study, the best models are chosen based on validation set performance, and the reported accuracy is based on another set called the test set, which is also separated from the training data.

## **L2 regularization**

Overfitting can be controlled by reducing model complexity. A good approach for doing this is to first choose a relatively high number of hidden units, then pressure weights towards 0, i.e. penalizing model complexity. This can be achieved by adding a regularization term to the error function  $\vec{\delta}$ . [2, p. 256]

Two common techniques for penalizing model complexity are  $L^1$  regularization and  $L^2$  regularization. Results by Han et al. suggest that the  $L^1$  regularization gives better accuracy after pruning but before retraining, while  $L^2$  regularization

gives the best results after retraining. Both convolutional and fully connected layers can be pruned. By just pruning a network without retraining, they were able to achieve a  $2\times$  reduction in the number of connections. With retraining, the number of connections was reduced by  $9\times$  [11].

Bishop et al. describe the regularization term as

$$\tilde{\delta}(\mathbf{W}) = \bar{\delta}(\mathbf{W}) + \frac{\lambda}{2} \mathbf{W}^\top \mathbf{W}, \quad (2.13)$$

where  $\lambda$  is the regularization coefficient that defines the resultant effective model complexity.

Bishop et al. write that regularizers that are not invariant to linear transformations favor some equivalent models over others, so they introduce a regularizer that is invariant to linear transformation [2, p. 258]:

$$\frac{\lambda_l}{2} \sum_{w \in \mathbf{W}_l} w^2. \quad (2.14)$$

Correctly choosing the regularization term  $\lambda$  for  $L^2$  pruning is important. Choosing a too large  $\lambda$  results in a simple, underfit model, since more weights get squeezed to zero. Choosing a small lambda makes the model larger and more prone to overfitting. This can be seen as the accuracy on the validation set being larger than the accuracy on the test set.

In this thesis,  $L^2$  regularization is used in conjunction with masking of weights during training. This makes the fully connected layers have significantly fewer nonzero weights in total, effectively reducing model complexity.

## 2.3 Proposed optimizations

In this study, three strategies for reducing the model size of a pre-trained network were used: threshold pruning, pruning with  $L^2$  regularization, and separable convolutions. The effect on model size and performance was studied. Some parts of the network are more efficiently run on a graphics processor, and OpenCL was used to enable that.

### 2.3.1 Weights pruning

The aim of pruning weights is to reduce the total amount of values and computations required in a fully connected layer. Neural networks are often over-parameterized [7], which helps weights pruning be an effective technique in making the model smaller.

Weights pruning during training has long been used as a technique for regularization. In an early work by Le Cun et al. [22], they identified and dropped the least significant weights in a network to improve its ability to generalize in classification,

and to speed up training of the model. There has been a lot of recent research interest in reducing the model size by reducing the number of weights and connections. In 2015, Han et al. [11] reduced the number of parameters of AlexNet by a factor of  $9\times$  and of VGG-16 by a factor of  $13\times$  without loss in accuracy. They did this by removing connections below a threshold and retraining the network afterwards using  $L^2$  regularization. This process could be repeated multiple times to achieve final accuracy and to minimize network complexity. This is also one of the approaches used in this thesis.

### 2.3.2 Sparse models

A recent study by Zhu et al. [34] found that sparsified large models consistently outperform respective dense models of equal memory footprint across many domains. A particularly relevant class of optimizable models is that of the MobileNets [14], where Zhu et al. were able to reduce the number of parameters by 50 % with a 1.1 %-point reduction in top-1 accuracy and no reduction in top-5 accuracy.

Pruned weights matrices can be stored as sparse matrices. In this study, compressed row storage (CRS) format is used to represent a sparse matrix. In the format, the original matrix is encoded as non-zero (nnz) values and nnz indices along a selected axis. These compressed matrices require  $2a + n + 1$  values where  $a$  is the number of nnz elements and  $n$  is either the number of rows or the number of columns [10].

The format was probably first introduced in 1967 by Tinney and Walker [31] and first fully described by Buluç et al. [3]. Note that in this work, the CRS/CCS (compressed column storage) format is used instead of the more general compressed sparse blocks (CSB) format that Buluç et al. describe. The CRS format allows efficient computation of the  $Ax$  or the  $A^T x$  matrix-vector product where  $A$  is an  $m$  by  $n$  sparse matrix and  $x$  is a dense vector of length  $n$ . Either the  $Ax$  or the  $A^T x$  computation can be made efficient, depending on if CRS or CCS is used for matrix storage.

### 2.3.3 Separable convolutions

Using weights pruning to compress models has been found to be effective for the weights of the fully connected layers but less so for the kernels of the convolutional layers. Li et al. achieved inference cost reduction of 38 % on the CIFAR10<sup>2</sup> training set with negligible loss in accuracy by pruning out entire filters. He et al. were able to reach  $2\times$  speed-up on state of the art networks ResNet and Xception but

---

<sup>2</sup><https://www.cs.toronto.edu/~kriz/cifar.html>

with a significant 1.4 % and 1.0 % reduction in accuracy respectively [12]. Kernel factorization is an alternative to pruning entire channels or filters.

Jaderberg et al. [17] showed that the learned full rank kernels can be approximated by two rank-1 kernels. For a scene text character recognition application implemented in Caffe (CPU), this nets a  $4.5\times$  speedup with a 1 %-point reduction in accuracy or a  $2.5\times$  speedup with no degradation in accuracy. Their scheme 2 optimization seems to give best results in practice, as described by Jaderberg et al. [17]. They also showed that at least in their test case, the separable convolutions optimization scheme produces slightly better results than FFT based CNN optimizations. Scheme 2 means approximating a single convolutional kernel with two rank-1 basis kernels, one vertical and one horizontal:

$$\mathbf{S} \star \vec{x} = \vec{v} \star (\vec{h} \star \vec{x}).$$

Here,  $\star$  is again used to denote cross-correlation. Rank-1 kernels  $\vec{v}$  and  $\vec{h}$  are defined as

$$\vec{v}_k \in \mathbb{R}^{d \times 1 \times C} : k \in [1..K], \quad (2.15)$$

where  $K$  is a hand-picked dimension for an intermediate feature map, and

$$\vec{h}_f \in \mathbb{R}^{1 \times d \times K} : f \in [1..F], \quad (2.16)$$

where  $F$  is the number of feature maps in the original unoptimized network.

This optimization scheme reduces the complexity of computation from  $\mathcal{O}(k^2CFH'W')$  to  $\mathcal{O}(FKH'W')$ , where  $W'$  and  $H'$  are the dimensions of the output feature map.

In deep learning terms, each unoptimized convolutional layer is replaced by two decomposed rank-1 layers. The separated vertical convolution produces an intermediate feature map that is then convolved with a set of horizontal kernels. These kernels are learned by minimizing data reconstruction error, i.e. by aiming to reconstruct the outputs of the convolutional layers [17]. Jaderberg et al. minimize data reconstruction error as  $L^2$  error:

$$\min_{\{\vec{h}_{k,f}\}, \{\vec{v}_{c,k}\}} \sum_{i=1}^n \sum_{f=1}^F \left\| \mathbf{W}_f \star \vec{a}_{l-1}(\vec{x}_i) - \sum_{c=1}^C \sum_{k=1}^K \vec{h}_{k,f} \star \vec{v}_{c,k} \star \vec{a}_{l-1}(\vec{x}_i) \right\|^2. \quad (2.17)$$

Equation 2.17 can be intuitively understood as minimizing the mean squared error (or  $L^2$  error) between the outputs of the unoptimized convolutional layer and the outputs of the optimized layers. This operation is done layer by layer and the kernels can be optimized as part of backpropagation. Jaderberg et al. note

that just replacing the original convolutional layers with the optimized ones for backpropagation results in model overfitting. Trying to combat the problem with approaches such as dropout make the model underfit instead [17]. Thus, the best approach to train the kernels is by data reconstruction as in Equation 2.17.

This optimization is effective if  $K(F + C) \ll FCk$  [17]. For the network under study, in the first layer  $F = 32$ ,  $C = 3$ ,  $k = 3$ , and  $K$  is selected as  $K = 7$ . Thus  $K(F + C) < FCk \implies 245 < 288$ . In the second layer:  $F = 32$ ,  $C = 32$ ,  $k = 3$ , and  $K$  is selected as  $K = 7$ . Thus  $K(F + C) \ll FCk \implies 448 \ll 3072$ .

### Depthwise separable convolutions

In this study, Jaderberg's separable convolutions are used to optimize the convolutional layers. However, other similar approaches have been developed. An optimization called the depthwise separable convolution was first introduced in 2014 by Sifre [29]. This means separating the convolution operation into two steps: depthwise and pointwise convolution. This strategy was used successfully by Howard et al. in 2017 to optimize convolutional neural networks for resource constrained platforms [14]. In their paper, they also introduced two simple hyperparameters that could be used to trade off between network latency and accuracy. Networks optimized with this strategy are called MobileNets, and they are available as part of the TENSORFLOW software library.

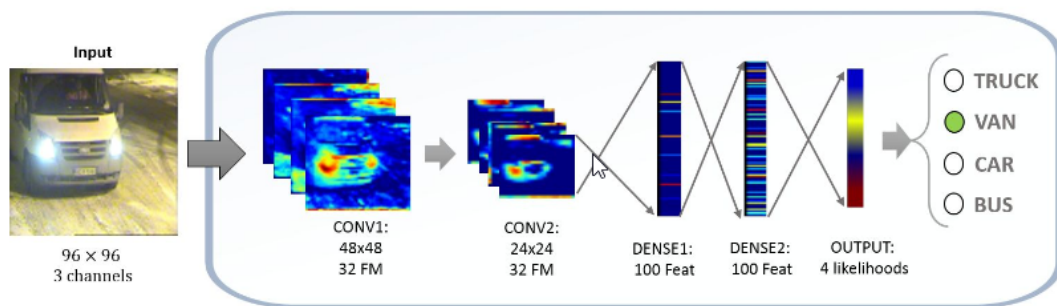
### 3 Implementation of optimizations

To compare the different optimizations described in Section 2.3, a convolutional neural network design by Huttunen et al. [15] was selected as a baseline implementation. The network is described in terms of dataflow programming, which facilitates easy altering of some aspects of the implementation.

Multiple versions of the different layers of the network were implemented and combined to get a view on how the optimizations stack up w.r.t. each other. In this chapter, an outline of the network to be optimized is first given, followed by descriptions of the implementations of these optimized layers and the final optimized networks formed of them.

#### 3.1 Use case: a convolutional network for car type recognition

The network under study is a network for vehicle classification (VCN) by Huttunen et al. [15]. This is a 5-layer<sup>1</sup> CNN with 1.88 million parameters, resulting in a model size of 7.18 MB. It was trained using 6555 sample images collected originally in collaboration with Visy Oy<sup>2</sup> [15]. An illustration of the general structure of the network is presented in Figure 3.1. The first two layers are convolutional layers with max pooling and ReLU as post-processing. The layers three and four are fully connected layers with ReLU post-processing. The last layer is fully connected, with softmax to process the values into action probabilities. There are 4 action probabilities, one for each type of vehicle: bus, car, truck, and van.



