UPPSALA
UNIVERSITET

UPTEC F 21021
Examensarbete 30 hp
Juni 2021

Optimizing Convolutional Neural Networks for Inference on Embedded Systems

Lucas Strömberg

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Optimizing Convolutional Neural Networks for Inference on Embedded Systems

Lucas Strömberg

Convolutional neural networks (CNN) are state of the art machine learning models used for various computer vision problems, such as image recognition. As these networks normally need a vast amount of parameters they can be computationally expensive, which complicates deployment on embedded hardware, especially if there are contraints on for instance latency, memory or power consumption. This thesis examines the CNN optimization methods pruning and quantization, in order to explore how they affect not only model accuracy, but also possible inference latency speedup.

Four baseline CNN models, based on popular and relevant architectures, were implemented and trained on the CIFAR-10 dataset. The networks were then quantized or pruned for various optimization parameters. All models can be successfully quantized to both 5-bit weights and activations, or pruned with 70% sparsity without any substantial effect on accuracy. The larger baseline models are generally more robust and can be quantized more aggressively, however they are also more sensitive to low-bit activations. Moreover, for 8-bit integer quantization the networks were implemented on an ARM Cortex-A72 microprocessor, where inference latency was studied. These fixed-point models achieves up to 5.5x inference speedup on the ARM processor, compared to the 32-bit floating-point baselines. The larger models gain more speedup from quantization than the smaller ones.

While the results are not necessarily generalizable to different CNN architectures or datasets, the valuable insights obtained in this thesis can be used as starting points for further investigations in model optimization and possible effects on accuracy and embedded inference latency.

Handledare: Jesper Månsson
Ämnesgranskare: Ayca Özcelikkale
Examinator: Tomas Nyberg
ISSN: 1401-5757, UPTEC F 21021

Populärvetenskaplig Sammanfattning

Maskininlärning omfattar ett flertal matematiska metoder, vars utrymme i industrin ständigt ökar. Inom området datorseende, som avser bland annat bildigenkänning och objektdetektion, anses maskininlärningsmodellen CNN (convolutional neural network) vara en av de mest toppmoderna metoderna. Dessa modeller (nätverk) är dock oerhört beräkningstunga, och kan vara svåra att implementera på begränsad hårdvara; exempelvis drönare, robotar eller bilar. I detta examensarbete undersöks ett antal optimeringsmetoder, som gör nätverken mer lämpliga att applicera på sådan hårdvara, och hur dessa metoder påverkar nätverkens precision och beräkningstid. Specifikt undersöks kvantisering, ett sätt att minska nätverksparametrarnas upplösning, och pruning, som tar bort ett givet antal parametrar i nätverket.

Fyra CNN modeller implementerades och tränades på bildigenkänningsdata (med bilder på bilar, hundar, katter m.fl.), och därefter kvantiserades eller prunades. För 8-bitars heltalskvantisering implementerades också nätverken på en mikroprocessor. Alla modeller kunde kvantiseras ned till 5-bitar, eller prunas med 70% gleshet, utan att deras precision påverkades. Nätverken som implementerades på mikroprocessorn nådde upp till 5.5$\times$ minskning i beräkningstid.

Trots att resultaten inte nödvändigtvis kan generaliseras till andra typer av CNN modeller eller dataset, uppnåddes goda insikter som kan ligga till grund för fortsatt forskning inom hur kvantisering och pruning påverkar modellprecision och beräkningstid på begränsad hårdvara.

Dedicated to all the people who've made
these last five years some of the absolute best of my life.

Acknowledgements

I would like to sincerely thank Synective Labs AB for giving me this great and exciting opportunity. In particular, thanks to Jesper Månsson for his thorough guidance, and to both Gunnar Stjernberg and Niklas Ljung for their continuous support.

Many thanks to Ayca Özcelikkale for her supervision, helpful discussions and valuable feedback.

Lastly, I wish to express my gratitude to all my friends and my family, for your unconditional love and support.

Contents

List of Algorithms **xi**

List of Figures **xii**

List of Tables **xiv**

List of Abbreviations **xv**

1 Introduction **1**
1.1 Background . . . 1
1.2 Project Purpose . . . 1
1.3 Scope . . . 2
1.4 Related Work . . . 2
1.5 Thesis Structure . . . 3

2 Theory **4**
2.1 Deep Learning and Convolutional Neural Networks . . . 4
2.1.1 Machine Learning and Deep Learning . . . 4
2.1.2 Feedforward Neural Networks . . . 5
2.1.3 Convolutional Neural Networks . . . 6
2.1.3.1 Convolutional Layer . . . 6
2.1.3.2 Activation Functions . . . 7
2.1.3.3 Pooling Layers . . . 8
2.1.4 Training A Neural Network . . . 8
2.1.5 Batch Normalization . . . 9
2.1.6 Dropout . . . 10
2.2 CNN Architectures . . . 11
2.2.1 Popular Network Designs . . . 11
2.2.2 VGG . . . 11
2.2.3 MobileNetV2 . . . 11
2.2.3.1 Depthwise Separable Convolutions . . . 12
2.2.3.2 Bottleneck Residual Blocks . . . 12
2.3 CNN Optimization . . . 14
2.3.1 Quantization . . . 14
2.3.1.1 Quantization Schemes . . . 15
2.3.1.2 Post-Training Quantization . . . 16
2.3.1.3 Quantization Aware Training . . . 16
2.3.1.4 Quantized Inference . . . 17
2.3.2 Pruning . . . 18
2.3.3 Other Methods . . . 19

3 Implementation **20**
3.1 Baseline Networks . . . 20
3.1.1 Dataset . . . 20
3.1.2 Network Architectures . . . 20
3.1.3 Training . . . 21

3.2 Quantization . . . 22
3.2.1 Post-Training Quantization . . . 22
3.2.2 Quantization Aware Training . . . 23
3.3 Pruning . . . 25
3.4 Hardware Implementation . . . 26
3.4.1 ARM Cortex-A72 Details . . . 26
3.4.2 Embedded Inference . . . 26

4 Experiments and Results **28**
4.1 Post-Training Quantization . . . 28
4.2 Quantization Aware Training . . . 28
4.3 Pruning . . . 31
4.4 Embedded Inference . . . 33
4.4.1 C++ Implementation . . . 33
4.4.2 TensorFlow Lite Implementation . . . 34

5 Discussion **35**
5.1 Post-Training Quantization . . . 35
5.2 Quantization Aware Training . . . 35
5.3 Pruning . . . 36
5.4 Embedded Inference . . . 36
5.4.1 C++ Implementation . . . 36
5.4.2 TensorFlow Lite Implementation . . . 36
5.5 Generalizability of the Results . . . 37
5.6 Future Work . . . 37

6 Conclusion **38**

References **39**

Appendix A: Baseline Model Architectures . . . 42

Appendix B: Hyperparameter Tuning . . . 46

List of Algorithms

3.1 Floating-point QAT model to fixed-point conversion. 25
3.2 Forward propagation procedure of the C++ implementation. 27

List of Figures

1.1 Edge computing, where data processing is performed on the embedded device, versus cloud computing, where an external cloud server is used instead. 2
2.1 A feedforward neural network with 5 (L) layers. 5
2.2 A convolutional neural network with an input layer and a single hidden layer. . . . 6
2.3 A convolutional neural network with an input layer and a single hidden layer, with stride 2. 7
2.4 A convolutional neural network with an input layer and a single hidden layer, with zero-padding. 7
2.5 A simple CNN architecture with a $128 \times 128 \times 3$ input and 1×128 output. Created with NN-SVG (LeNail, 2019). 8
2.6 Gradient descent applied to a generic three-dimensional function as viewed from above (Alexandrov, 2004). 9
2.7 The VGG-16 architecture (Nash et al., 2018). 12
2.8 Standard convolution filters as well as the two building blocks for depthwise separable convolution layers; depthwise convolution and pointwise convolution filters. . 13
2.9 Residual and inverted residual blocks (Shanchen et al., 2019). 13
2.10 Bottleneck residual blocks as how they are implemented in the MobileNetV2 architecture. 14
2.11 A simple straight-through estimator which approximates the gradient for the real-valued weights by replacing the zero-gradient quantizer with the identity function. 17
3.1 Ten randomly chosen images from each class in the CIFAR-10 dataset (Krizhevsky, 2012). 20
3.2 Learning curves for the ConvNet baseline networks. 22
3.3 Learning curves for the BRNet baseline networks. 22
3.4 Relative accuracies for each quantization aware baseline model after being fine-tuned for a certain amount of epochs with 4-bit weights and activations. 23
3.5 Relative accuracies for each pruned baseline model after being fine-tuned with the *ConstantSparsity* schedule for a certain amount of epochs with 70% sparsity. . . . 25
4.1 Relative accuracies for each baseline quantized with QAT for varying weight bit-width. Activations are fixed to 8-bits. 29
4.2 Relative accuracies for each baseline quantized with QAT for varying activation bit-width. Weights are fixed to 8-bits. 29
4.3 Relative accuracies for the ConvNet baselines quantized with QAT for varying weight and activation bit-widths. Best viewed in colour. 30
4.4 Relative accuracies for the BRNet baselines quantized with QAT for varying weight and activation bit-widths. Best viewed in colour. 30
4.6 Relative accuracies for each pruned baseline for varying initial sparsity with the *PolynomialDecay* schedule. The final sparsity is set to 90%. 32
4.5 Relative accuracies for each pruned baseline for varying sparsity with the *ConstantSparsity* schedule. 32
4.7 Relative accuracies for each pruned baseline for varying sparsity function exponent with the *PolynomialDecay* schedule. The final sparsity is set to 90%. 33
A.1 ConvNet-S architecture. 42
A.2 ConvNet-L architecture. 43
A.3 BRNet-S architecture. 44
A.4 BRNet-L architecture. 45

B.1 Hyperparameter tuning for ConvNet-S. 46
B.2 Hyperparameter tuning for ConvNet-L. 46
B.3 Hyperparameter tuning for BRNet-S. 46
B.4 Hyperparameter tuning for BRNet-L. 47

List of Tables

2.1 A linear bottleneck transforming from M to M' channels, with stride s and expansion factor t (Sandler et al., 2019). 14
3.1 Baseline convolutional neural networks. 21
4.1 Relative accuracies [%] of the fixed-point post-training quantized baseline models. 28
4.2 TensorFlow's QAT parameters' effect on relative accuracy [%] for each baseline model. Each parameter (from Section 3.2.2) is separated by a horizontal line, and its best value for each baseline is highlighted in gray. 31
4.3 Inference latency for a single depthwise convolutional layer in C++, on an ARM Cortex-A72. 33
4.4 Computation time for the Hadamard product of two 100000×1 vectors in C++, on an ARM Cortex-A72. 34
4.5 Inference latency for each 8-bit post-training quantized baseline model, with the TFLite Inference Python API on an ARM Cortex-A72. 34

List of Abbreviations

API - Application Programming Interface
CNN - Convolutional Neural Network
CPU - Central Processing Unit
CUDA - Compute Unified Device Architecture
DL - Deep Learning
DRAM - Dynamic Random-Access Memory
FNN - Feedforward Neural Network
FPGA - Field-Programmable Gate-Array
GPU - Graphics Processing Unit
JSON - JavaScript Object Notation
LUT - Lookup Table
MAC - Multiply-Accumulate
ML - Machine Learning
PTQ - Post-Training Quantization
QAT - Quantization Aware Training
RAM - Random Access Memory
SGD - Stochastic Gradient Descent
SIMD - Single Instruction, Multiple Data
SoC - System on a Chip
SRAM - Static Random-Access Memory
STE - Straight-Through Gradient Estimator
TF - TensorFlow
TFLite - TensorFlow Lite

1. Introduction

1.1 Background

Machine learning (ML) is a subject of rapidly increasing interest and constantly expanding field of use, ranging from for example speech and image recognition to fraud detection and disease outbreak tracking (Meenakshi, 2020). The field confines a set of methods for adapting (*training*) mathematical functions to data, often by means of iteratively optimizing a loss function, which may be a computationally demanding problem. In fact, the increased availability of both data and high performing computers is largely what has enabled machine learning to grow during recent years (Goodfellow et al., 2016).