**Figure 3.1** Illustration of the vehicle classifier network by Huttunen et al. (©2016 IEEE)

The structure and inference implementation of the network for a single image can be particularly understandably expressed in a Matlab-like pseudo code as in

<sup>1</sup>The number of adaptive layers of weights is used to count the number of layers in a network [2, p. 229].

<sup>2</sup><https://visy.fi/>

Listing 1. Note that as an exception for this code sample, shapes are indexed in FORTRAN-order<sup>3</sup> as is done in Matlab code.

```

%% Non-trivial function explanations
% read_jpg(file):
%   reads a 3-channel jpg image from a `file` as 32-bit floats
% read_f32s(file):
%   reads a series of 32-bit floats from a `file`

% get input and weights
input_image = reshape(read_jpg('image.jpg'),
                      [96, 96, 3])
conv1_weights = reshape(read_f32s('w_conv1.bin'),
                       [5, 5, 3, 32])
conv2_weights = reshape(read_f32s('w_conv2.bin'),
                       [5, 5, 32, 32])
fc3_weights = reshape(read_f32s('w_fc3.bin'),
                     [24*24*32, 100])
fc4_weights = reshape(read_f32s('w_fc4.bin'),
                     [100, 100])
fc5_weights = reshape(read_f32s('w_fc5.bin'),
                     [100, 4])

% layer 1 activations
a1 = relu(convolve(input_image, conv1_weights))
a1_mxp = maxpool(a1)

% layer 2 activations
a2 = relu(convolve(a1_mxp, conv2_weights))
a2_mxp = maxpool(a2)

% layer 3 activations
a2_mxp_reordered = reshape(a2_mxp, [24*24*32, 1])
a3 = relu(a2_mxp_reordered * fc3_weights)

% layer 4 activations
a4 = relu(a3 * fc4_weights)

% layer 5 activations (softmax)
z5 = a4*fc5_weights
a5 = exp(z5)/sum(exp(z5))

```

**Listing 1** Matlab-like pseudo code for single-image inference.

The original network by Huttunen et al. was trained using backpropagation with

---

<sup>3</sup>FORTRAN order: most rapidly changing index first

learning rate 0.001643. The reported accuracy of the network is based on the weights selected at 30,000 iterations, though the accuracy of the network seemed to converge already in 15,000 iterations in 20 minutes on Nvidia Tesla K40t GPU. The dropout technique was utilized between each layer to reduce the effect of overfitting. [15]

During measurements described in Chapter 4, it was found that the first fully connected layer of the network is highly redundant. Pruning the fully connected layer revealed that many parameters can be removed, as a similar network accuracy can still be achieved with only around 1 in 1000 of the weights retained in the network. This makes it interesting to test if simple threshold pruning would be an effective method in quickly and easily eliminating a high amount of least impactful weights. It is also interesting to see how pruning with  $L^2$  regularization affects the network.  $L^2$  pruning was found effective by Han et al. in a very general case [11], and there is redundancy available in the network under testing. In the 2017 study, Zhu and Gupta also found that *large-sparse* networks like this tend to be more effective than their *small-dense* counterparts [34].

## 3.2 Implementation of training

The network under optimization is trained like the original network [15] using TensorFlow API, with weights for separated convolution attained by minimizing data reconstruction error, i.e. Jaderberg’s approach. The network is then pruned with either of the two pruning strategies described next.

### 3.2.1 Pruning

Two different pruning strategies are used. In threshold pruning, some weights are periodically and permanently set to zero based on a threshold. These weights are masked out of the network for the remainder of training. Threshold  $T(t_o)$  is defined as

$$T(t_0) = \min(\mathbf{W}^l) + t_0(\max(\mathbf{W}^l) - \min(\mathbf{W}^l)), \quad (3.1)$$

where the threshold parameter  $t_0$  represents pruning sensitivity. Setting pruning sensitivity to 0.0 would mean no pruning, and setting it to 1.0 would mean pruning everything, leaving the model empty. After pruning, the model is retrained until convergence.

In  $L^2$  pruning, the model is regularized, masked as in threshold pruning, and retrained. It was found via experimentation that early stopping each retraining at 10 epochs allows the model to fit to the training data. Regularization, masking and retraining is chosen to be applied in steps as follows:

1. load trained weights,



2. repeat in 10 stages:
  - (a) eliminate parameters below halfway-through the current weight range:  

$$T = \frac{1}{2}(\max(\mathbf{W}^l) + \min(\mathbf{W}^l)),$$
  - (b) mask the zeroed weights out of the network,
  - (c) retrain or tune for 10 epochs with  $L^2$ -regularization using regularization term  $\lambda$ .

After each tuning step, the model is accepted if its validation accuracy has degraded less than 1 %-point from accuracy of the baseline initial unpruned model on the validation data set. Note that the validation accuracy on the augmented data set may differ from the baseline accuracy reported previously by Huttunen et al. [15]. Tuned models are saved as long as the validation accuracy keeps increasing.

### 3.2.2 Assessment of model accuracy and overfitting

The images used for training the network are separated into three sets: training (90 %, 14108 images), validation (6 %, 898 images) and test set (4 %, 655 images). The model is first fitted into the training set over a set amount of epochs. Next, the accuracy of the model on the validation set is assessed, and the accuracy on the test set is stored with the validation accuracy. The best validation accuracy is always kept, and the respective test accuracy is what is reported. Divergence of test accuracy from validation accuracy is also used to assess overfitting.

## 3.3 Inference optimizations

Every algorithm is run on a specific device. The structure and function of a computational device strongly affects how a layer transformation can be applied most efficiently. The modern computer processing unit works fastest when input values are read in the order they are used in the computation, organized into blocks of appropriate size. In terms of computer memory, an array that is organized like this without any extra space in between values is said to be contiguous.

A modern CPU is often capable of applying one operation to multiple contiguous elements, i.e. single instruction, multiple data (SIMD) of Flynn’s taxonomy [25]. Modern graphics processors (GPUs) use a similar model: single instruction, multiple threads (SIMT). Processors can often combine multiple mathematical operations together, eg. the multiply-accumulate (MAC) operation and fused multiply-add (FMA). To leverage these features, some vendors have created their own APIs and toolkits that can be used to optimize the program code for a particular device. However, in this thesis, only cross-platform optimization schemes are used: the LLVM compiler infrastructure and OPENCL.

Inference code and benchmarks were written in RUST and OPENCL. RUST is a relatively new, open-source, systems programming language by Mozilla Foundation. RUST is compiled with RUSTC of the LLVM compiler infrastructure. Part of the inference code was written in OPENCL, that was then compiled by the driver on each target device. OPENCL is a computing language by Khronos Group, Inc., used for describing algorithms for GPUs and other highly parallelized processors.

The inference works in two steps: initialization and execution. During initialization, the OPENCL devices are selected and the OPENCL code is compiled by the driver for the selected devices. Then the network graph is built into an OPENCL command queue, and the weights are loaded into the OPENCL buffers.

After the timed execution starts, the input image buffer is loaded onto the GPU, and the OPENCL command queue is allowed to execute. After operations, `queue.wait()` is called on the command queue to wait for the output buffers to flush and the operations to complete. An implementation for manually executing the inference graph across multiple devices is given in Listing 2.

In the inference implementation of Listing 2, input is mapped to the GPU buffer. This code is considered "unsafe" in RUST, because it uses an underlying C-API to control OPENCL, and the compiler cannot verify the memory safety of that code. After mapping inputs, the OPENCL kernels for convolutions 1 and 2 are run. These can be run on GPU or CPU based on instructions given per measurement. The buffer is then copied to the secondary device, which may be a GPU or a CPU. This is a no-op if the primary and secondary devices are the same. Next, the output buffers are copied onto the host memory for application of the final two fully connected layers. On the last line, softmax is applied to outputs, and the result is returned.

### 3.3.1 Sparse matrix multiplication

Training and pruning of weights was implemented using the TENSORFLOW API via PYTHON. An optimized inference runtime<sup>4</sup> was implemented in RUST. The main component of the optimized inference runtime is the sparse matrix storage format. The sparse matrix storage format was first described by Tinney and Walker [31] and it is used as implemented in the community library RUST crate `sprs`<sup>5</sup>.

In the sparse storage format, a matrix containing both zero and nonzero values (nnz) can be represented as three one-dimensional arrays. The arrays are as follows:

1. nnz values,
2. an accumulating counter of nnz values by row in original matrix, padded with a zero in the beginning,

---

<sup>4</sup><https://github.com/hegza/vcn-inference-rs>

<sup>5</sup><https://github.com/vbarrielle/sprs>, license: MIT / Apache 2.0

```

let mut event_list = EventList::new();

// GPU ops are unsafe
unsafe {
    map_to_buf(&input_buf, input_data).unwrap();

    // Enqueue the kernel for the 1st layer (Convolution + ReLU)
    conv1_kernel.cmd().enq().unwrap();
    // Enqueue the kernel for the 2nd layer (Convolution + ReLU)
    conv2_kernel.cmd().enq().unwrap();

    // Copy GPU buffer to host
    conv2_out_buf
        .copy(&dense3_in_buf, None, None)
        .enew(&mut event_list)
        .enq()
        .unwrap();

    // Enqueue the 3rd layer (fully connected)
    dense3_kernel.cmd().ewait(&event_list).enq().unwrap();
}

// Wait for all on-device calculations to finish
cl.cpu_queue.finish().unwrap();

// Read buffer into host memory
let dense3_out = &unsafe { read_buf(&dense3_out_buf).unwrap() };
// Run the 4th layer (fully connected)
let dense4_out = relu(dense4.compute(&dense3_out));
// Run and return the 5th layer (fully connected + softmax)
softmax(dense5.compute(&dense4_out))

```

**Listing 2** VCN inference code.

3. column indices of nnz values.

Consider a matrix with zeroes like

$$\begin{bmatrix} 0 & 1 & 0 \\ 7 & 3 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

This matrix would be stored in three arrays as follows:

```

/// computes matrix multiplication in_buf * wgts,
/// when wgts is column-compressed
pub fn sprs_mtx_mul(
    wgts: &sprs::CsMat<f32>,
    in_buf: &[f32],
    out_buf: &mut [f32]) {

    let mat = wgts.view();

    // creates an iterator over the columns of the matrix
    // and associated indices
    for (col_idx, vec)
        in mat.outer_iterator().enumerate() {
        let mut acc = 0f32;
        // creates an iterator over the rows of the matrix
        for (row_idx, &value) in vec.iter() {
            acc += in_buf[row_idx] * value;
        }
        out_buf[col_idx] = acc;
    }
}

```

**Listing 3** Calculating  $v \times M$  in RUST with sprs.

```

// nonzero values
let nnz = [ 1, 7, 3 ];

// number of accumulated nonzeros by row
let nnz_counter = [ 0, 1, 3, 3 ];

// column indices of nonzeros
let col_idxxs = [ 1, 0, 1 ];

```

This matrix is stored in the `sprs` library as `CsMat`. In practice, the transpose of the weights matrix needs to be computed instead of the original matrix. This is compensated for by using column compression instead of row compression. In a column compressed matrix, row indices are stored in the third array instead of column indices. This also changes `nnz` values to run from top-to-down then left-to-right. A  $\vec{v} \times \mathbf{M}$  product can be computed using the RUST code of Listing 3.