A specific family of ML models, *convolutional neural networks* (CNN), have shown significant progress in the computer vision area. Ever since the network AlexNet (Krizhevsky et al., 2012) won the ImageNet challenge: ILSVRC 2012, thereby beating multiple traditional image recognition methods, interest in CNNs has rapidly increased (Howard et al., 2017). The task of the competition was to both classify and localize hand-labeled objects of the ImageNet dataset, which contains 10M images and over 10 000 object classes (Russakovsky et al., 2015). AlexNet did not only win the challenge, but achieved an accuracy far beyond any of the competitors. Many people have shown great success in similar tasks using CNNs, and it has become the de facto model to use in computer vision problems (Kiranyaz et al., 2019).

As the complexity of real-world image recognition problems is usually quite high, modern CNNs need a vast number of trainable parameters (> 1M) to accurately fit to the data statistics. This means that not only training (using known input/output data to improve a model's accuracy by tweaking its parameters), but also *inference* (obtaining an unknown output from known input data, e.g. an object's name from an image's pixel values) demands a large amount of calculations. This issue complicates deployment of high-performing CNNs for applications where low-power processing units, or low-latency inference, is a must (Howard et al., 2017). Some examples are the automotive, robotics, drone and mobile industries. To combat this difficulty, inference is rarely performed on the local device itself but on a CPU/GPU cloud server. This approach introduces new problems, such as the requirement of constant network connectivity, which not only rapidly drains batteries but could also mean privacy and security concerns for the user. Hence, there is an incentive to perform inference on the device instead, often called *inference at the edge* (Moons et al., 2018) (see Figure 1.1).

1.2 Project Purpose

As common embedded devices normally have both limited memory availability and processing power, deploying a CNN to such a device means to shrink and speedup the network without sacrificing too much accuracy. There are multiple methods one may consider for such an optimization, of which *quantization* and *pruning* are examined in this thesis, with special focus on quantization. How the optimization methods affect model accuracy is of special interest, but inference latency (for the case when the CNN is implemented on embedded hardware) is also examined. The thesis provides guidelines for how, and when, to implement optimization methods for a specific type of CNN architecture, so that a user can improve an existing baseline network for a platform where e.g. inference latency, RAM or power consumption is a constraint.

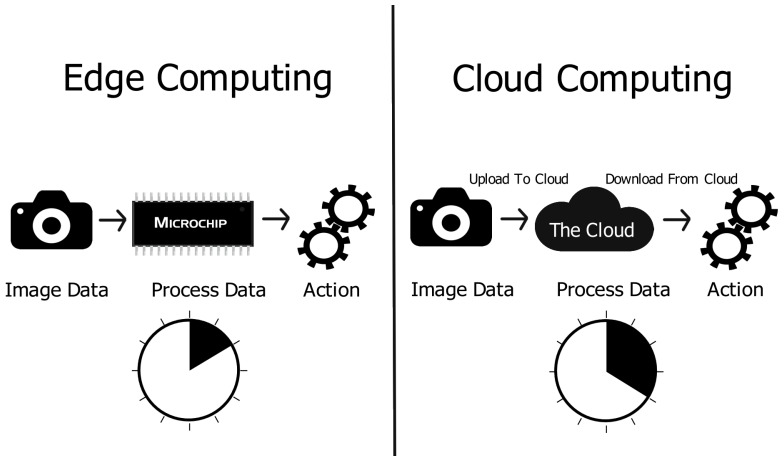


Figure 1.1. Edge computing, where data processing is performed on the embedded device, versus cloud computing, where an external cloud server is used instead.

1.3 Scope

Optimization of CNN models was implemented in *TensorFlow* (TF) (Abadi et al., 2016), which limits the extent of these results to what is achievable in the TF API. Further, a small set of CNN architectures are examined, and for a single dataset. Inference latency was studied on a single microprocessor, and not on any other hardware types. While these choices were all made with generalizability in mind, the presented results are thus not necessarily applicable to other CNN architectures, datasets or hardware types.

1.4 Related Work

The idea of how to optimize a neural network can be approached in multiple ways. One may shrink the network's size by defining an alternative architecture, thereby reducing the amount of required calculations. One example is the small MobileNet (Howard et al., 2017), which obtained comparable accuracy to state of the art models at the time, with much fewer parameters (30× fewer than the popular VGG16 network). Two other approaches are either truncating the network *weights* (parameters) to low-bit representations (*quantization*) or removing the weights of lowest usefulness completely (*pruning*).

Many early quantization methods focused on using either binary (Courbariaux et al., 2016) or ternary (Zhu et al., 2017) weights. Some focused on clustering the weights in fixed groups, where all weights in a group are represented by the same value (Han et al., 2016). Using binary/ternary weights may however not yield speedup on non-custom hardware (e.g. CPUs) unless the *activations* (further described in Chapter 2) are also quantized similarly, which may lead to extreme accuracy degradation. Further, weight clustering (or *lookup tables*, LUT) may be inefficient on SIMD hardware, and hence quantization schemes with arithmetic mapping to arbitrary bit-width weights was proposed (Jacob et al., 2017). This mapping is usually linear, but schemes utilizing non-linear mappings have also been examined (Nayak et al., 2019). The quantization schemes can be applied either during (*quantization aware training*, QAT) or after (*post-training quantization*, PTQ) training, both of which have been compared, with QAT yielding a lower accuracy degradation than PTQ (Krishnamoorthi, 2018).

Pruning instead performs direct changes to a CNN's architecture by removing parts of the network. The most common method is to remove weights by iteratively scoring their usefulness in the network and removing the ones with the lowest scores (Han et al., 2015). This method compresses the model size, but does not necessarily improve latency, which will be further discussed

in Chapter 2. To achieve inference speedup one may instead remove other structures in the network, such as deleting *channels* or *kernels* (Anwar et al., 2015) and (Li et al., 2017), or trimming the network into specific shapes (Krishnan et al., 2019).

Pruning and quantization have been explored together (Paupamah et al., 2020), however not as thoroughly as needed, in terms of how both methods' can be individually altered. Some papers with focus on quantization perform in-depth accuracy trade-off comparisons but only report power consumption and memory usage but not inference time (Moons et al., 2017), while others only report inference time (Jacob et al., 2017). Sometimes when network speedup is reported, results have been gathered from running on a GPU and not on relevant embedded hardware (Narang et al., 2017).

In this thesis both quantization and pruning are explored, in a way that aims to obtain results that can be generalized to other network architectures and problem formulations.

1.5 Thesis Structure

The structure of this document is as follows: Chapter 1 gives an overview of the subject and project. Chapter 2 goes in-depth into the relevant theory. This includes some elemental machine learning theory along with a formulation of standard neural networks as well as the convolutional type. Some important CNN architectures are then presented, and lastly both quantization and pruning is thoroughly explained. The practical approach of the project is then described in Chapter 3, before a defined explanation of the performed experiments and gathered results in Chapter 4. The results are then discussed in Chapter 5 and the project as a whole is concluded in Chapter 6.

2. Theory

2.1 Deep Learning and Convolutional Neural Networks

This section presents some machine learning basics, as well as the fundamentals of feedforward and convolutional neural networks and how to train them. The theory covered in this section is based on MIT Press' Deep Learning book (Goodfellow et al., 2016) and the soon to be published Cambridge University Press' Supervised Machine Learning book (Lindholm et al., 2021).

2.1.1 Machine Learning and Deep Learning

A subset of machine learning, *deep learning* (DL) methods have become prominent in countless fields. The general idea is to utilize multiple levels of composition of more primitive methods, meaning the DL method is a complex representation consisting of multiple simpler representations. While the idea can be generalized, the fundamental DL model is the *feedforward neural network* (FNN), which has given rise to multiple other eminent methods, such as the convolutional neural network. However, before one can understand neural networks one must first also comprehend the basics of machine learning.

Machine learning is the field of building computer algorithms that learn and improve their parameters from data. The field can be split in different categories, depending on the nature of the training procedure and data. This thesis focuses on *supervised learning*, where the training data contains input values and their corresponding labeled outputs (compared to *unsupervised learning* where the algorithm learns only from input data). Hence, as opposed to classical mathematical modelling where a model representation is gathered by e.g. physical or medical knowledge of a system, machine learning models learn an input-output relation from data examples.

One of the most simple machine learning models is *linear regression*, where a true output $y \in \mathbb{R}$ is modelled as

$$y = \mathbf{w}^T \mathbf{x} + \varepsilon, \tag{2.1}$$

and estimated as

$$\hat{y} = \mathbf{w}^T \mathbf{x}, \tag{2.2}$$

where $\mathbf{x} \in \mathbb{R}^m$ is the input data, ε is unknown noise, and the weights (parameters) $\mathbf{w} \in \mathbb{R}^m$ are learned from training data

$$\mathbf{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}, \quad \mathbf{X} = \begin{bmatrix} \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_n \end{bmatrix} = \begin{bmatrix} x_{11} & \dots & x_{1m} \\ \vdots & \ddots & \vdots \\ x_{n1} & \dots & x_{nm} \end{bmatrix}. \tag{2.3}$$

Here m is the order of the model and n is the number of data examples in the dataset. Normally, training a linear regression model corresponds to estimating the true weights by solving the closed-form *normal equation* as

$$\hat{W} = (\mathbf{X}^T \mathbf{X})^\dagger \mathbf{X}^T \mathbf{y}, \tag{2.4}$$

where † is the *Moore-Penrose inverse* operator. While this procedure is quite trivial, training more advanced models can sometimes be a difficult task, which is described later in this chapter.

While linear regression is quite a simple approach on it's own, a composition of multiple models can produce a complex representation of an input-output relation called a neural network, which will now be further described.

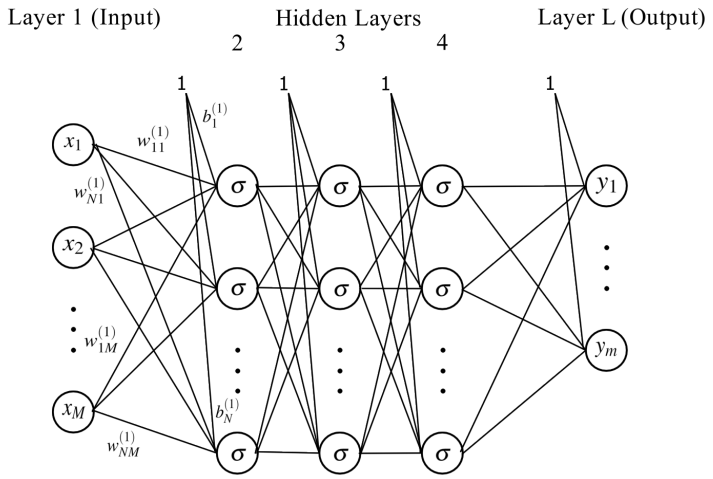


Figure 2.1. A feedforward neural network with 5 (L) layers.

2.1.2 Feedforward Neural Networks

A feedforward neural network is the most fundamental deep learning model. It maps an input-output relation with a nonlinear function $\mathbf{y} = f(\mathbf{x};\theta)$, where $\mathbf{y}$ is the output, $\mathbf{x}$ is the input and θ are the parameters (including both *weights* and *biases*). As the name implies, an FNN does not have any feedback-blocks but only feed information from the input forward in the network. The architecture of feedforward networks is important to understand as it is easily generalized to other network classes, such as the convolutional neural network which is the main focus of this thesis.

Feedforward neural networks are structured in *layers* ℓ, where each layer have one or multiple *nodes*, or *units*, u. Each node in a layer is connected to every node in the next layer, with a unique weight w associated to each connection. For a single node in the next layer, all connected previous nodes' outputs are scaled by their respective weights and summed. Normally, a translation term called a *bias* b is also added to the the node (each node has a unique bias). The input and output layers are referred to as such, and all intermediate layers are referred to as *hidden layers*.