### 3.3.2 Separable convolutions

Weights for separable convolutions were trained using the approach by Jaderberg et al. by minimizing data reconstruction error in `TENSORFLOW`. Intuitively, this means that the aim is to train the separated rank-1 kernels to produce feature maps that are as similar as possible to the feature maps produced by the original kernels.

Training is started by having the set of the original trained weights, and a set of untrained weights for the separable convolutions. The difference between the feature maps generated by each is then minimized using gradient descent, as shown in Equation 2.17 in Subsection 2.3.3.

These trained weights are then loaded in the `RUST / OPENCL` inference code for benchmarking. `OPENCL` code for separable convolutions is provided at <https://github.com/hegza/vcn-inference-rs/blob/master/src/cl/sepconv.cl>.

## 4 Measurements

Network performance was measured on multiple different representative platforms. Platform specifications are presented in Table 4.1.

**Table 4.1** *Platforms by given name.*

<b>Name</b>	<b>CPU</b>	<b>GPU</b>	<b>Operating system</b>
AMD desktop	Phenom II X6 1090T	Radeon HD 7800	Windows 10
Intel i7	Intel i7-2640M	N/A	Arch Linux
Mali	ARM Cortex-A72×2/-A53×4	ARM Mali T860	Linux Firefly 4.4

Of the platforms presented in Table 4.1, AMD desktop is a typical desktop computer with a mid-tier GPU with 2 Gb of global memory and an OPENCL work group size of 256. Intel i7 is an a typical mid-tier laptop and Mali is a representative embedded platform, used in single-board computers. SoCs of similar performance are used in contemporary smartphones, like the Samsung Exynos 5422 chip in Samsung Galaxy S5.

The structures of the trained networks are described in Table 4.2.

**Table 4.2** *Network structures by given name.*

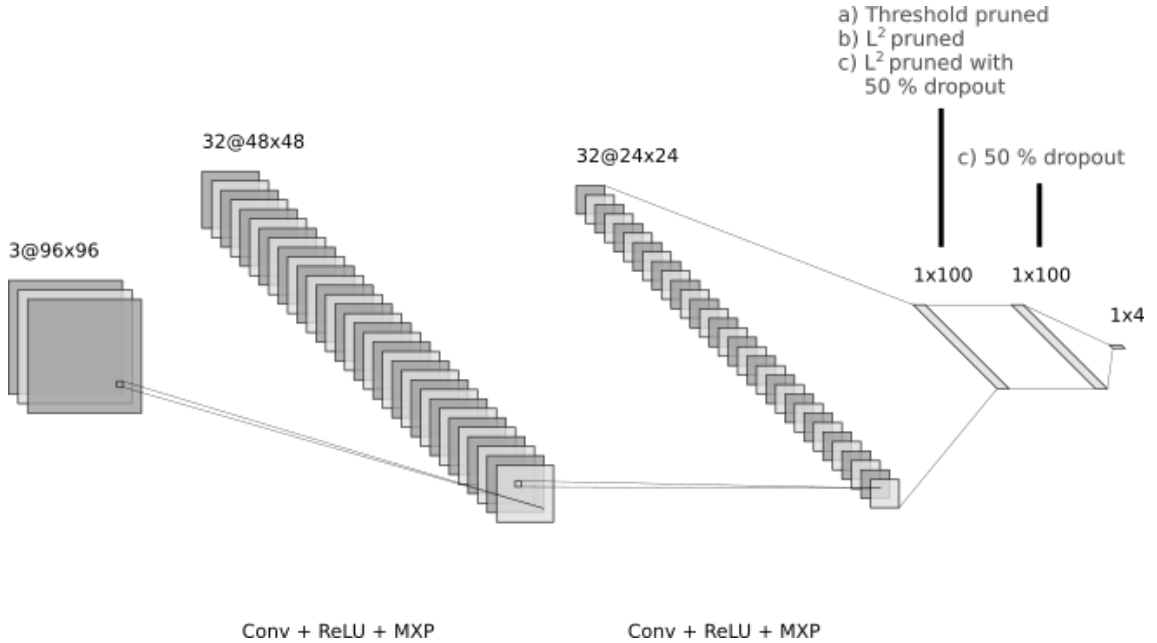
<b>Name</b>	<b>L1</b>	<b>L2</b>	<b>L3</b>	<b>L4</b>	<b>L5</b>
VCN	Conv	Conv	FC	FC	FC
VCN-sepconv	SepConv	SepConv	FC	FC	FC
VCN-sparse	Conv	Conv	Sparse	FC	FC

*Conv* is the baseline implementation of cross-correlation between weights and inputs. *SepConv* is the optimized separable convolution implementation. Fully connected layer (*FC*) is the baseline matrix multiplication implementation between weights and inputs, and *Sparse* is the pruned fully connected layer. Activations are rectifier linear unit (ReLU) or softmax.

### 4.1 Data collection

A set of weights was first trained to baseline accuracy ( $\geq 97\%$ ) in TENSORFLOW, using a training set of 6555 images augmented to 14108 images by flipping and Gaussian blurring some of the images at random. The set of images was then separated to 12555 training images, 898 validation images, and 655 test images, which were used as-is for  $L^2$  pruning methods, but with changes for threshold pruning, as detailed in Subsection 4.1.1. The trained models were then pruned with three strategies. The first strategy used threshold pruning that clamps the values below the threshold  $T$  of the range of the weight to be zero, and then eliminates them from

further training. The second strategy used  $L^2$  pruning, as explained in Section 2.2.4. The third strategy used  $L^2$ -pruning with 50 % dropout in the pruned layer and the one following it. The pruned models were then retrained to the training set, always storing the model with the best validation accuracy, and the respective accuracy on the test set. After all steps were completed, overfit was assessed using the test set that was left out of all other steps. The networks were pruned with multiple pruning thresholds and  $L^2$ -coefficients  $\lambda$ , to obtain an idea of if and how the regularization term  $\lambda$  affects parameter count, and how the parameter count affects network accuracy. Strategies are further illustrated in Figure 4.1.

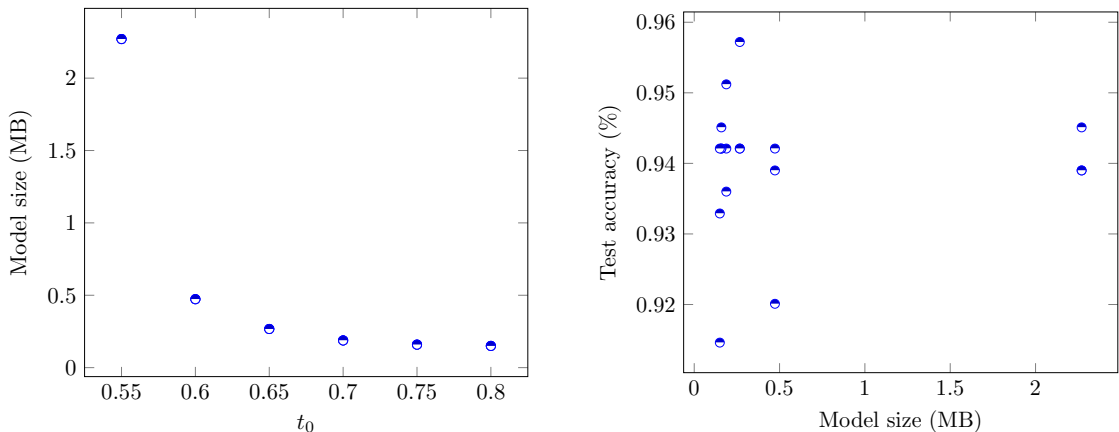


**Figure 4.1** Illustration of the network under pruning. **a)** threshold pruning, **b)**  $L^2$  pruning, **c)**  $L^2$  pruning with 50 % dropout. [24]

Accuracy on test set is reported for all models, and accuracy on validation set is reported for  $L^2$  pruning methods. To assess model overfit, divergence of test accuracy from validation accuracy is measured as "val-test". The value of overfit is calculated by subtracting test accuracy from validation accuracy. Overfit is only reported for  $L^2$  pruned models. On threshold pruning, validation accuracy was used to select the best model like for  $L^2$  pruning, but it was not stored for later retrieval. Test accuracy for threshold pruning can still be used to evaluate effectiveness of strategy in creating an accurate, compressed model. Baseline model accuracy and size were retrieved from the study [15] by Huttunen et al. All pruned model sizes account for CCS storage overhead. All pruning data points are included as Appendix A.

### 4.1.1 Threshold pruning

In threshold pruning, the trained model was first pruned to a reduced size based on a threshold. The pruned model was then retrained to the training set. Issues with computer memory led to the elimination of the validation set and to the use of test set for accuracy evaluation in such a way that half of the test set of the 328 images were used for validation, and 327 were used for test. This led to increased variance in final model metrics for threshold pruning, and reduced performance of best model selection algorithm (sa. Section 4.1), but the principle remained the same. Reported accuracy is the accuracy of the model on the test set, which was not used in selecting the best model. Using AdamOptimizer for gradient descent, it seemed that the model was fit on the training set after 10 epochs, and the validation accuracy started to fall-off after 13 epochs, suggesting overfitting. Threshold pruned models were thus only retrained for 10 epochs. Figure 4.2 shows model size by pruning sensitivity  $t_0$  on the left, and test accuracy by model size on the right. Models with less than 300 parameters underfit on the training set and were left out of the measurements, reflected in Figure 4.2 of results ( $t_0 > 0.80$ ,  $\lambda > 0.40$ ).