The architecture described above is presented in Figure 2.1. This network has L layers, where a specific layer ℓ with N nodes is parameterized by its weights $\mathbf{W}^{(\ell)}$ and biases $\mathbf{b}^{(\ell)}$ according to

$$\mathbf{W}^{(\ell)} = \begin{bmatrix} w_{11}^{(\ell)} & \dots & w_{1M}^{(\ell)} \\ \vdots & \ddots & \vdots \\ w_{N1}^{(\ell)} & \dots & w_{NM}^{(\ell)} \end{bmatrix}, \quad \mathbf{b}^{(\ell)} = \begin{bmatrix} b_1^{(\ell)} \\ \vdots \\ b_N^{(\ell)} \end{bmatrix}, \tag{2.5}$$

and its output is

$$\mathbf{h}^{(\ell)} = \sigma(\mathbf{W}^{(\ell)}\mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)}), \tag{2.6}$$

where $h^{(\ell-1)}$ is the previous layer's output and $\sigma : \mathbb{R} \to \mathbb{R}$ is an *activation function*, acting element-wise on each node. This concept is further described later in this chapter, along with a few examples of commonly used activation functions. It should be noted that while a layer's output is a function of the previous layer's output; $\mathbf{h}^{(\ell)} = f(h^{(\ell-1)};\theta^{(\ell)})$, two special cases are the first hidden layer; $\mathbf{h}^{(1)} = f(\mathbf{x};\theta^{(1)})$, and output layer; $\mathbf{y} = f(h^{(L)};\theta^{(L)})$.

In conclusion, a feedforward neural network is a nonlinear function $\mathbf{y} = f(\mathbf{x};\dots)$ with the fol-

lowing form:

$$\mathbf{h}^{(1)} = \sigma(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) \tag{2.7a}$$

$$\vdots$$

$$\mathbf{h}^{(L-1)} = \sigma(\mathbf{W}^{(L-1)}\mathbf{h}^{(L-2)} + \mathbf{b}^{(L-1)}) \tag{2.7b}$$

$$\mathbf{y} = \sigma(\mathbf{W}^{(L)}\mathbf{h}^{(L-1)} + \mathbf{b}^{(L)}). \tag{2.7c}$$

For the output layer special activation functions are often used, which will be covered later.

The discovery of the hidden layers' importance is the cornerstone of deep learning. In fact, the word *deep* intuitively refers to the use of multiple hidden layers, as opposed to only using for instance a single hidden layer, which is referred to as a *shallow* network. Now that the foundation of deep learning and the neural network architecture has been covered, we'll advance to CNNs as well as other variants of network designs.

2.1.3 Convolutional Neural Networks

2.1.3.1 Convolutional Layer

Convolutional neural networks are a class of neural networks made specifically for learning patterns in data that is best represented in a grid-like form, such as images. While it is possible to vectorize such data and use a standard FNN, this approach means that a lot of information in the data is lost and never utilized by the model.

Instead, the approach of a CNN is having both the input and hidden layers multidimensional, which would mean a huge amount of weights are needed. However, while an FNN would let every unit in a layer interact with every unit in the next, CNNs utilize *sparse interactions*, meaning that a unit is connected to only some of the following ones. This also makes sense, since it is more natural to assume that a pixel in an image only correlates to some of the surrounding ones (as they make up a shape or an object) and not all the pixels in the image. Further, CNNs utilize *weight sharing*, meaning that the same weights are used for all unit-pairs in a layer. The weights for a convolutional layer are hence structured in a 2-dimensional matrix, called a *filter*, that moves over (*convolve*) an input matrix to produce a matrix output. It should be noted that *input* and *output* matrices here refer to inputs and outputs of an arbitrary convolutional layer, and not the input and outputs of the network. The weight sharing feature also lets the network achieve *equivariance to translation*, which means that if an input is changed by translation (shifting), the output will then be changed in the same manner. Since image data gathered from either a non-stationary camera or a camera targetting a non-stationary object is likely to be affected by this kind of noise, this is a much needed feature for the network. Sparse interactions and weight sharing can be viewed in Figure 2.2.

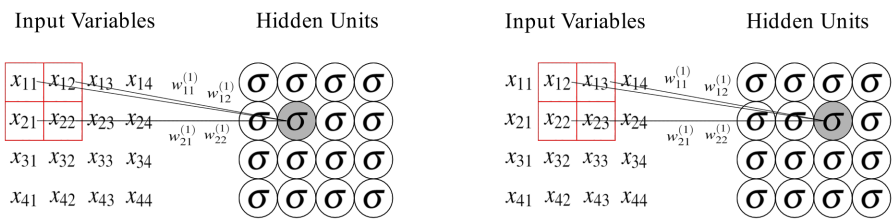


Figure 2.2. A convolutional neural network with an input layer and a single hidden layer.

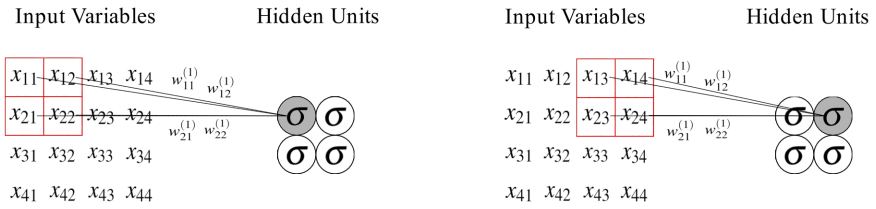


Figure 2.3. A convolutional neural network with an input layer and a single hidden layer, with stride 2.

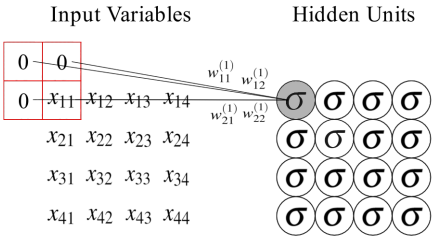


Figure 2.4. A convolutional neural network with an input layer and a single hidden layer, with zero-padding.

For many CNN architectures the desired output is not a matrix, but for instance a vector of scalar class probabilities. The information flowing through the network must therefore successively decrease in dimension, normally enabled by either using *strides* or *pooling layers*. Stride means that when the filter is applied to the input layer, some elements are purposely skipped. More specifically, e.g. stride 2 means that the filter moves in steps of 2, meaning that the output will be halved in size. See Figure 2.3 for an example of a convolutional layer with stride 2.

Looking back at Figure 2.2, it may be noted that the filter in top-left position of the input does not produce a top-left output element. If each filter position would correspond to the same output position, the output dimension would be reduced due to the filter's non-unit size. To maintain the same dimension throughout the convolutional layer, *zero-padding* is normally used. The technique simply adds zeros to the borders of the input, where elements are missing, so that the output has the same number of rows and columns as the input. See an example in Figure 2.4.

The most common use for CNNs is with image data. Normally, due to most image colour models containing multiple colour components (for example red, green and blue), the data is hence not structured in a 2-dimensional matrix but a 3-dimensional *tensor* (rows $\times$ columns $\times$ channels), where the last dimension corresponds to colours. This adds another axis to every layer's input, called *channels*, each corresponding to a certain colour component. To maintain, and further encode, this additional dimension of data, the filter is also converted into a 3-dimensional tensor. Normally, multiple tensor-valued filters are used, each producing a separate output channel for the next layer. Each 2-dimensional set of weights along the third axis are referred to as *kernels*. See Figure 2.5 for an illustration of a tensor-valued CNN's full architecture.

2.1.3.2 Activation Functions

Activation functions σ are normally scalar functions that act element-wise on each layer output. Its purpose is to enable the network to capture nonlinear behaviour in the data, and is therefore typically a nonlinear function. For hidden layers, the most common activation function is the *rectified linear unit* (ReLU), $\sigma^{\text{ReLU}} : \mathbb{R} \to \mathbb{R}$, or variations of it.

For the output layer of a neural network, special activation functions are normally used. For

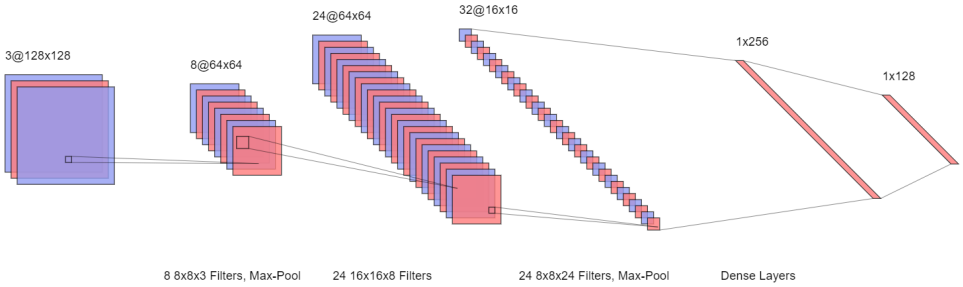


Figure 2.5. A simple CNN architecture with a $128 \times 128 \times 3$ input and 1×128 output. Created with NN-SVG (LeNail, 2019).

regression problems there is typically no activation function at all, $\sigma(z) = z$, and for classification problems the *softmax* function, $\sigma^{\text{softmax}} : \mathbb{R}^{\mathbb{K}} \to \mathbb{R}^{\mathbb{K}}$, is used for multi-class problems and the *sigmoid* function, $\sigma^{\text{sigmoid}} : \mathbb{R} \to \mathbb{R}$, for binary class problems. All of the mentioned activation functions can be viewed in equation 2.8.

$$\sigma^{\text{ReLU}}(z) = max(0, z) \tag{2.8a}$$

$$\sigma_i^{\text{softmax}}(\mathbf{z}) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}, \quad i = 1, \ldots, K, \quad \mathbf{z} = [z_1, \ldots, z_K] \tag{2.8b}$$

$$\sigma^{\text{sigmoid}}(z) = \frac{e^{z}}{e^{z} + 1} \tag{2.8c}$$

While sometimes omitted, most convolutional layers are followed by a scalar activation function.

2.1.3.3 Pooling Layers

As with strides, pooling layers are a way of reducing the dimension of an input layer. However, instead of skipping pixels of the input tensor, pooling utilizes a function gathering summary statistics of nearby pixels, normally the adjacent ones. Common examples are *max* or *average pooling*, which outputs the maximum or average value of a rectangular neighborhood of pixels respectively.

Pooling enables the model to become locally *invariant* to pixel translations, meaning that if the input is slightly changed (e.g. a specific object in the image moves briefly), the pooled outputs remain the same. This is especially useful in classification, where one does not care about the specific location of an object but instead only about its presence.

Pooling is normally used after one or multiple convolutional layers, each with a subsequent activation function.

2.1.4 Training A Neural Network

Most machine learning models are trained by minimizing a problem-specific cost function, parameterized by the model's weights and biases. While such an optimization problem might have a closed-form solution (like linear regression), the nonlinearity of neural networks causes most cost functions of interest to become nonconvex in terms of the model parameters. Hence, the cost function cannot be minimized globally, and numerical optimization is needed. While alternative approaches are possible (Taylor et al., 2016), the optimization method used to train a neural network is almost exclusively some form of *gradient descent* (Figure 2.6).

Gradient descent is a method for finding an optimal set of a model's parameters θ that minimize

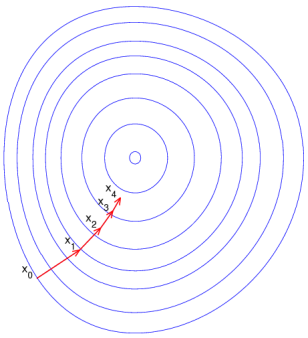


Figure 2.6. Gradient descent applied to a generic three-dimensional function as viewed from above (Alexandrov, 2004).

a cost function $J(\theta)$ by

$$\hat{\theta} = \arg\min_{\theta} J(\theta), \quad J(\theta) = \frac{1}{n}\sum_{i=1}^{n} L(\mathbf{x}_i, \mathbf{y}_i, \theta), \tag{2.9}$$

where n is the amount of data points and $L(\mathbf{x}_i, \mathbf{y}_i, \theta)$ is the problem specific *loss function*. In gradient descent, the gradient of the cost function $\nabla_\theta J(\theta)$ is calculated, and the parameters are updated so that the value of the cost function decreases. This is done iteratively, so that the parameters each time step are updated according to

$$\theta^{(t+1)} = \theta^{(t)} - \gamma \nabla_\theta J(\theta^{(t)}), \tag{2.10}$$

where γ is the *learning rate*.

To obtain the gradient of the cost function, the *back-propagation* algorithm is used. Compared with the previously covered case where information flows forward in the network and computes an output from an input (*forward propagation*), back-propagation is a way to compute the gradient from an output. For a thorough explanation of this algorithm, the reader is referred to Chapter 6 in Goodfellow et al. (2016).

For large datasets (n large) it is common to not use all data for gradient computation, but a subsample of data called a *mini-batch*. This is the approach of *stochastic gradient descent* (SGD), where a randomized set of data points is linearly divided in multiple mini-batches, and each new iteration of the back-propagation algorithm uses the subsequent mini-batch to compute the gradient. A complete pass through all training data is called an *epoch*. The most commonly used learning algorithms are extensions of the stochastic gradient descent method. Some examples are *AdaGrad*, *RMSProp* or *ADAM* (see Section 8.5 in Goodfellow et al. (2016)).

To ensure convergence during gradient descent, both the initialization of the model parameters and a good choice of learning rate is needed. While there are no guarantees, a common guideline is to choose small and random initial parameters θ_0 and learning rate by some *validation technique*. Validation theory will not be covered in this thesis, but is explained thoroughly in Berrar (2018).

2.1.5 Batch Normalization

When computing the optimal parameter update with gradient descent as in Equation 2.10, for each layer that are passed in back-propagation the other layers are assumed to be constant. Since all parameters are updated simultaneously after an iteration, this assumption does obviously not hold. A suggested approach that addresses this problem is *batch normalization* (Ioffe and Szegedy, 2015).

The idea is to, for each mini-batch B in back-propagation, normalize the layers' activations (outputs) $\mathbf{h}^{(\ell)}$ by re-centering and re-scaling according to

$$\hat{\mathbf{h}}_i^{(\ell)(k)} = \frac{\mathbf{h}_i^{(\ell)(k)} - \mu_B^{(\ell)(k)}}{\sigma_B^{(\ell)(k)}}, \tag{2.11}$$

for each dimension k in layer ℓ and data point $i \in [1, \ldots, n]$ separately. For each layer, the statistics of batch B is obtained from

$$\mu_B = \frac{1}{n}\sum_{i=1}^{n} \mathbf{h}_i, \quad \sigma_B^2 = \frac{1}{n}\sum_{i=1}^{n} (\mathbf{h}_i - \mu_B)^2. \tag{2.12}$$

This normalization may however reduce the flexibility of the network, which can be prevented by introducing a new mean and variance to the normalized activation, which enables the network to represent the same family of functions as before. For a general activation $\mathbf{h}$ this is achieved by

$$\hat{\mathbf{h}} \to \alpha\hat{\mathbf{h}} + \beta, \tag{2.13}$$

for some learned parameters α and β.

2.1.6 Dropout

As neural networks tend to be quite large models, in terms of parameter amount, they sometimes might have the risk of *overfitting* to the data. This means that the model's adaptable ability (*flexibility*) is greater than the data's complexity, and hence that the model captures non-desired patterns in the data, such as noise. There exist multiple techniques for reducing a neural networks's flexibility (*regularization techniques*), and one of them is *dropout*.

When training a network with dropout, some input units for each layer are removed during each back-propagation iteration. In other words, for each mini-batch a sub-network with some input units removed is used to approximate the gradient, instead of the full network. While still quite different, the approach is similar to a method called *bagging*, where an *ensemble* (set) of models are trained independently, and their output is averaged. While bagging is commonly used for less complex models, the sheer size of neural networks makes the method inapplicable, which motivated the alternative dropout approach.

2.2 CNN Architectures

This section presents an overview of some common CNN architectures, as well as an in-depth description of the two relevant architectures *VGG* and *MobileNetV2*.

2.2.1 Popular Network Designs

Following the famous AlexNet (Krizhevsky et al., 2012) and subsequent interest in CNNs, significant findings and innovative improvements have successively been made within the subject. In Zeiler and Fergus (2013) a method of visualizing CNNs layer-wise was found, and the results later motivated feature extraction at low spatial resolutions (Khan et al., 2020). An example of such a network design is the VGG architecture (Simonyan and Zisserman, 2015), which uses only 3×3 kernels in each convolutional layer's filter. A network with similar accuracy at the time was the Inception architecture (Szegedy et al., 2014), concatenating outputs of multiple filters each with different kernel sizes. Not long thereafter, a successful alternative network structure called *residual blocks* (He et al., 2015) based on connections that skip layers and connect to later ones; *shortcut* or *skip connections*, was invented. The residual network design is motivated by its ability to improve gradient propagation during training.

While the mentioned network designs are all successful in terms of accuracy, they are also computationally inefficient. The task of optimizing neural network performance can be approached in multiple ways. Perhaps the most apparent solution is to simply define a smaller network architecture (fewer parameters), without penalizing the model's accuracy. The topic has been thoroughly explored, with early papers mainly focusing at approximating filters with lower dimensions (Jin et al., 2015), (Jaderberg et al., 2014). The highly successful DenseNet (Huang et al., 2016) utilizes skip connections (similarly to residual network). The widespread MobileNet (Howard et al., 2017) uses low-cost *depthwise separable convolution* operators, and at the time achieved state-of-the-art accuracy with fewer parameters and remarkably fewer necessary operations, on popular datasets. The recent MobileNetV2 (Sandler et al., 2019) uses *bottleneck residual* blocks to further improve the initial version's architecture design.

2.2.2 VGG

While new innovative network designs constantly emerge, most are still conceived from the simple principles of the VGG architecture (Simonyan and Zisserman, 2015), according to Khan et al. (2020). The idea of the VGG design is to stack multiple convolutional layers, each with filters consisting of 3×3 kernels, and reduce the feature map dimension with max pooling layers. After each pooling layer the channel dimension is increased by a factor of 2, so that each feature map is smaller, but wider. At the end of the network multiple fully connected layers are inserted, where the last one has a softmax activation. All other convolutional and fully connected layers use ReLU activations. The general network design can be viewed in Figure 2.7, and the full architecture is presented in Simonyan and Zisserman (2015).

2.2.3 MobileNetV2

In a comparison of multiple popular CNN architectures benchmarked on the ImageNet dataset (Bianco et al., 2018), MobileNetV2 is the fastest model (in images per second) among those with an accuracy over 70%, on embedded hardware. The MobileNetV2 architecture stacks the gradient- and computationally friendly bottleneck residual blocks, with successively increasing channel dimension for each feature map reduction. This dimension reduction is performed by using blocks with stride 2, as opposed to the remainders which all use stride 1. While the bottleneck residual block is the main component, the network is initiated and ended with convolutional blocks, with average pooling used at the end. The bottleneck residual block is of importance and

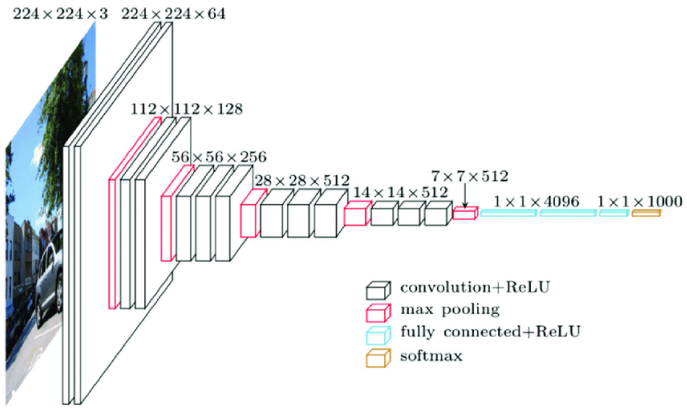


Figure 2.7. The VGG-16 architecture (Nash et al., 2018).

will be further explained throughout this section, and the full MobileNetV2 architecture can be found in Sandler et al. (2019).

2.2.3.1 Depthwise Separable Convolutions

For a convolutional layer with stride 1, assume that N filter tensors each of size $D_K \times D_K \times M$ convolve an input tensor $D_F \times D_F \times M$ to produce an output tensor $D_F \times D_F \times N$. This operator consists of both filtering the input and combining each channel to a new set of outputs. With depthwise separable convolutions, the input tensor is initially filtered with a *depthwise convolution*, and the outputs are combined with *pointwise convolution*. These two operators put together makes for an enormously more efficient convolutional layer, in terms of computational cost.

As described in the initial MobileNet paper (Howard et al., 2017), depthwise convolutions use M 2-dimensional filters that initially convolve the input tensor. The pointwise convolution operator then convolve these outputs with N filters each consisting of M unit size kernels. This is opposed to a standard convolutional operator with N filters each with M sized kernels. The procedure can be viewed in Figure 2.8. MobileNet specifically uses depthwise convolution filters of size 3×3.

While standard convolution layers have a computational cost of

$$D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F \tag{2.14}$$

multiply-accumulate (MAC) operations, depthwise separable convolutional layers have a cost of

$$D_K \cdot D_K \cdot M \cdot D_F \cdot D_F + M \cdot N \cdot D_F \cdot D_F \tag{2.15}$$

operations. This corresponds to a reduction in computation of

$$\frac{1}{N} + \frac{1}{D_K^2}. \tag{2.16}$$

2.2.3.2 Bottleneck Residual Blocks

The main building block for the improved MobileNetV2 architecture (Sandler et al., 2019) is the bottleneck residual block. The block consists of both a *linear bottleneck* and *inverted residual*.

Linear bottlenecks are motivated by experimental findings that indicate a need for linear layers, as well as the assumption that layer activations may be encoded in low-dimensional subscapes. In practice they are used as a pointwise convolution with linear activation function, that lowers a tensor's channel dimension.

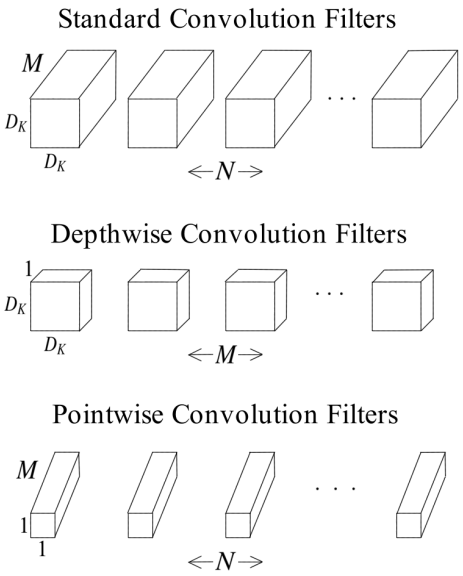


Figure 2.8. Standard convolution filters as well as the two building blocks for depthwise separable convolution layers; depthwise convolution and pointwise convolution filters.

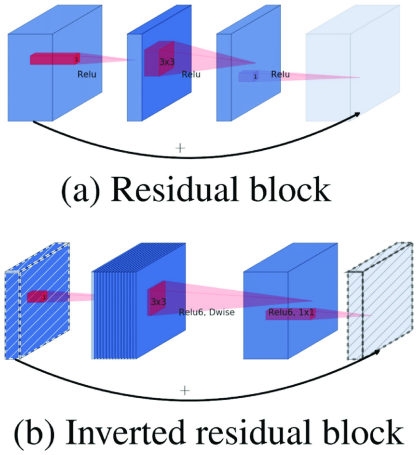


Figure 2.9. Residual and inverted residual blocks (Shanchen et al., 2019).

Inverted residuals are based on *residual blocks* (He et al., 2015) that, similarly to DenseNet (Huang et al., 2016), utilize one or multiple *shortcut* or *skip* connections that skip the next layer and instead connect to a succeeding one. The residual network design is motivated by their ability to improve gradient propagation during training. While a residual block skips layers of lower dimension, the inverted residual block skips layers of higher dimension, which is considerably more memory efficient (Sandler et al., 2019). The difference between the two blocks can be viewed in Figure 2.9.

The bottleneck residual block can be viewed in Table 2.1. The last pointwise convolution has a linear output, and the other layers use a custom activation function; ReLU6. This function (Equation 2.17) is mostly the same as a normal ReLU, but clips all values over 6. For the MobileNetV2 architecture (Sandler et al., 2019), the inverted residual is implemented as in Figure 2.10. Only blocks with stride 1 use residual connections.

$$\sigma^{\text{ReLU6}}(z) = min(max(0,z),6) \tag{2.17}$$

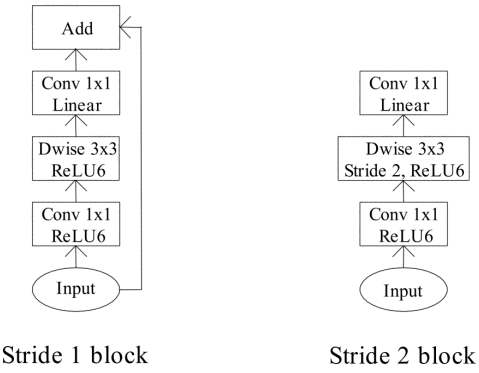


Figure 2.10. Bottleneck residual blocks as how they are implemented in the MobileNetV2 architecture.