**Figure 4.2** Left: model size by  $t_0$  for the threshold pruned model. Right: test accuracy by model size.

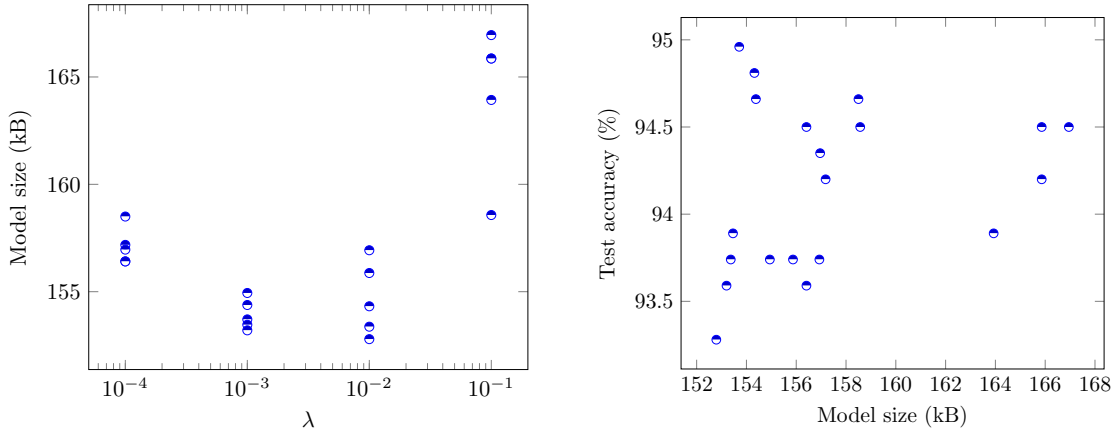
Model size has a relatively small impact on model accuracy with  $t_0 \leq 0.80$ . Test accuracy is  $< 96\%$  for all models produced by threshold pruning. Model size may slightly correlate with model accuracy at this parameter range. Some of the smallest of the models were inaccurate, and the model was not able to recover on training set at  $\lambda < 0.55$ .

### 4.1.2 L2 pruning

In  $L^2$  pruning, weights below zero were pruned, then the network was retrained for 10 epochs with  $L^2$  regularization with regularization term  $\lambda$ . This process was

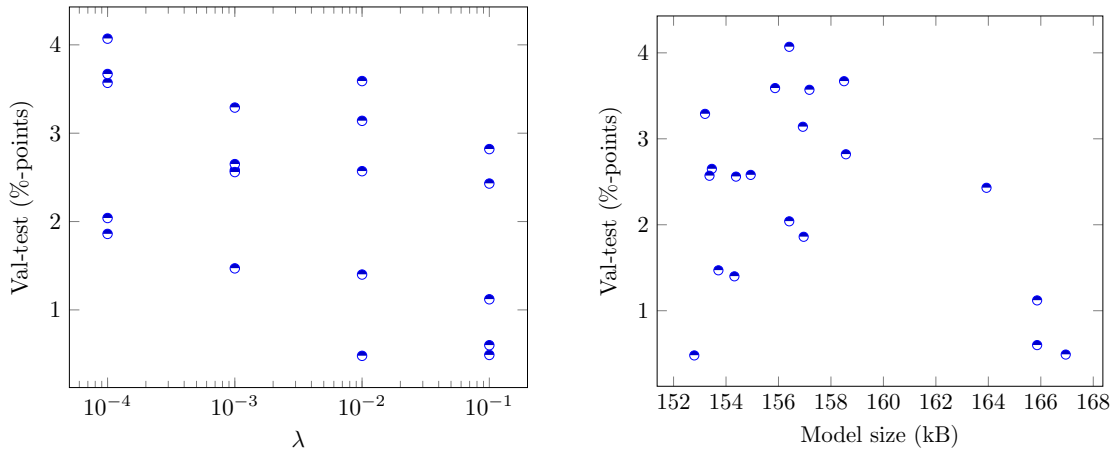


repeated 10 times. Figure 4.3 shows model size by regularization term  $\lambda$  on the left, and test accuracy by model size on the right. Figure 4.4 shows how  $\lambda$  and model size correlate with overfit.



**Figure 4.3** Left:  $L^2$  pruned model size by regularization term  $\lambda$ . Right: test accuracy by model size.

The regularization term seems to have a quadratic correlation with model size. A larger regularization term  $\lambda$  generally pressures the model to be smaller, but after a point, the high regularization factor starts gimping the ability of the model to recover and it starts becoming harder to eliminate parameters from the model.

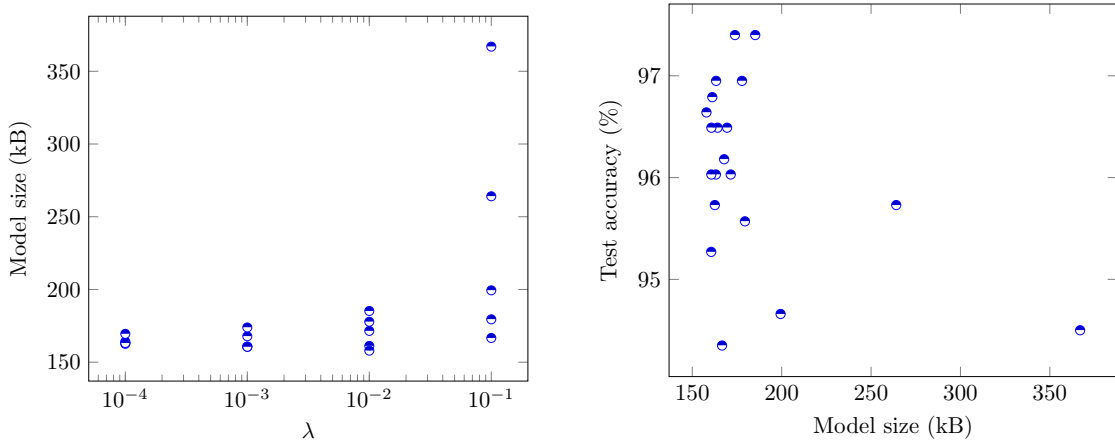


**Figure 4.4** Left:  $L^2$  pruned model overfit by regularization term  $\lambda$ . Right: overfit by model size.

Increase in regularization term  $\lambda$  seems to correlate negatively with model overfit. Stronger regularization may reduce overfit. Model size seems to have a quadratic relationship with overfit for this model.

### 4.1.3 L2 pruning with 50 % dropout

Figure 4.5 shows model size by regularization term  $\lambda$  on the left, and test accuracy by model size on the right. Figure 4.6 shows how  $\lambda$  and model size correlate with overfit.



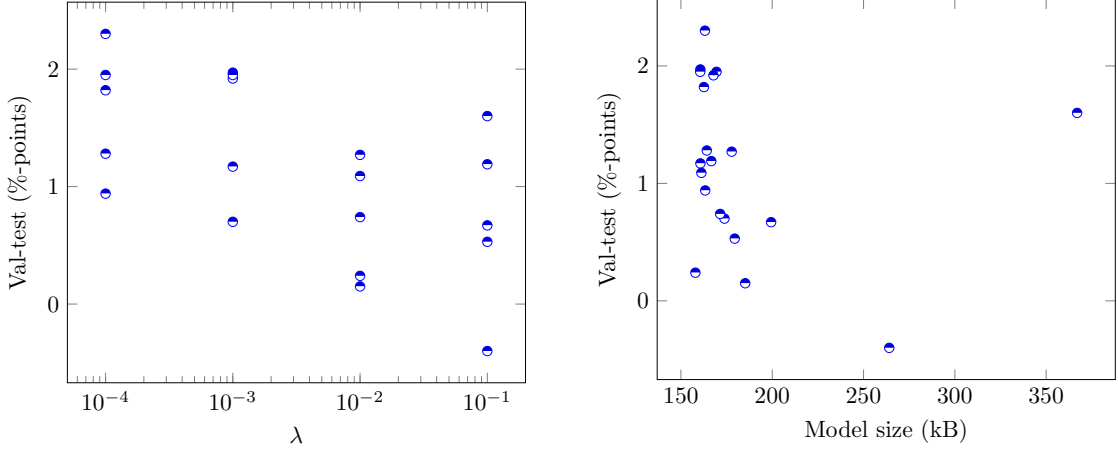
**Figure 4.5** Left:  $L^2$  pruned with dropout model size by regularization term  $\lambda$ . Right: test accuracy by model size.

Increase in regularization term  $\lambda$  seems to correlate with model size. The implication is that a high regularization term sometimes gimps the ability of the model to recover while using dropout. Because the model accuracy is generally slightly higher than that of threshold pruning with similar compression factors, it seems that a small amount of regularization is helpful for compression of the model without loss in accuracy. However, results are inconclusive without a more rigorous statistical analysis.

When model size is the smallest, either test accuracy is the highest, or overfit is the smallest. One outlier produced a negative overfit, undoubtedly by luck.

Increase in regularization term  $\lambda$  seems to correlate negatively with model overfit. Stronger regularization may reduce the ability of the model to fit when used with dropout.

These three pruning strategies all caused the model to lose test accuracy on average. On average, threshold pruning lost 3.78 %-points,  $L^2$  pruning lost 3.6 %-points and  $L^2$  pruning with dropout lost 1.67 %-points.  $L^2$  pruning with dropout produced the smallest, the most accurate, and the least overfit models. Threshold pruning seems to have no effect on accuracy until the point of collapse at  $t_0 > 0.1$ . Assessment of which of the  $L^2$  pruned models would be the best for a given application would be difficult. Picking the model based on the best test accuracy gives a model that happens to score well on that test set. Picking the model with the highest validation accuracy gives the most overfit model. Assessment of result goodness can



**Figure 4.6** Left:  $L^2$  pruned with dropout model overfit by  $\lambda$ . Right: overfit by model size.

therefore be made based on the average results of the method. Strategy performance averages are shown in Table 4.3. Averages were calculated based on the results presented in Appendix A.

**Table 4.3** Averages of measurements on pruned models. Baseline model is provided for reference.

Strategy	Validation accuracy (%)	Test accuracy (%)	Over-fit (%)	Parameters in 3rd layer	Model size (kB)
VCN	-	97.75	-	1,843,200	7180
Threshold pruning	-	93.97	-	-	-
$L^2$ pruning	96.48	94.15	2.32	907	157
$L^2$ pruning with dropout	97.24	96.08	1.15	4279	184

On average,  $L^2$  pruning with with 50 % dropout produced  $39.1\times$  smaller models (183.82 kB) with  $431\times$  less parameters in the third layer (4279 on average) with 1.67 %-point reduction in accuracy. Values for model size are missing in Table 4.3 because parameter count can be arbitrarily chosen as long as the threshold  $t_0 \leq 0.8$ .

#### 4.1.4 Performance measurements

Alternative implementations for the fully connected layer were measured to find a good baseline implementation for the model. The implementations compared were:

- naive host: a double for-loop matrix multiplication in RUST,
- OPENCL (CPU): a vectorized single-loop OPENCL FMA implementation on

the host CPU as presented at [https://github.com/hegza/vcn-inference-rs/blob/master/src/cl/mtx\\_mul.cl](https://github.com/hegza/vcn-inference-rs/blob/master/src/cl/mtx_mul.cl),

- `matrixmultiply`: the most used RUST library for matrix multiplication <sup>1</sup>,
- `cnugteren 10`: an efficient OpenCL matrix multiplication algorithm adapted from an online tutorial<sup>2</sup>.