Table 2.1. *A linear bottleneck transforming from M to M' channels, with stride s and expansion factor t (Sandler et al., 2019).*

Input	Operator	Output
$D_F \times D_F \times M$	Pointwise, ReLU6	$D_F \times D_F \times tM$
$D_F \times D_F \times tM$	3x3 Depthwise, stride s, ReLU6	$\frac{D_F}{s} \times \frac{D_F}{s} \times tM$
$\frac{D_F}{s} \times \frac{D_F}{s} \times tM$	Pointwise, Linear	$\frac{D_F}{s} \times \frac{D_F}{s} \times M'$

2.3 CNN Optimization

This section describes the CNN optimization method *quantization* in detail; how it can be applied to a model, and how to perform inference with a quantized model. The method *pruning* is also briefly explained.

2.3.1 Quantization

As an alternative to improving model performance with architectural changes, quantization is the means of optimization by reducing precision of a model's parameters and activations. More specifically, these quantities are normally stored in 32-bit floating-point numbers, and a quantized network instead stores quantized variables in lower-bit data types (normally integers). It should be noted that *activations* do not refer to the model's activation functions, but instead layer outputs.

According to Krishnamoorthi (2018), quantization is not only an easily adopted approach for existing models (no need to define a new architecture), but can improve a model's efficiency in terms of memory, power consumption and inference time. Model size, or memory usage, is quite evident; as low-bit data requires less data to be stored. Power consumption is reduced since memory access is the dominating factor in energy usage for both on-chip SRAM or off-chip DRAM (Han et al., 2016). As for inference time, the subject is more complicated. For extreme quantization, where both weights and activations are binary variables, neural networks can be implemented on custom logical hardware circuits (Roth et al., 2020). As for the more common case, where the network is to be implemented on existing hardware (CPUs or GPUs), there is no definite answer regarding speedup. While integer operations generally require fewer clock cycles to compute than floating-point operations (Le Maire et al., 2016), a 4-bit network is not necessarily faster than a 8-bit network, for instance. This is since e.g. CPUs and GPUs lack data types for uncommon bit widths, generally < 8 bits (Lam et al., 2020). This is also true for all bit-widths between two data type; for instance a 9-bit variable would be stored in a 16-bit integer data type. Achieving speedup with non-conventional bit-widths is still an active area of research, and has been studied in Choi et al. (2018) and Lam et al. (2020)

2.3.1.1 Quantization Schemes

As for how quantization is implemented, there are multiple options available. An early approach was to use lookup tables, where the weights are stored as indices, each one corresponding to a value in a finite array (Han et al., 2016). This technique, called *weight clustering*, compresses the model as the weights can easily be shared in storage; yielding a smaller memory footprint. According to Jacob et al. (2017) LUT approches allegedly perform poorly on hardware that utilize SIMD operations; a common way of data parallelism by executing an operation on multiple data items during a single instruction (Hughes, 2015). This motivated a purely arithmetic approach, where the *quantization scheme* is instead defined as (Jacob et al., 2017)

$$r = \Delta(q - z). \tag{2.18}$$

Here r is the real value, normally stored in 32-bit floating-point, q is the quantized value of arbitrary bit-width and Δ and z are the quantization parameters *scale* and *zero-point*, respectively. It should be noted that Δ is a real number, and that z is of the same data type as the quantized value (normally an integer). To quantize a real value, the approach is hence to divide it with the scale, and add the zero-point. The result must then be rounded to the nearest integer, and then clipped to fit inside the data type's range. This corresponds to

$$q = clamp(\lfloor \frac{r}{\Delta} \rceil + z), \tag{2.19}$$

$$clamp(x) = min(max(x,\, N_{min}),\, N_{max}), \tag{2.20}$$

where $\lfloor . \rceil$ means rounding to the nearest integer, and N_{min} and N_{max} are the lower and upper bounds of the data type's range; $[N_{min},\, N_{max}]$. For an N-bit integer this range is $[-2^{N-1},\, 2^{N-1} - 1]$ for the signed type, and $[0, 2^N - 1]$ for the unsigned type. Equation 2.19 is referred to as *uniform asymmetric*, or *uniform affine*, *quantization* (Krishnamoorthi, 2018).

Another common scheme, which restricts the zero-point to 0 is *uniform symmetric quantization* (Krishnamoorthi, 2018), according to

$$q = clamp(\lfloor \frac{r}{\Delta} \rceil). \tag{2.21}$$

Further, for symmetric quantization with signed integers the lower range of the N-bit integer can be restricted to $-2^{N-1} - 1$, so that the lower and upper limits are the same. This may yield even faster SIMD implementations (Krishnamoorthi, 2018). This restriction is commonly known as using *narrow range*.

The scale and zero-point parameters may be defined in different ways. Some designate them as learnable parameters to the network, and hence train the quantizer with data (Zhang et al., 2018), (Jain et al., 2019). Another, possibly less cumbersome, method is to compute the parameters with the minimum and maximum values, r_{min} and r_{max}, of the real-valued tensor or channel elements (known as either *per-tensor* or *per-channel* quantization). For example when quantizing a model's weights with per-tensor quantization, this means that for each layer (weight tensor) in the network, the quantizer scales them based on each tensor's minimum and maximum weight values separately. An example, found in the documentation of Intel's Neural Network Distiller

(Zmora et al., 2019), is to compute the parameters according to

$$\Delta^{\text{asym}} = \frac{r_{max} - r_{min}}{2^N - 1}, \tag{2.22a}$$

$$\Delta^{\text{sym, full}} = \frac{max(|r_{max}|, |r_{min}|)}{(2^N - 1)/2}, \tag{2.22b}$$

$$\Delta^{\text{sym, narrow}} = \frac{max(|r_{max}|, |r_{min}|)}{2^{N-1} - 1}, \tag{2.22c}$$

$$z^{\text{asym}} = \lfloor \frac{r_{min}}{\Delta^{asym}} \rceil - 2^{N-1}, \tag{2.22d}$$

$$z^{\text{sym}} = 0. \tag{2.22e}$$

The asymmetric equations are for the signed integer case. If unsigned integers are used, 2^{N-1} must be added to the zero-point.

Finally, as mentioned in the beginning of the section, not only the weights but also the activations can be quantized. While they can be quantized in the manner described in this section, activations should specifically use per-tensor quantization for maximum performance (Krishnamoorthi, 2018). Further, as the activations' ranges cannot simply be extracted from the model, some data is normally needed for computing the scale and zero-point parameters (Wu et al., 2020).

2.3.1.2 Post-Training Quantization

Being a post-processing method applicable on pre-trained models, post-training quantization is the simplest quantization technique. It is especially useful for cases where available data is limited (due to e.g. privacy concerns) (Krishnamoorthi, 2018). While some data is normally used, mainly to calibrate the activations' ranges, recent studies have successfully performed post-training quantization without any calibration data, for example through synthetic data generation (Cai et al., 2020) or by using statistical knowledge of the network model (Nagel et al., 2019).

As no further training is needed, post-training quantization is computationally cheap to perform, but may result in significant accuracy degradation for low-bit quantization (Banner et al., 2019).

2.3.1.3 Quantization Aware Training

As opposed to PTQ, which is simply applied to a pre-trained model, quantization aware training inserts quantization operations into a network before it is thereafter trained with these induced additions. This lets the network adapt to its quantized parameters and activations, which may lead to lower accuracy degradations (Wu et al., 2020). Normally, the network is retained in floating-point but with inserted *fake quantization operations*, described in Wu et al. (2020) as

$$\hat{r} = dequantize(quantize(r, \ldots)). \tag{2.23}$$

As intuitively described in a blog post by the TensorFlow Model Optimization team (see Chapter 3 for more information about TensorFlow); "This introduces the quantization error as noise during the training and as part of the overall loss, which the optimization algorithm tries to minimize. Hence, the model learns parameters that are more robust to quantization" (TensorFlow Model Optimization team, 2020). Hence, during back-propagation, the effects of the fake quantization operators are approximated and included in the gradient calculation (Jacob et al., 2017), which is performed by the use of the *straight-through gradient estimator* (STE) (Bengio et al., 2013).

The STE approximates the gradient by replacing the normally zero-gradient piecewise constant quantizer with a new differentiable function that is similar to the quantizer (Roth et al., 2020). An example can be viewed in Figure 2.11, where the quantizer is replaced by the non-zero-gradient identity function during back-propagation. In a recent study, the quantizer itself is replaced by a non-linear differentiable function, so that a better approximation of its gradient contribution is

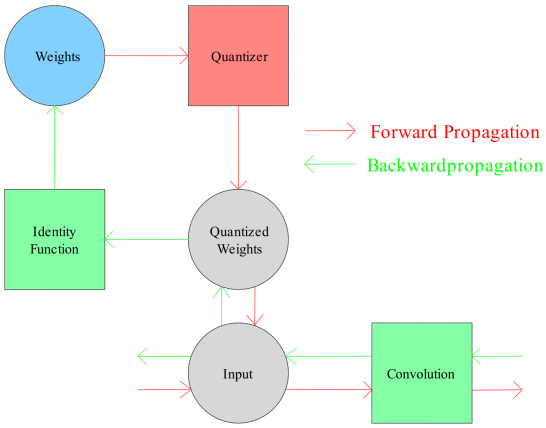


Figure 2.11. A simple straight-through estimator which approximates the gradient for the real-valued weights by replacing the zero-gradient quantizer with the identity function.

attained during back-propagation (Yang et al., 2019).

QAT can be applied to a newly defined model and affect training from scratch, or it can instead only be used to fine-tune a pre-trained model. The latter approach has shown to preserve accuracy better than the former (Wu et al., 2020).

2.3.1.4 Quantized Inference

After a model is quantized, normally to fixed-point parameters and/or activations, one must decide how to perform inference with it. While the task may be approached in different way, the implementation used in this thesis follows the method in Jacob et al. (2017), which is based on the arithmetic quantization scheme previously described. This is also the implementation used by TensorFlow (Jacob et al., 2017).

Given two floating-point matrices $r_1 \in \mathbb{R}^{m \times n}$ and $r_1 \in \mathbb{R}^{n \times p}$, the definition of matrix multiplication gives us the (i, j):th element of the resulting matrix r_3 by

$$r_3^{(i,j)} = \sum_{k=1}^{n} r_1^{(i,k)} r_2^{(k,j)}, \tag{2.24}$$

for $i \leq m$ and $j \leq p$. The quantized entries of the corresponding matrices are denoted as $q_\alpha^{(i,j)}$, $\alpha = [1,2,3]$. By using the quantization scheme in Equation 2.18 and definition of matrix multiplication in Equation 2.24 we get

$$\Delta_3(q_3^{(i,j)} - z_3) = \sum_{k=1}^{n} \Delta_1(q_1^{(i,k)} - z_1)\Delta_2(q_2^{(k,j)} - z_2), \tag{2.25}$$

which gives us the quantized output element as

$$q_3^{(i,j)} = z_3 + M \sum_{k=1}^{n} (q_1^{(i,k)} - z_1)(q_2^{(k,j)} - z_2) \tag{2.26a}$$

$$M := \frac{\Delta_1 \Delta_2}{\Delta_3}. \tag{2.26b}$$

During inference, each layer will compute its quantized activations with Equation 2.26. In this case q_1 are the previous layer's activations, q_2 is the weight matrix and q_3 are the resulting activations of the layer.

Most often a bias term is also added to the activations. While its zero-point is always zero, its scale is determined by the scales of the previous layer's activations along with the weight's scale (Jacob et al., 2017), as

$$\Delta_{\text{b}} = \Delta_1 \Delta_2 \tag{2.27a}$$

$$z_{\text{bias}} = 0. \tag{2.27b}$$

With inclusion of both the quantized bias q_b and integer rounding and clamping, this gives us our final equation for fixed-point quantized inference as

$$q_3^{(i,j)} = clamp(z_3 + \lfloor M(q_b + \sum_{k=1}^{n}(q_1^{(i,k)} - z_1)(q_2^{(k,j)} - z_2))\rceil). \tag{2.28}$$

2.3.2 Pruning

Pruning is a method of compressing a model by removing a specified amount of non-zero weights (or other architectural structures) in the network. As described in Blalock et al. (2020), the idea of weight-based pruning is to produce a model $f(\mathbf{x}; \mathbf{M} \odot \theta')$, given a base model $f(\mathbf{x}; \theta)$. Here θ are the parameters of the base model, θ' are the parameters to be pruned (possibly slightly altered from the original ones due to fine-tuning), $\mathbf{M} \in \{0,1\}^{dim(\theta')}$ is a binary pruning mask that sets some of the parameters to 0, and $\odot$ is the element-wise product operator. The mask $\mathbf{M}$ is more of a formal construct than an actual variable of interest, as the parameters to be pruned are mostly often set to zero iteratively. Finally, the parameters to prune are almost always only the weights $\mathbf{W}$ and not the bias, as pruning the bias may actually harm model accuracy (TensorFlow, 2021a).

The task of implementing pruning may be approached in multiple different ways (Blalock et al., 2020). Pruning parameters at arbitrary positions as mentioned above is normally called *unstructured pruning*, as opposed to pruning entire structures in the model's architecture; *structured pruning*. Comparing weight (unstructured) and filter (structured) pruning, the former wins in terms of compression ratio but loses in terms of hardware compatibility (Meng et al., 2020). This refers to the fact that unstructured pruning targets weights at arbitrary locations in the network, producing sparse weight matrices, which does not necessarily yield an inference speedup as the amount of necessary MAC operations stay the same. Structured pruning ultimately requires a new architecture definition, which means that some calculations can be skipped entirely. Achieving speedup with unstructured pruning requires either purpose-built hardware that enables loading of sparse matrices or utilization of sparse matrix-vector product operations (Zhu and Gupta, 2017), which has been realized in Narang et al. (2017).

According to Blalock et al. (2020), the most common way of choosing which parameters to prune is the algorithm used in Han et al. (2015). The procedure is to distribute scores to each weight of which the lowest ones are then removed. The remaining weights are then fine-tuned by additional training, after which the whole process is iterated until a desired weight sparsity is reached. The method is applied to a pre-trained model, as opposed to pruning a model from scratch. Each weight's score is normally based either on weight magnitude $|w|$, or the product of gradient and weight magnitude $|w\nabla_w J(\theta)|$, where the former is more common (Blalock et al., 2020). Omitting the scores and instead pruning randomly is also an option that has been explored, however with far worse performance in terms of accuracy, as expected.

Finally, one may choose not to prune an entire model and instead only specific layers. Choosing these with caution may yield better performance than by pruning a network uniformly (Blalock et al., 2020).

2.3.3 Other Methods

Additional methods in model compression and optimization beyond quantization and pruning exist, but will not be considered in this thesis. Some examples are Huffman Coding (Han et al., 2016), vector quantization (Le Tan et al., 2018) and knowledge distillation (Hinton et al., 2015).

3. Implementation

3.1 Baseline Networks

3.1.1 Dataset

All baseline models in this thesis were trained on the CIFAR-10 dataset (Krizhevsky, 2012); the most popular low resolution image dataset (Basha et al., 2020). CIFAR-10 contains 50 000 train and 10 000 test images in colour, with 10 classes and a total of 6000 images per class. The labels are common and relevant objects, such as *airplane*, *cat* or *truck*. The current state-of-the-art accuracy on the dataset is over 99% (Papers With Code, n.d) and the average human accuracy is 94% (Ho-Phuoc, 2019). Figure 3.1 showcases some example images from each class.

While it might be more relevant to use a larger dataset for generalizable results (such as ImageNet), low resolution datasets of small size facilitate a swift training procedure, which may be desirable in projects of time constraint. Training a model on the ImageNet dataset may require several months (Chrabaszcz et al., 2017).

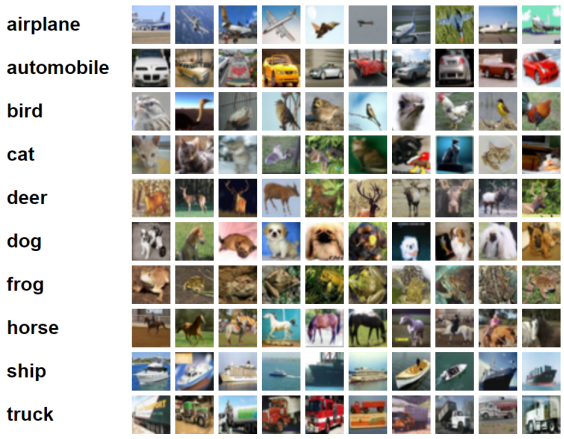


Figure 3.1. Ten randomly chosen images from each class in the CIFAR-10 dataset (Krizhevsky, 2012).

3.1.2 Network Architectures

Four different baseline models were constructed, of varying sizes and architectures. Due to the abundance of VGG-based CNN's (Khan et al., 2020), and the desire to obtain generalizable results, VGG's architectural features were the basis for two of the networks. The other two were based on MobileNetV2, as it is the fastest architecture with acceptable accuracy for embedded systems (benchmarked on a specific embedded device on the ImageNet dataset (Bianco et al., 2018)). For both architecture types, a small (∼0.2M parameters) and a large (∼2M parameters) variant were produced, as model size affects the compressibility (Krishnamoorthi, 2018). The VGG networks will hereafter be referred to as *ConvNet-S* and *ConvNet-L* (**Conv**olutional **Net**work), and the MobileNetV2 networks as *BRNet-S* and *BRNet-L* (**B**ottleneck **R**esidual **Net**work).

For both architectures the larger network was designed first, and the smaller created by systematically stripping layers and features until reaching approximately a tenfold decrease in size. TensorFlow (Abadi et al., 2016), a Python ML library originally developed by Google, was used for

Table 3.1. *Baseline convolutional neural networks.*

Model	Parameters	CIFAR-10 Accuracy	MAC ops (Million)
BRNet-L	2 289 034	90.2%	152.4
ConvNet-L	1 482 554	89.1%	107.8
BRNet-S	226 030	85.9%	13.6
ConvNet-S	161 194	89.5%	29.2

model building, hyperparameter searching and training. For BRNet-L the original MobileNetV2 architecture was a starting point, which initially yielded terrible validation accuracy. This was found to be due to the low resolution of the images in CIFAR-10, as the stride blocks' downsampling caused a tremendous information loss. Therefore multiple of the stride 2 blocks were replaced by stride 1, so that the feature map resolution never dropped below 8×8. BRNet-S was then designed by lowering the amount of repetitive bottleneck residual blocks, until a desired parameter count was reached. ConvNet-L was initially designed with convolutional layer stacks of two, as in VGG, each following a max pooling layer and a sequent stack of doubled filter amount. The model was then tweaked heuristically by removing some pooling layers and adding an intermediate convolutional layer with 160 filters. ConvNet-S was then constructed by removing some of the larger convolutional layers, and re-introducing some pooling layers. Each baseline model utilizes both batch normalization and dropout.

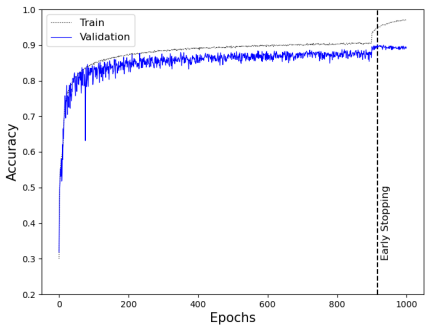
The learning and dropout rates were found for each model by performing hyperparameter grid search for a range of values of the two parameters. For each parameter set the model was trained for only a single epoch, in order to save time. The hyperparameters were then chosen based on validation accuracy. Hyperparameter tuning results are presented in Appendix B.

The models can be viewed in Table 3.1, where their parameter count, MAC operations and test accuracy are presented. While BRNet-S have more parameters than ConvNet-S it requires fewer MAC operations during inference. However even though BRNet-L is also MobileNetV2-based, it is in fact more computationally heavy than ConvNet-L. The full architectures for the baseline models can be viewed in Appendix A.

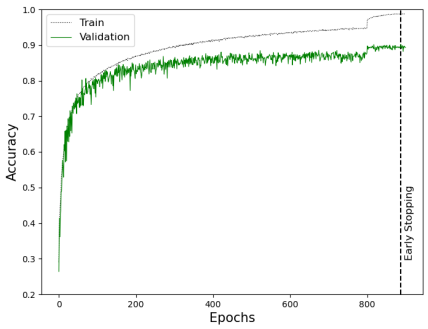
3.1.3 Training

To prevent overfitting, a data generator was used in order to expand the 50 000 images large CIFAR-10 training set with new images, produced by altering the original ones. The generator randomly shifts (max 10% of total width/height), rotates (max 15°) and horizontally flips the images. All baseline models were trained until validation accuracy convergence with this generator. It was then noted that the models instead had underfitted the original dataset, after which each model was additionally trained without image generation. Again, to prevent overfitting, *early stopping* was used, meaning that each model was saved after its best-performing epoch (in terms of validation accuracy). For each training procedure, 10% of the data was used for validation. The learning curves of the models can be viewed in Figures 3.2 and 3.3. The models were trained with CUDA on a computer with a GeForce GTX 1070 GPU.

Finally, all models were evaluated on the test set containing 10 000 previously unseen images. Their accuracy results are viewed in Table 3.1.

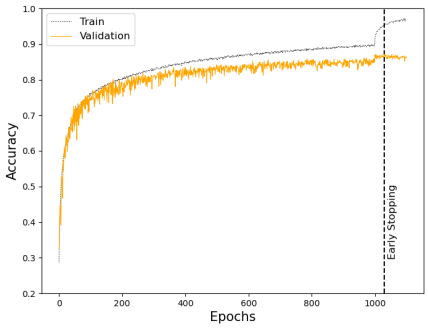


(a) Learning curve for ConvNet-S.

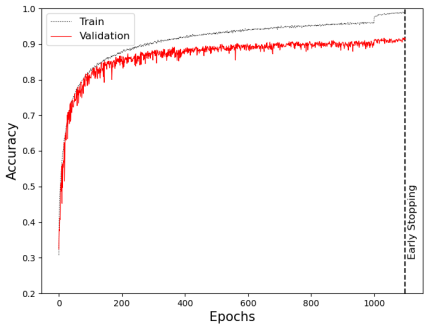


(b) Learning curve for ConvNet-L.

Figure 3.2. Learning curves for the ConvNet baseline networks.



(a) Learning curve for BRNet-S.



(b) Learning curve for BRNet-L.

Figure 3.3. Learning curves for the BRNet baseline networks.

3.2 Quantization

3.2.1 Post-Training Quantization

The TensorFlow Lite (TensorFlow, n.d.a) API was used to examine post-training quantization. The following quantization modes can be used:

- **Float16**: float16 weights, float32 activations
- **Dynamic range**: int8 weights, float32 activations
- **Full integer**: int8 weights, int8 activations
- **Int16 activations**: int8 weights, int16 activations

The *full integer* and *int16 activations* modes can also use a *float fallback* option, meaning that the activations in specific layers might revert to type float32 if it has no available integer implementation in TensorFlow. It should be noted that neither the weights nor activations can be quantized to fixed-point below 8-bits. This restriction motivated instead focusing on quantization aware training, which in TensorFlow has highly more customizable API calls.

The process of applying post-training quantization in TensorFlow is quite simple. First one creates a TFLite converter object with one of the quantization modes above specified, targetting the model to be quantized. If the quantization mode is either *full integer* or *int16 activations* a representative dataset of the original training data must also be defined (used to tweak the activations' ranges). A batch of 100 images from the training set was used for this implementation. Finally, the converter is executed and a quantized TFLite model is returned.