Measurements were taken using a benchmark suite of 100 samples over 5050 iterations. Values reported are means after elimination of outliers. Table 4.4 shows that the vectorized OPENCL implementation of matrix multiply for the CPU was the lowest in latency (sa. [https://github.com/hegza/vcn-inference-rs/blob/master/src/cl/mtx\\_mul.cl](https://github.com/hegza/vcn-inference-rs/blob/master/src/cl/mtx_mul.cl)).

**Table 4.4** *Alternative implementations for fully connected layer.*

Implementation	Latency (ms)	
	AMD desktop	Intel i7
Naive host	3.47	2.03
OpenCL (CPU)	0.615	0.615
<code>matrixmultiply</code>	3.21	1.60
<code>cnugteren 10</code>	5.37	-

The weights were then input into an inference runtime written in RUST. The OPENCL code was compiled by the OPENCL driver for a target device, and the Rust code gets compiled into native CPU code as is the case with C. For comparison, the baseline variant of the network was run against a version written in C. Results are presented in Table 4.5. The C-version was compiled with `gcc -o main -std=c11 -O3` with `-lm`, `-lOpenCL`, and associated flags as required per platform. The OPENCL implementation targets the GPU on the AMD desktop and the CPU on the i7. All implementations use the same timing strategy: the clock is started after loading the inputs into RAM or after inputs have been mapped to the GPU, and before dispatching the first kernel. The clock is stopped after the final values are in RAM.

**Table 4.5** *Inference for one image, C and Rust baselines.*

Implementation	Inference time / image (ms)		Speedup over C	
	AMD desktop	Intel i7	AMD desktop	Intel i7
C	297	47.9	N/A	N/A
C (OpenCL)	2.70	10.6	110×	4.52×
Rust (OpenCL)	1.87	23.0	159×	2.08×

<sup>1</sup><https://github.com/bluss/matrixmultiply>

<sup>2</sup><https://cnugteren.github.io/tutorial/pages/page1.html>

The GPU-utilizing OPENCL versions on AMD desktop run a lot faster than the plain C version. The OPENCL implementation also attains a speedup running on the i7.

Separating convolutions with  $K_1 = 7$ ,  $K_2 = 7$  to produce a  $2.5\times$  reduction to layer-wise computation complexity reduced accuracy to 96.49 %. The best sparse weights provide an accuracy of 97.40 %. These two pretrained optimized networks were then run on the three devices. An efficient OPENCL implementation was used as baseline (VCN). VCN-sepconv uses separable convolution as explained in Subsection 2.3.3. VCN-sparse uses a sparse model created with  $L^2$  pruning as explained in Subsections 2.3.1 and 2.3.2. Results are presented in Table 4.6. Results were obtained by timing the network execution time over the test set of 655 images and then dividing the total time by 655.

**Table 4.6** Per-image inference time by platform, sampled 100 of 5050 runs.

Platform	Execution time (ms) / speedup			
	VCN (97.75 %)	VCN-sepconv (97.3 % <sup>3</sup> )		VCN-sparse (97.40 %)
AMD desktop	1.87	1.46	1.28 $\times$	0.931 2.01 $\times$
Intel i7	23.0	7.67	3.00 $\times$	29.8 0.77 $\times$
Mali <sup>4</sup>	120	80	1.50 $\times$	66 1.82 $\times$

While measuring the i7 performance on the whole test set of 655 images, there was a significant drop in performance of around 30 % when compared to running the network on a single image only. This is most likely due to CPU throttling during benchmark runs. Additionally, all inputs were black-boxed from the compiler on benchmark runs to prevent cache preloads outside the bench loop.

A network with the sparse CCS matrix storage as the third layer was implemented and compared to the baseline model. Results are shown in Table 4.7.

**Table 4.7** Execution time of VCN and VCN-sparse, and the sparse layer on AMD desktop. Share of sparse layer of total time is provided as a percentage.

Implementation	Execution time	
	of network	of third layer (standalone)
VCN	1.87 ms	615 $\mu$ s 32.9 %
VCN-sparse	0.931 ms	13.7 $\mu$ s 1.4 %

With the sparse layer, the model is compressed by  $46.6\times$ . The smaller size makes the network latency twice as low on desktop. As can be seen in Table 4.7, the pruning process makes the third layer so small that computing it becomes almost free on desktop (13.7  $\mu$ s).

## 4.2 Analysis of results

These results are compared to previous results in other studies. The original model for vehicle classification by Huttunen et al. was obtained via optimizing the hyperparameters with the aim of producing the model with best predictive power [15]. However, their hyperparameter space only allowed for fully connected layers, while large-sparse models have seemed to produce better results [34]. Additionally, Jaderberg et al. have shown that separable convolutions are an effective technique in speeding up convolutions with sometimes negligible loss.

### 4.2.1 Pruning for model size and latency reduction

All tested pruning methods were effective in reducing model size to a fraction of the original ( $\geq 42.1\times$ ), while  $L^2$  pruning with dropout was the most accurate while being the least overfit. Notably, returning the original unpruned model to original accuracy was difficult with the newly chosen training, validation, and test sets. Reorganizing of the data set may have contributed to the resultant accuracy.

Finding such a threshold  $t_0$  for threshold pruning that would result in acceptable loss in accuracy was difficult with the used data set, as the number of test images was low enough to cause a relatively high variance in resultant test accuracy. However, many of the parameters were found to be low-impact for the model in the first place, and cutting off more than 75 % of the weights can be an effective and easy way of reducing model size, provided that the model is retrained to the training set afterwards.

$L^2$  pruning by itself did not retain enough test accuracy either and was overfit. The deficiency in number of test images may have contributed to loss in accuracy. Combining  $L^2$  pruning with dropout allowed for controlling the overfit while allowing the model to converge. The model retained accuracy well considering the compression. The number of parameters was reduced from 1.84 million to around 2-3 thousand. That is a  $720\times$  reduction in parameter count in the first fully connected layer layer.

In addition to model size reduction, pruning allows for a faster inference runtime. The latency of the pruned 3rd layer is  $44.9\times$  lower when compared to the fully connected layer on AMD desktop. Compared to the baseline OPENCL-accelerated model, sparse format improved latency by  $2.01\times$  on AMD desktop, and  $1.82\times$  on Mali as convolutions then became the bottleneck. Sparse format did not improve latency on i7 ( $0.77\times$ ).

In the 2015 study by Han et al. [11], the number of parameters in AlexNet was reduced by  $10\times$ , and the number in VGG-16 by  $13\times$  with no reduction in accuracy. In 2016, AlexNet level of accuracy was achieved with SqueezeNet and  $50\times$  less parameters. In 2017, state-of-the-art models were pruned by Zhu et al. [34] for

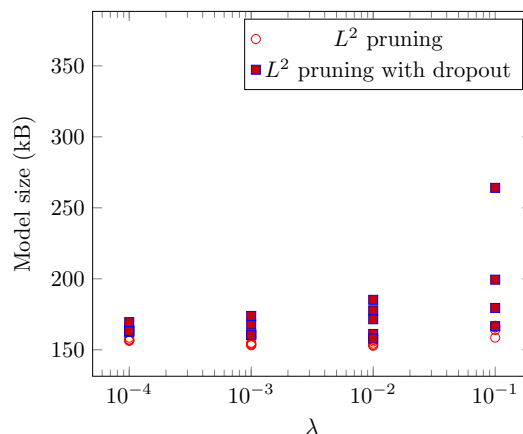
compression factors of  $10\times$ , while the number of nnz parameters of the network under study were reduced by a factor of  $46.6\times$ . One possible interpretation is that the third layer of VCN is considerably overparameterized. This may still be taken as further evidence towards the usability of sparse models for effectively representing more complex dense models as shown by Zhu et al. [34].

Similar compression factors were achieved on this model, as have been achieved on state-of-the-art models. The total model size was reduced to 170.72 kB with  $L^2$  pruning with dropout, which is similar in magnitude with SqueezeNet with its 500 kB model size. Many less optimized models would likely benefit from a simple  $L^2$  pruning with dropout followed by sparse storage. These results may also indicate that purposefully creating overparameterized models and then aggressively pruning them may also be a good approach to creating effective and small deep learning models. This agrees with results by Zhu et al. [34].

The fastest implementation for the fully connected layer in this network was the simple OPENCL implementation on CPU. Using that implementation, the baseline OPENCL optimized model latency is 1.87 ms for the AMD desktop and 23.0 ms for i7, compared to the 297 ms and 47.9 ms of the plain C-version. This is a  $159\times$  speedup on AMD desktop with GPU and a  $2.08\times$  speedup on i7 over the original model. OPENCL seems to be an effective accelerator for this purpose.

#### 4.2.2 Impact of dropout on pruning with L2 regularization

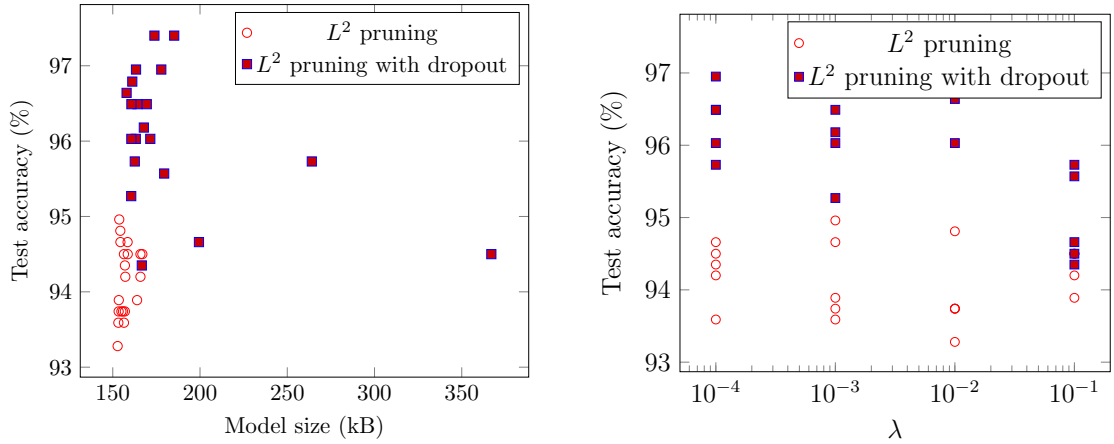
In this subsection, the influence of dropout on pruning results is illustrated. Figures combining results from both variants of  $L^2$  pruning show how dropout improves the results of  $L^2$  pruning. Figure 4.7 shows how model size is dependent on the chosen regularization term  $\lambda$  for both variants of  $L^2$  pruning.