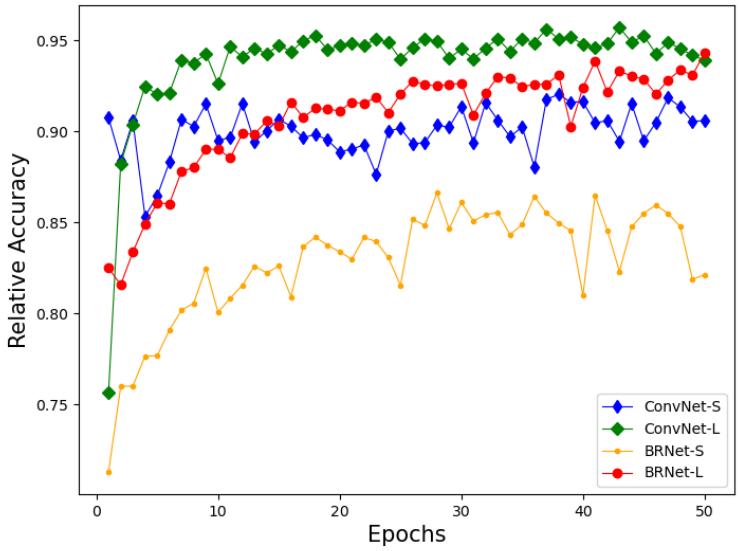


Figure 3.4. Relative accuracies for each quantization aware baseline model after being fine-tuned for a certain amount of epochs with 4-bit weights and activations.

3.2.2 Quantization Aware Training

The TensorFlow Model Optimization (TensorFlow, n.d.b) API was used to implement quantization aware training on the baseline models. The API lets the user specify certain parameters during QAT, namely:

- **Quantizer**: [*AllValues*, *LastValue*, *MovingAverage*]
- **Number of bits**: [2, . . . , 16] (integer)
- **Quantized range symmetry**: [*Asymmetric*, *Symmetric*]
- **Quantizer scope**: [*Per-tensor*, *Per-channel*]
- **Quantized range restriction**: [*Full range*, *Narrow range*]

Here *Quantizer* is the method of finding the minimum and maximum values of the tensor to be quantized. The *AllValues* quantizer uses values from all fine-tuning batches, while the *LastValue* quantizer only uses the last batch. Finally, the *MovingAverage* quantizer uses a moving average across all batches. The other parameters listed above have been explained in Chapter 2.

Using QAT on a model with the API means to initially annotate it as quantization aware, whereafter it is trained for a specified amount of epochs. The resulting model will still remain in floating-point, but with updated parameters that have been fine-tuned to adapt to the quantizers' addition to the loss function. When fine-tuning, it is naturally desired to let the model reach a validation accuracy convergence, however the amount of necessary epochs for this might vary depending on the model's architecture and size. Hence each baseline model was fine-tuned for an increasingly amount of epochs, and the amount of epochs needed for convergence was noted for future result gathering. The result is presented in Figure 3.4. Here, each baseline model was quantized with the *LastValue* quantizer, with 4-bit full range per-tensor symmetrical weights and asymmetrical activations. It was decided that ConvNet-S and ConvNet-L require 12 epochs, whereas BRNet-S and BRNet-L require 30 epochs.

While Tensorflow's floating-point QAT models include a built-in prediction function that simulates quantization, obtaining the fixed-point parameters explicitly is still necessary for inference

on embedded hardware. However, with TensorFlow these can only be acquired for the 8-bit case. Hence an algorithm to obtain the quantized model was instead manually crafted. Initially, the minimum and maximum values of the weights and activations were extracted from the QAT model (these are available in the API) and used to calculate the scales and zero-points for each layer and/or channels. Equation 2.22 was used as a starting point, but tweaked as to follow TensorFlow's hidden implementation as closely as possible. This was made possible by the fact that, again, extracting quantized parameters for 8-bit QAT models is available with TF. Hence the manually computed quantization parameters could be compared to TFs result, and the algorithm could be altered to mimic these.

The scaling parameters for the weights were computed for the asymmetrical case with

$$\Delta_w^{\text{asym, full}} = \frac{w_{max} - w_{min}}{2^N - 1}, \tag{3.1a}$$

$$\Delta_w^{\text{asym, narrow}} = \frac{w_{max} - w_{min}}{2^N - 2}, \tag{3.1b}$$

and for the symmetrical case with

$$\Delta_w^{\text{sym, full}} = \frac{max(|w_{min}|, |w_{max}|)}{(2^N - 2)/2}, \tag{3.2a}$$

$$\Delta_w^{\text{sym, narrow}} = \frac{max(|w_{min}|, |w_{max}|)}{2^{N-1} - 1}. \tag{3.2b}$$

w_{min} and w_{max} are the minimum and maximum values of the weights, extracted from the TF API. Further, the bias scales were computed with

$$\Delta_b = \Delta_w \Delta_a^{(\ell-1)}, \tag{3.3}$$

where $\Delta_a^{(\ell-1)}$ is the activation scale of the previous layer. The activation scales were computed with

$$\Delta_a = \frac{a_{max} - a_{min}}{2^N - 1}, \tag{3.4}$$

where a_{min} and a_{max} are the minimum and maximum activations. Finally, the zero-points were computed with

$$z_w^{\text{asym, full}} = -\lfloor \frac{w_{min}}{\Delta_w^{\text{asym, full}}} \rceil + 2^{N-1}, \tag{3.5a}$$

$$z_w^{\text{asym, narrow}} = \lfloor \frac{w_{min}}{\Delta_w^{\text{asym, narrow}}} \rceil + 2^{N-1} - 1, \tag{3.5b}$$

$$z_w^{\text{sym}} = 0, \tag{3.5c}$$

$$z_b = 0, \tag{3.5d}$$

$$z_a = -\lfloor \frac{a_{min}}{\Delta_a} \rceil + 2^{N-1}, \tag{3.5e}$$

$$z_a^{\text{batch norm}} = -\lfloor \frac{a_{min}}{\Delta_a} \rceil + 2^{N-1} - 1. \tag{3.5f}$$

The next step was to use the quantization parameters to determine the quantized weights and biases with Equation 2.19 or Equation 2.21. Lastly these fixed-point model parameters were stored in a custom JSON file. As for the activations, even though they are not model parameters they must be computed and quantized during runtime. Hence, the activation scales and zero-points were also stored in the exported file. Other necessary model parameters, such as batch normalization means and variances were also stored in this file. The full procedure can be viewed in Algorithm 3.1.

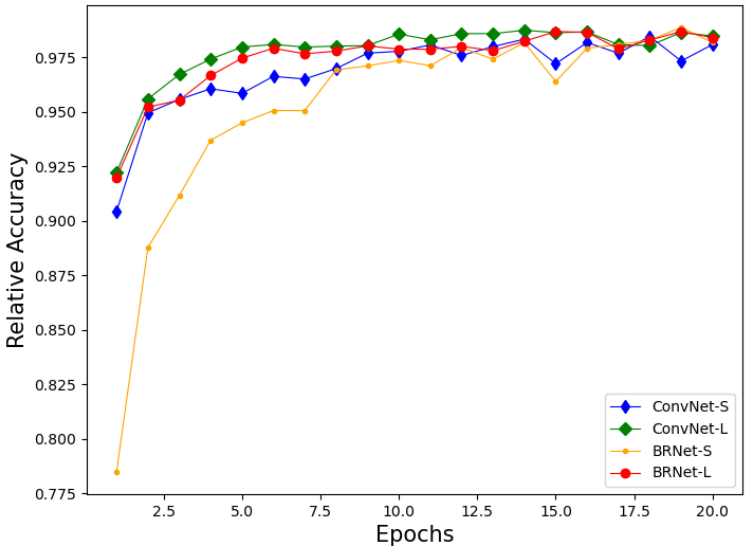


Figure 3.5. Relative accuracies for each pruned baseline model after being fine-tuned with the *ConstantSparsity* schedule for a certain amount of epochs with 70% sparsity.

Algorithm 3.1: Floating-point QAT model to fixed-point conversion.

for *Layers in Model* **do**
    Extract tensor/channel minimum and maximum values from QAT API
    Compute weight, bias and activation scales and zero-points from Equations 3.1 - 3.5
    Compute fixed-point weights and biases with Equation 2.19 or Equation 2.21
    Store fixed-point parameters, quantization parameters and other model parameters in JSON format

3.3 Pruning

As for quantization aware training, pruning was also implemented with the TensorFlow Model Optimization API, which allows the user to tweak multiple pruning properties, including:

- **Pruning Schedule**: [*ConstantSparsity*, *PolynomialDecay*]
- **Target sparsity**: [0, . . . , 0.99] (float, only for ConstantSparsity)
- **Initial sparsity**: [0, . . . , 0.99] (float, only for PolynomialDecay)
- **Final sparsity**: [0, . . . , 0.99] (float, only for PolynomialDecay)
- **Sparsity function exponent**: (float, only for PolynomialDecay)

Where *ConstantSparsity* prunes with a constant target sparsity throughout fine-tuning, the *PolynomialDecay* schedule iteratively changes its target sparsity according to a sparsity function. This function is defined by the *initial* and *final sparsity* as well as the *sparsity function exponent*. It should be noted that both schedules perform only unstructured pruning, as structured pruning is not yet supported in TensorFlow.

To prune a model in TensorFlow one initially defines and compiles a new model with a specified *pruning schedule*. The model is then fine-tuned with representative data (in this case with the same training and validation sets as for baseline training), preferably until convergence. As for

QAT, the amount of epochs needed for convergence might vary, and hence all baselines were fine-tuned with a fixed sparsity and their necessary amount of epochs was noted. This was done with the *ConstantSparsity* schedule with a *target sparsity* of 70%. The result is presented in Figure 3.5, where it was concluded that all models converge after approximately 14 epochs.

3.4 Hardware Implementation

3.4.1 ARM Cortex-A72 Details

The ARM Cortex-A72 microprocessor was the hardware chosen to implement the convolutional neural networks on. While other embedded hardware circuits, such as microcontrollers or SoCs, may seem more relevant than CPUs, the hassle of hardware design or working with limited memory (for model storage and/or inference computations) was considered too great for a project of this time span. It should still be noted that ARM processors are ubiquitous in embedded systems, and the use of microprocessors in certain SoC devices is in fact increasing (Koelling, n.d.).

The ARM Cortex-A series support an SIMD architecture called *Neon*, used to speedup computationally heavy applications, such as signal processing algorithms, video processing, and deep learning. Neon instructions allow, for instance, 4 32-bit float operations, but as many as 16 8-bit integer operations (arm, n.d.). Hence, exchanging floating-point arithmetics for fixed-point during inference could yield a reduction of up to $4\times$ in latency.

The implementation was done on a Raspberry Pi 4 Model B single-board computer, that hosts an ARM Cortex-A72. This enabled development within the Raspberry Pi OS, an operating system based on the Linux distribution Debian.

3.4.2 Embedded Inference

Forward propagation was implemented in C++, in order to obtain inference speedup results on the ARM processor. The implementation was made as an extension to previous code at Synective Labs. Initially, the algorithm loads a model architecture in JSON format and creates a model object with the same layers. All model parameters (weights, biases, batch normalization properties etc.) are then inserted into each layer. Finally, the forward pass of each layer is executed and a prediction is obtained. The procedure can be viewed in Algorithm 3.2.

Algorithm 3.2 was tested with the ConvNet architecture and baseline weights and biases, and the C++ model architecture was tweaked until the same inference result as TensorFlow's built-in prediction function was obtained. The BRNet architecture was not implemented, due to limited time. For quantized inference, the fixed-point weights and biases as well as quantization parameters (all obtained from Algorithm 3.1) were loaded into each layer, instead of the baseline parameters. Additionally, the fixed-point quantized inference in Equation 2.28 was included in each layer's forward pass function. Again, the code was tweaked until the inference results matched TensorFlows QAT prediction function (for the ConvNet architecture).

The original code library by Synective Labs included model building/loading and forward pass functions for common CNN layers. The implementation made for this thesis included adding support for some layers, reading and loading quantized weights and biases as well as quantization parameters, and performing quantized inference. Further, the code uses *Eigen* (Guennebaud et al., 2010) for matrix and vector multiplications, with 32-bit float as a standard data type for scalars. Hence, to achieve speedup with quantized inference, not only must the weights, biases and activations be quantized, but the default scalar data type must also be changed to integer.