**Figure 4.7** Model size by regularization term  $\lambda$  for both variants of  $L^2$  pruning.

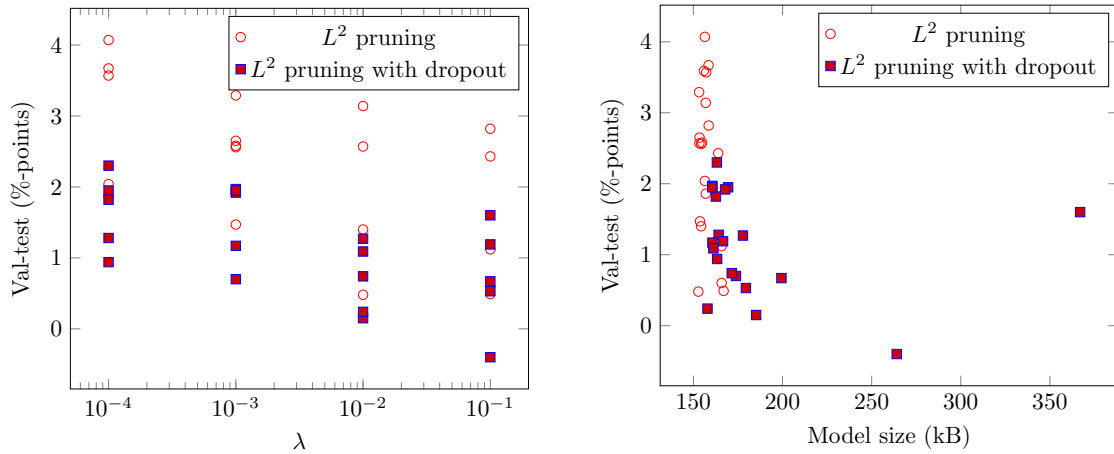
As can be seen in Figure 4.7, dropout tends to make resultant model size larger when combined with  $L^2$  pruning. Figure 4.8 show how the accuracy of the model

behaves for different model sizes and  $\lambda$  parameters.



**Figure 4.8** Left: test accuracy by model size for both variants of  $L^2$  pruning. Right: test accuracy by  $\lambda$  for both variants of  $L^2$  pruning.

Test accuracy is generally higher for models that use dropout. Best trade-off between model accuracy and size seems to come with dropout, with correctly selected regularization term  $\lambda$ . Results are inconclusive as to whether regularization is helpful or not. Measurements on model retraining with 0 regularization would reveal this. Figure 4.9 shows how dropout impacts overfit for regularization term  $\lambda$  and model size.



**Figure 4.9** Overfit in both variants of  $L^2$  pruning. Left: overfit by  $\lambda$ . Right: overfit by model size.

Figure 4.9 shows how overfit tends to be lower for models that were retrained with dropout. All in all, not using dropout tends to result in smaller models, but with the cost of higher amounts of overfitting.



### 4.2.3 Latency across devices

Compared to the baseline OPENCL-accelerated model, separable convolutions improve latency by  $1.28\times$  on AMD desktop,  $3.00\times$  on i7, and  $1.50\times$  on Mali. Pruning improved latency by  $2.01\times$  on desktop and  $1.82\times$  on Mali. Pruning did not improve latency on i7 ( $0.77\times$ ).

Separable convolutions appear to improve latency ( $3.00\times$  for i7,  $1.50\times$  for Mali) on platforms without a dedicated GPU for a very small reduction in accuracy ( $\approx 0.45$  %-points). These are similar to the original results by Jaderberg et al. It is reasonable to assume that their optimization scheme affects their network in a very similar way to how it affects the VCN, by reducing latency by reducing the total number of computations. The achieved speedup is dependent on hardware: higher on a platform without a GPU.

On i7 without GPU, the separable convolutions runtime is the fastest while on AMD desktop, the pruned runtime is the fastest. This is because on platforms without a GPU, convolutions become a bottleneck while the fully connected layer is otherwise the bottleneck.

### 4.2.4 Combining pruning and separable convolutions

It should be possible to combine pruning with separable convolutions, though no such experiment was carried out in this thesis. For comparison, Lane and Bhattacharya [1] sped up VGG by  $13.3\times$  on embedded platforms with a similar approach. Speedups with Jaderberg’s separable convolutions and pruning stay within  $2 - 3\times$  due to bottlenecking. Combining these approaches should in theory result in a speedup of  $\approx 4.5\times$  on the desktop platform. While accuracy cannot be guaranteed without experimentation, there’s no reason to believe that the network would not be able to recover from both optimizations via retraining. Further experimentation would clarify the issue.

### 4.2.5 Reliability of results

The whole set of images was divided into a test set of 655 images, a validation set of 898 images and a training set of 14108 images. The low amount of validation and test images produces a greater variance in the performance metric, especially for threshold pruning with the modified validation and test sets. Because there were many measurements taken, choosing a higher number of images for test and validation sets would have produced more consistent results. However, it is fairly clear that pruning with dropout produces most accurate non-overfitting models.

As presented in Subsection 4.1.4, Table 4.4, the OPENCL implementation of the first fully connected layer only took a suspicious  $615 \mu s$  to run on OPENCL on both

desktop and i7. This matches with the OPENCL kernel dispatch time. However, verifying and benchmarking the whole network on known inputs and results seems to suggest that the times are reasonable. The likely explanation is that OPENCL is able to take advantage of the known size of the inputs, and the fully connected layer gets executed using vectorized multiply-adds in a small number of steps. This makes it run in the time it takes to dispatch the kernel.

### 4.3 Summary of findings

$L^2$  pruning with  $\lambda = 0.01$  and dropout reduces total model size by  $42.1\times$  on average with minor trade-off in accuracy (0.99 %-points on average).  $L^2$  pruning with dropout in general reduces accuracy on average by 1.67 %-points, while reducing model size by  $39.1\times$ . Threshold pruning can easily reduce the model size down to around 180 kB with a high variance baseline reduction in accuracy of 1–6 %-points, possibly partially due to validation/test set selection. Latency of the pruned layer is reduced by  $44.9\times$  on AMD desktop. This brings the latency of the whole network to a  $2.00\times$  speedup on AMD desktop and to a  $1.82\times$  speedup on Mali. Model size was brought from the original 7.18 MB to 173.87 kB by use of  $L^2$  pruning with dropout. These reductions indicate that either the first fully connected layer is severely overparameterized or that large-sparse networks have some intrinsic advantage over small-dense networks. Results are in close agreement with results by Zhu et al. [34].

Separable convolutions improve latency by  $1.28\times$  on AMD desktop,  $3.00\times$  on i7, and  $1.50\times$  on Mali, while reducing accuracy by 0.45 %-points. Results agree with results by Jaderberg et al. [17]. Devices with less GPU-like processors benefit from separable convolutions the most.

## 5 Conclusions

$L^2$  pruning with dropout is clearly the best out of the measured pruning methods, though it's inconclusive whether  $L^2$  regularization helps or interferes with the result, while dropout seems instrumental. Using sparse storage of pruned models can produce great improvement in memory footprint during inference, if the number of prunable weights is high. Sparse models can also eliminate the CPU bottleneck of the fully connected layer, when the size of the unpruned model is excessively large. In this case, a speedup of  $44.9\times$  was achieved for the sparse layer. Based on this small sample, GPU platforms may benefit slightly from use of separable convolutions, though mileage may vary and sometimes the results are worse. However, separable convolutions can be used to accelerate convolutions on CPU platforms when using a GPU is not feasible.

### 5.1 Discussion

Studies by others have shown that  $L^2$  pruning and sparse models seem to be a good alternative for compressing neural networks [10, 11, 34]. This study has shown that the techniques are applicable to the CNN [15] by Huttunen et al. When comparing the effectiveness of  $L^2$  pruning on this network and state-of-the-art networks, the technique is similarly effective. This might be taken as further evidence that the principles can be applied to other similar networks as well. Application is likely to be easy and effective for any convolutional neural network with a large fully connected layer. Latency is also improved, if the fully connected layer is a computational bottleneck.

With the application of  $L^2$  pruning, loss in accuracy is expected. Overfitting and resultant reduction in test accuracy can be ameliorated by use of dropout. It's up to the user and use case whether the reduction in accuracy is acceptable or not. Use of pruning alone should reduce overfitting of the model, and dropout can be used to augment the effect.

There is also another possible interpretation for the high compression factor, and the ability of the model to recover from losing most of the weights. It is possible that the third layer of the network by Huttunen et al. [15] needed not be so large in the first place. A counterargument to this is provided by Zhu et al. In their study [34], they claim that large-sparse models consistently outperform small-dense models. Considering that, the pruned model is likely to compare favorably with a model trained from scratch with a smaller third layer.

Separable convolutions always reduce the total number of computations required,

and thus increase general power efficiency. For devices with less GPU-like processors, separable convolutions can also effectively improve latency. With the wide adoption of convolutional networks for image processing tasks, edge computing applications would surely benefit from the optimization.

Another possible use for the optimizations is to create a minimum-latency model by applying both techniques on a network. Combining  $L^2$  pruning with separable convolutions should be achievable. There is no obvious obstacle to such a model performing well, if the model manages to recover and compensate for both approximations.

## 5.2 Recommendation

Pruning with dropout seems to be a good strategy for creating a considerably smaller model with minimal loss in accuracy. Of course, problem domain affects parameter optimization, and these results should not be taken as proof that all problem domains can be optimized to work so well with large-sparse models. Pruning does appear to reduce accuracy slightly.

Separable convolutions can be used to increase power efficiency of hardware running convolutional neural networks. Separable convolutions are a particularly good fit for devices that lack GPU-like processors.

The original model by Huttunen et al. [15] can be compressed from 7.18 MB to around 180 kB by pruning the first fully connected layer. Most effective pruning results with 0.99 %-point reduction in accuracy were achieved by incrementally pruning all weights below halfway of weight range per step, with  $L^2$  regularization with  $\lambda = 0.01$ , and by using 50 % temporary dropout on the remaining weights in the first and second fully connected layer during retraining.

## 5.3 Future work

Investigating the interaction between dropout and  $L^2$  pruning may help understand whether  $L^2$  pruning is necessary, or if threshold pruning with dropout works as-is. A combination of pruning and separable convolutions may be attempted to further reduce model size. This combination might also improve latency in applications where either or both form a bottleneck. Combining optimizations into a single model may improve performance metrics with acceptable trade-off in accuracy.

## References

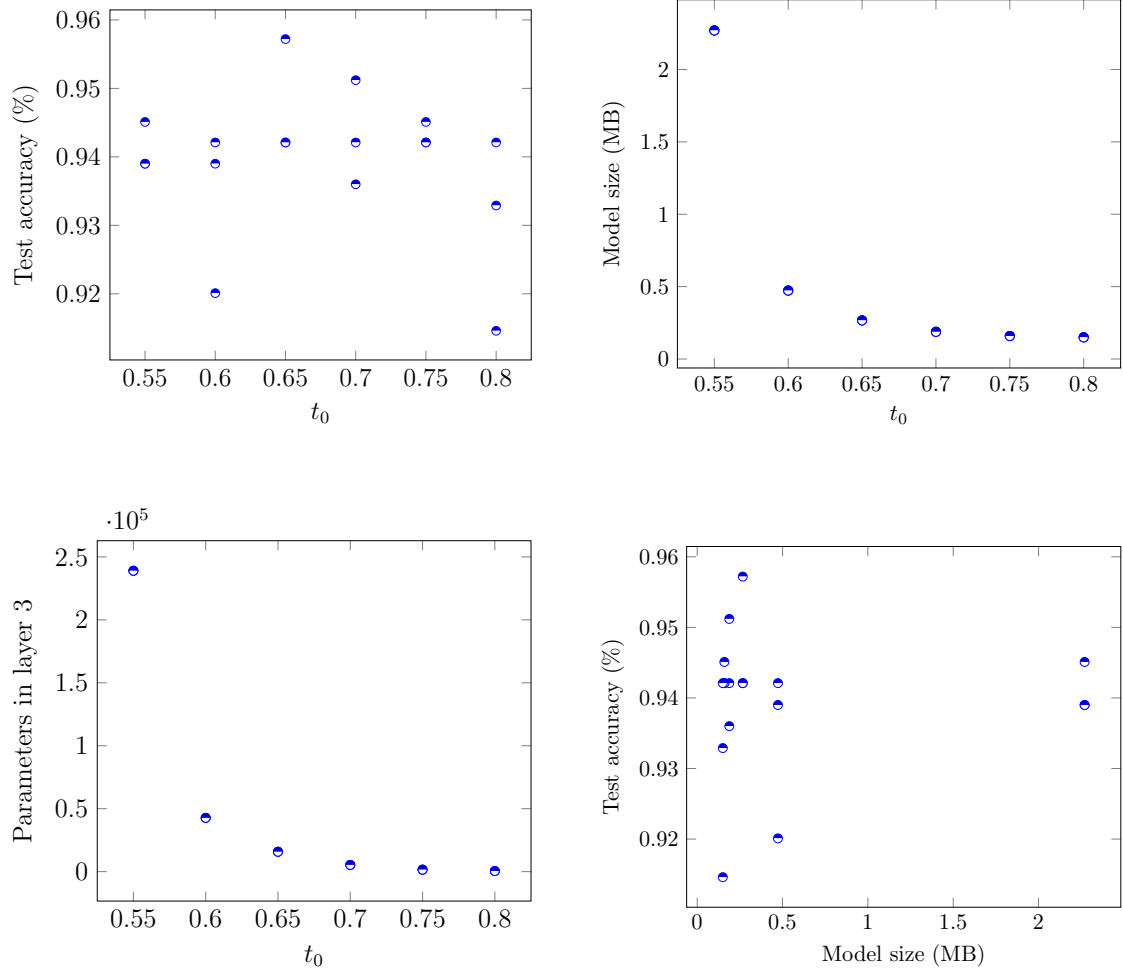
- [1] Sourav Bhattacharya and Nicholas D Lane. “Sparsification and separation of deep learning layers for constrained resource inference on wearables”. In: *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*. ACM. 2016, pp. 176–189.
- [2] Christopher M. Bishop. *Pattern recognition and machine learning*. New York: Springer, 2006.
- [3] Aydin Buluç et al. “Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks”. In: *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. ACM. 2009, pp. 233–244.
- [4] Kumar Chellapilla, Sidd Puri, and Patrice Simard. “High performance convolutional neural networks for document processing”. In: *Tenth International Workshop on Frontiers in Handwriting Recognition*. Suvisoft. 2006.
- [5] Dan Claudiu Cireşan et al. “Flexible, high performance convolutional neural networks for image classification”. In: *Twenty-Second International Joint Conference on Artificial Intelligence*. 2011.
- [6] Dan Claudiu Cireşan, Ueli Meier, and Jürgen Schmidhuber. “Multi-column deep neural networks for image classification”. In: *arXiv preprint arXiv:1202.2745* (2012).
- [7] Misha Denil et al. “Predicting parameters in deep learning”. In: *Advances in neural information processing systems*. 2013, pp. 2148–2156.
- [8] Kjell Magne Fauske. *TeXample.net. Example: Neural network*. [Online; MIT License; accessed November 8, 2019]. URL: <http://www.texample.net/tikz/examples/neural-network/>.
- [9] Richard HR Hahnloser et al. “Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit”. In: *Nature* 405.6789 (2000), p. 947.
- [10] Song Han, Huizi Mao, and William J Dally. “Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding”. In: *arXiv preprint arXiv:1510.00149* (2015).
- [11] Song Han et al. “Learning both weights and connections for efficient neural network”. In: *Advances in neural information processing systems*. 2015, pp. 1135–1143.
- [12] Yihui He, Xiangyu Zhang, and Jian Sun. “Channel pruning for accelerating very deep neural networks”. In: *Proceedings of the IEEE International Conference on Computer Vision*. 2017, pp. 1389–1397.

- [13] Geoffrey E Hinton et al. “Improving neural networks by preventing co-adaptation of feature detectors”. In: *arXiv preprint arXiv:1207.0580* (2012).
- [14] Andrew G Howard et al. “Mobilenets: Efficient convolutional neural networks for mobile vision applications”. In: *arXiv preprint arXiv:1704.04861* (2017).
- [15] Heikki Huttunen, Fatemeh Shokrollahi Yancheshmeh, and Ke Chen. “Car type recognition with deep neural networks”. In: *Intelligent Vehicles Symposium (IV), 2016 IEEE*. IEEE. 2016, pp. 1115–1120.
- [16] Forrest N Iandola et al. “SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and 0.5 MB model size”. In: *arXiv preprint arXiv:1602.07360* (2016).
- [17] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. “Speeding up convolutional neural networks with low rank expansions”. In: *arXiv preprint arXiv:1405.3866* (2014).
- [18] Tero Karras, Samuli Laine, and Timo Aila. “A style-based generator architecture for generative adversarial networks”. In: *arXiv preprint arXiv:1812.04948* (2018).
- [19] Diederik P Kingma and Jimmy Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).
- [20] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.
- [21] Nicholas D Lane et al. “An early resource characterization of deep learning on wearables, smartphones and internet-of-things devices”. In: *Proceedings of the 2015 international workshop on internet of things towards applications*. ACM. 2015, pp. 7–12.
- [22] Yann LeCun, John S Denker, and Sara A Solla. “Optimal brain damage”. In: *Advances in neural information processing systems*. 1990, pp. 598–605.
- [23] Michael A. Nielsen. *Neural Networks and Deep Learning*. 2015.
- [24] *NN-SVG*. [Online; accessed November 8, 2019]. URL: <http://alexlenail.me/NN-SVG/LeNet.html>.
- [25] “Flynn’s Taxonomy”. In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Boston, MA: Springer US, 2011, pp. 689–697. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4\_2. URL: [https://doi.org/10.1007/978-0-387-09766-4\\_2](https://doi.org/10.1007/978-0-387-09766-4_2).
- [26] Taesung Park et al. “GauGAN: semantic image synthesis with spatially adaptive normalization”. In: *ACM SIGGRAPH 2019 Real-Time Live!* ACM. 2019, p. 2.

- [27] Petar Veličković. *2D Convolution*. [Online; MIT License; accessed September 17, 2019]. 2014. URL: <https://github.com/PetarV-/TikZ/tree/master/2D%20Convolution>.
- [28] Mark Sandler et al. “Mobilenetv2: Inverted residuals and linear bottlenecks”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 4510–4520.
- [29] Laurent Sifre and Stéphane Mallat. “Rigid-motion scattering for image classification”. PhD thesis. Citeseer, 2014.
- [30] Richard S Sutton, Andrew G Barto, Francis Bach, et al. *Reinforcement learning: An introduction*. 1998. URL: <http://incompleteideas.net/book/ebook/the-book.html>.
- [31] William F Tinney and John W Walker. “Direct solutions of sparse network equations by optimally ordered triangular factorization”. In: *proc. IEEE* 55.11 (1967), pp. 1801–1809.
- [32] Wikipedia user Aphex34. *Typical CNN architecture*. [Online; CC-BY-SA-4.0; accessed August 19, 2019]. 2015. URL: [https://en.wikipedia.org/wiki/Convolutional\\_neural\\_network#/media/File:Typical\\_cnn.png](https://en.wikipedia.org/wiki/Convolutional_neural_network#/media/File:Typical_cnn.png).
- [33] Wikipedia users Michael Plotke and MortenZdk. *Various titles*. [Online; CC-BY-SA-3.0; accessed September 17, 2019]. 2013–2016. URL: [https://en.wikipedia.org/wiki/Kernel\\_\(image\\_processing\)](https://en.wikipedia.org/wiki/Kernel_(image_processing)).
- [34] Michael Zhu and Suyog Gupta. “To prune, or not to prune: exploring the efficacy of pruning for model compression”. In: *arXiv preprint arXiv:1710.01878* (2017).

# APPENDIX A. Results of model pruning

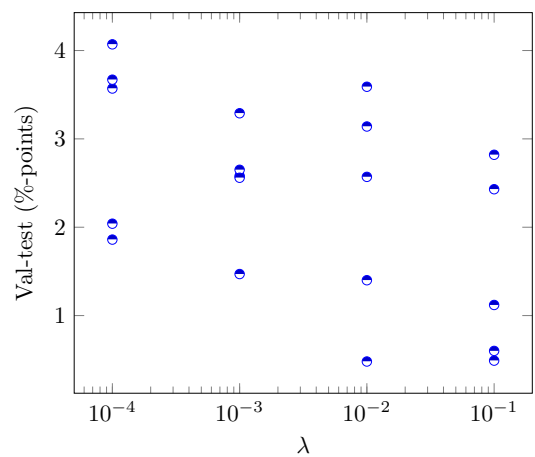
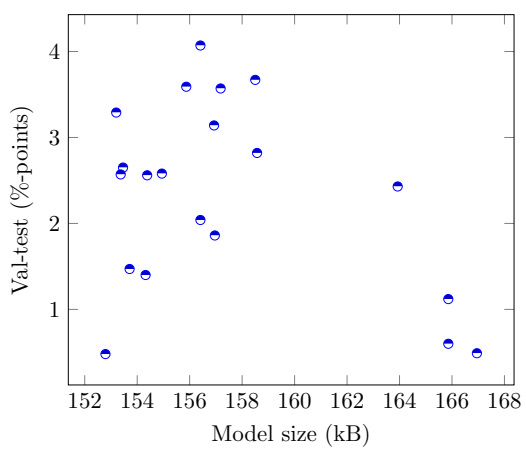
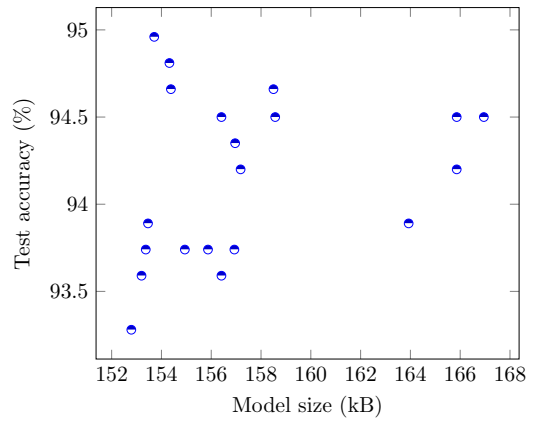
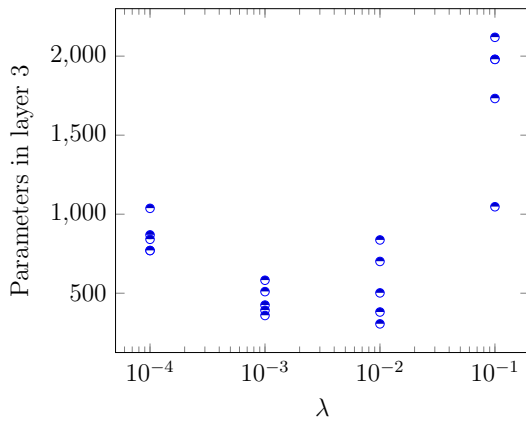
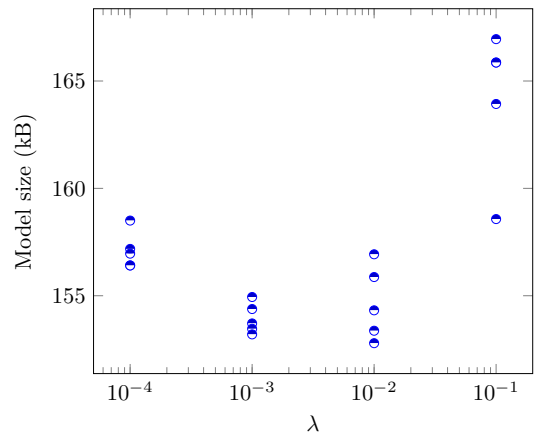
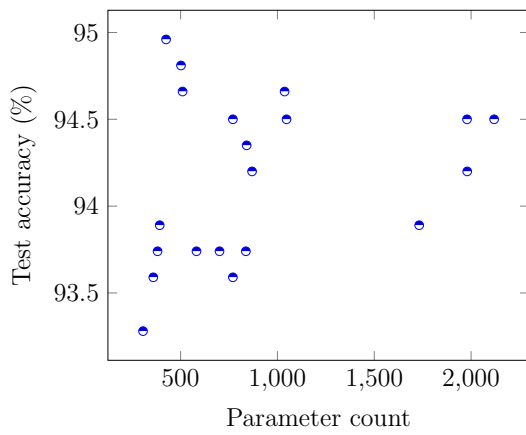
## Results of threshold pruning





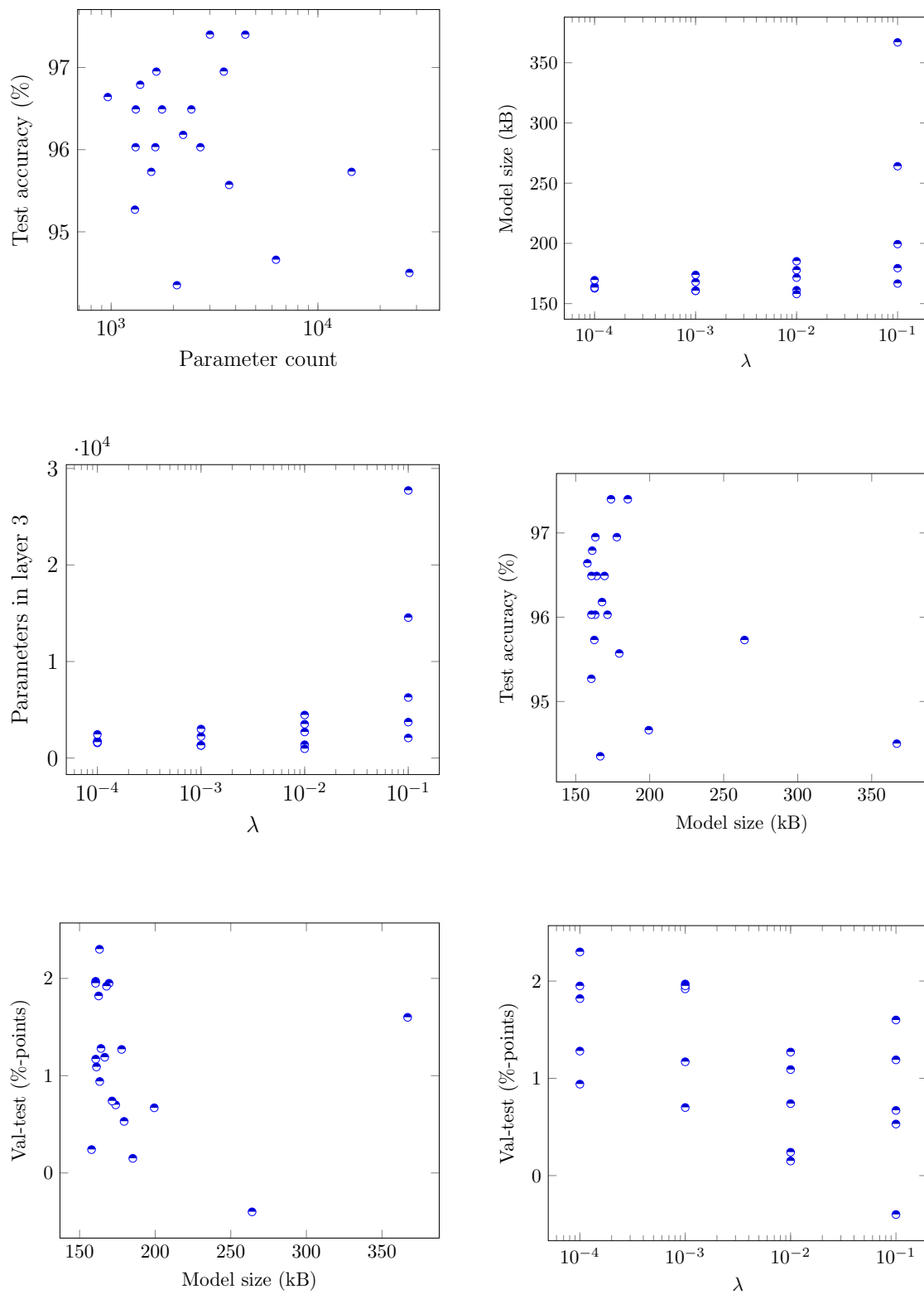
$t_0$	test accuracy	size (kB)	parameters in 3rd layer
0.55	0.9390	2.270	$2.39 \cdot 10^5$
0.55	0.9390	2.270	$2.39 \cdot 10^5$
0.55	0.9451	2.270	$2.39 \cdot 10^5$
0.60	0.9390	0.473	42,700
0.60	0.9421	0.473	42,700
0.60	0.9201	0.473	42,700
0.65	0.9421	0.267	15,800
0.65	0.9572	0.267	15,800
0.65	0.9421	0.267	15,800
0.70	0.9421	0.188	5,340
0.70	0.9512	0.188	5,340
0.70	0.9360	0.188	5,340
0.75	0.9421	0.159	1,570
0.75	0.9451	0.159	1,570
0.75	0.9421	0.159	1,570
0.80	0.9146	0.150	470
0.80	0.9421	0.150	470
0.80	0.9329	0.150	470

## Results of L2 pruning



$t_0$	parameters	validation	test		
	in 3rd layer	accuracy (%)	accuracy (%)	size (kB)	val-test
$1 \cdot 10^{-4}$	869	97.77	94.20	157.18	3.57
$1 \cdot 10^{-4}$	841	96.21	94.35	156.96	1.86
$1 \cdot 10^{-4}$	1,037	98.33	94.66	158.50	3.67
$1 \cdot 10^{-4}$	770	97.66	93.59	156.41	4.07
$1 \cdot 10^{-4}$	770	96.55	94.50	156.41	2.04
$1 \cdot 10^{-3}$	582	96.33	93.74	154.94	2.58
$1 \cdot 10^{-3}$	425	96.44	94.96	153.71	1.47
$1 \cdot 10^{-3}$	392	96.55	93.89	153.46	2.65
$1 \cdot 10^{-3}$	510	97.22	94.66	154.38	2.56
$1 \cdot 10^{-3}$	359	96.88	93.59	153.20	3.29
$1 \cdot 10^{-2}$	502	96.21	94.81	154.32	1.4
$1 \cdot 10^{-2}$	837	96.88	93.74	156.93	3.14
$1 \cdot 10^{-2}$	306	93.76	93.28	152.79	0.48
$1 \cdot 10^{-2}$	381	96.33	93.74	153.37	2.57
$1 \cdot 10^{-2}$	701	97.33	93.74	155.87	3.59
0.10	1,047	97.33	94.50	158.57	2.82
0.10	1,732	96.33	93.89	163.93	2.43
0.10	1,980	95.32	94.20	165.86	1.12
0.10	2,119	94.99	94.50	166.95	0.49
0.10	1,979	95.10	94.50	165.86	0.6

## Results of L2 pruning with 50 % dropout



$t_0$	parameters	validation	test		
	in 3rd layer	accuracy (%)	accuracy (%)	size (kB)	val-test
$1 \cdot 10^{-4}$	869	97.77	94.20	157.18	3.57
$1 \cdot 10^{-4}$	841	96.21	94.35	156.96	1.86
$1 \cdot 10^{-4}$	1,037	98.33	94.66	158.50	3.67
$1 \cdot 10^{-4}$	770	97.66	93.59	156.41	4.07
$1 \cdot 10^{-4}$	770	96.55	94.50	156.41	2.04
$1 \cdot 10^{-3}$	582	96.33	93.74	154.94	2.58
$1 \cdot 10^{-3}$	425	96.44	94.96	153.71	1.47
$1 \cdot 10^{-3}$	392	96.55	93.89	153.46	2.65
$1 \cdot 10^{-3}$	510	97.22	94.66	154.38	2.56
$1 \cdot 10^{-3}$	359	96.88	93.59	153.20	3.29
$1 \cdot 10^{-2}$	502	96.21	94.81	154.32	1.4
$1 \cdot 10^{-2}$	837	96.88	93.74	156.93	3.14
$1 \cdot 10^{-2}$	306	93.76	93.28	152.79	0.48
$1 \cdot 10^{-2}$	381	96.33	93.74	153.37	2.57
$1 \cdot 10^{-2}$	701	97.33	93.74	155.87	3.59
0.10	1,047	97.33	94.50	158.57	2.82
0.10	1,732	96.33	93.89	163.93	2.43
0.10	1,980	95.32	94.20	165.86	1.12
0.10	2,119	94.99	94.50	166.95	0.49
0.10	1,979	95.10	94.50	165.86	0.6