Algorithm 3.2: Forward propagation procedure of the C++ implementation.

```
Load and create model architecture from JSON file
for Layers in Model do
    Load layer parameters from JSON file
for Layers in Model do
    Induce previous layer's activation (or input image for the first layer) as input to layer
    Execute forward pass of layer
    Return layer's activation
Return prediction
```

As a completely separate experiment, embedded inference was also explored with the TensorFlow Lite Inference API (TensorFlow, n.d.a), available for Python and C++ on Linux. As implementation was performed on a Raspberry Pi, the Linux distribution optimized for ARM boards could be used.

Pruning was not considered, as speedup with unstructured pruning would require sparse matrix-vector product operations (Zhu and Gupta, 2017), which was not realizable for this thesis.

4. Experiments and Results

4.1 Post-Training Quantization

As fixed-point (and not floating-point) quantization is the focus in this thesis, *full integer* and *int16 activations* were the only examined PTQ modes. The relative accuracies for the baseline networks with these two quantization modes can be viewed in Table 4.1. Recall that the relative accuracy is the test set accuracy of the quantized model divided by the test set accuracy of the baseline model.

For the BRNets, the final softmax activation layer was not supported with *full integer*, hence float fallback (FF) was needed. With *int16 activations*, the *Add*-layers used in the residual blocks were not supported even with float fallback, these results were hence omitted. The *int16 activations* mode is expected to perform better than *full integer*, which uses 8-bit activations, therefore since 8-bit weights and activations yield near to no accuracy degradation for both BRNet models, the same would almost certainly be the case for larger activation bit-widths.

Table 4.1. *Relative accuracies [%] of the fixed-point post-training quantized baseline models.*

Model	Full Integer	Int16 Activations
ConvNet-S	99.9	100.0
ConvNet-L	99.9	100.0
BRNet-S	99.5 (FF)	-
BRNet-L	100.1 (FF)	-

4.2 Quantization Aware Training

Arguably the most interesting parameters in quantization are the weight and activation bit-widths. Hence it was examined how each baseline's relative accuracy dropped by a decreasing amount of bits, both in the weights and activations separately, and also their interplay. All baselines were fine-tuned for their respective amount of epochs needed for convergence, as found in Figure 3.4, and with the same learning rate as during their respective initial training. Further, as the curves in Figure 3.4 seemed quite noisy, the accuracies for the final three epochs were averaged.

The other QAT parameters were determined according to TFLite's standard 8-bit quantization specification (TensorFlow, 2021b): symmetric, narrow range, per-tensor weights and asymmetric, full range, per-tensor activations and 32-bit symmetric, full range, per-tensor biases. The *LastValue* quantizer was used for the weights, and *MovingAverage* quantizer for the activations, motivated by an example in the API (TensorFlow, n.d.b). The results can be viewed in Figures 4.1 - 4.4.

In Figure 4.1, the smallest networks ConvNet-S and BRNet-S start to deteriorate notably at 3-bit weights, while the two other larger networks for this bit-width almost maintain the accuracy they had for larger number of bits. However, while ConvNet-S expectedly has the lowest relative accuracy at 2-bits, BRNet-S has the highest one. BRNet-L, which has the highest relative accuracy as the models initially deteriorate, falls below the smaller network ConvNet-L for 2-bits.

Figure 4.2 displays a contrary behaviour for the activations than that of the weights. Although

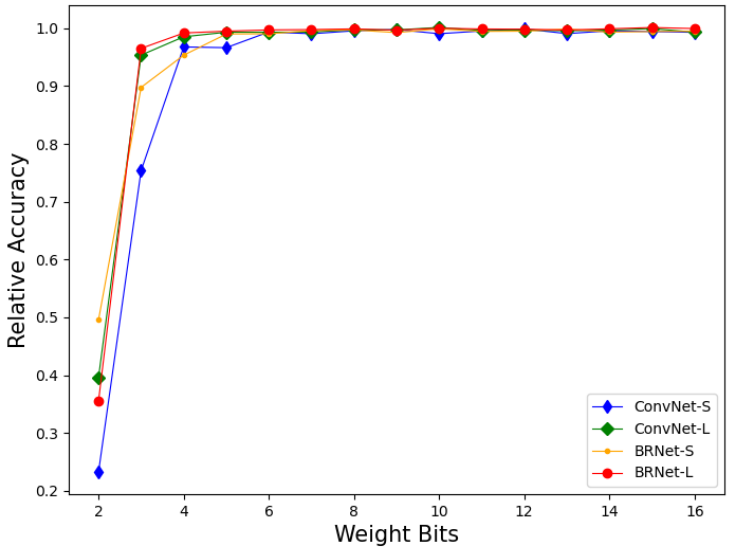


Figure 4.1. Relative accuracies for each baseline quantized with QAT for varying weight bit-width. Activations are fixed to 8-bits.

all models mostly uniformly deteriorate at 4-bit activations, the larger networks present a substantially larger accuracy drop for 3-bits when compared to the smaller networks. Suprisingly, the smallest network ConvNet-S maintains its accuracy the best. The 2-bit case, while still an interesting data point, is not relevant in a comparison of the models' relative accuracy drop, as they all have approximately 10% accuracy (which is the same as random guessing, as CIFAR-10 has 10 classes with an equal amount of samples each).

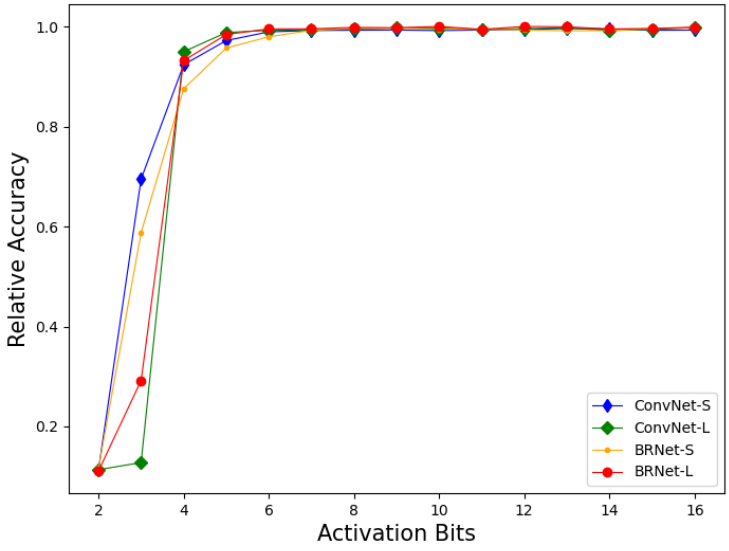


Figure 4.2. Relative accuracies for each baseline quantized with QAT for varying activation bit-width. Weights are fixed to 8-bits.

As for the cases where both the weight and activation bit-widths are varied, shown in Figures 4.3 and 4.4, the larger counterpart of each CNN architecture generally maintains their accuracy better than the smaller. However, as previously noted for Figure 4.2, the larger networks deteriorate faster than their smaller counterparts for low activation bit-widths. The opposite is the case for weight bit-widths.

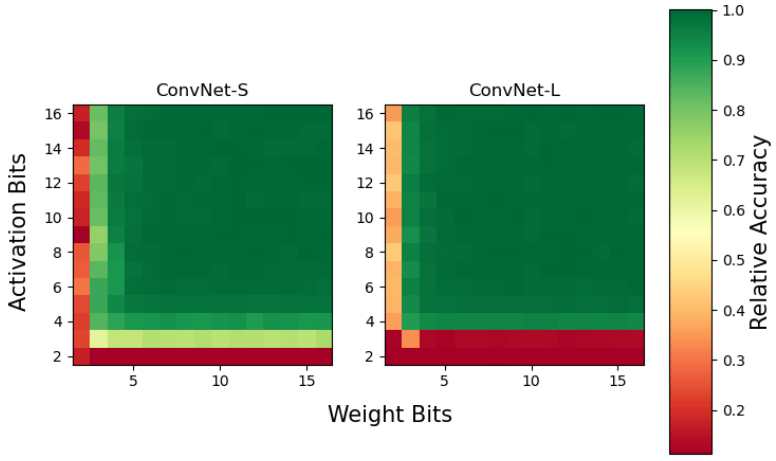


Figure 4.3. Relative accuracies for the ConvNet baselines quantized with QAT for varying weight and activation bit-widths. Best viewed in colour.

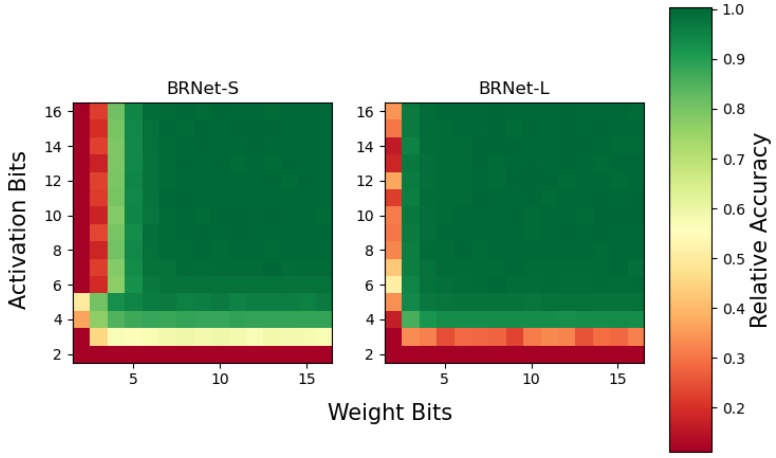


Figure 4.4. Relative accuracies for the BRNet baselines quantized with QAT for varying weight and activation bit-widths. Best viewed in colour.

Table 4.2. *TensorFlow's QAT parameters' effect on relative accuracy [%] for each baseline model. Each parameter (from Section 3.2.2) is separated by a horizontal line, and its best value for each baseline is highlighted in gray.*

QAT Parameter value	ConvNet-S	ConvNet-L	BRNet-S	BRNet-L
AllValues	80.5	88.2	-	-
LastValue	77.8	86.7	69.5	25.6
MovingAverage	69.2	88.1	70.6	23.7
Symmetric	77.8	86.7	69.5	25.6
Asymmetric	82.1	87.5	69.8	24.0
Per-Tensor	77.8	86.7	69.5	25.6
Per-Channel	78.8	87.9	74.7	34.3
Full range	74.9	87.3	72.4	31.2
Narrow range	77.8	86.7	69.5	25.6

Finally, the effect of all other QAT parameters was also explored. By using QAT with 4-bit symmetric, per-tensor, narrow range weights with the *LastValue* quantizer as foundation, each parameter was individually altered to compile the results displayed in Table 4.2. Throughout this experiment, activations were always quantized as 4-bit asymmetric, per-tensor and full range with the *MovingAverage* quantizer.

One note on Table 4.2, first looking at the choice of quantizer, is that the BRNet architecture was incompatible with the *AllValues* quantizer. While the exact cause has not been explored, it is suspected to be the *Add*-layers in the residual blocks. The *LastValue* and *MovingAverage* quantizers seem to perform quite inconsistently, as they have either negative or positive effects on the relative accuracy, from case to case. Per-channel weights are preferred in all cases, while the choice of range symmetry and restriction is not as apparent. While asymmetric and full range in most cases yields better accuracy, symmetric and narrow range are outperforming options in some cases.

4.3 Pruning

The parameter of main interest in pruning is sparsity; the percentage of weights removed from the original model. Sparsity versus relative accuracy was explored by using the *ConstantSparsity* schedule with increasing target sparsity on all baseline models, each fine-tuned for the amount of epochs needed for convergence, as found in Figure 3.5, and with their respective learning rate used for initial training. The remaining parameters in the schedule were set to their default values. As for QAT, the accuracies of the final three epochs were averaged. The result can be viewed in Figure 4.5. It should be noted that the last data point is for 99% sparsity, and not 100%.

Instead of setting a constant target sparsity, one may also prune by letting the target follow a function over fine-tuning iterations. This was explored with the *PolynomialDecay* schedule, by altering both the initial sparsity and sparsity function exponent. In all experiments, the final sparsity was set to 90%. The results can be viewed in Figures 4.6 and 4.7.

First looking at the *ConstantSparsity* schedule in Figure 4.5, all models slowly deteriorate at 70% sparsity and upward, and the drop in accuracy is distinct when pruned with over 90% sparsity. Mostly, there is no apparent difference between the models, with the exception of BRNet-S which clearly loses its accuracy faster than the other ones.

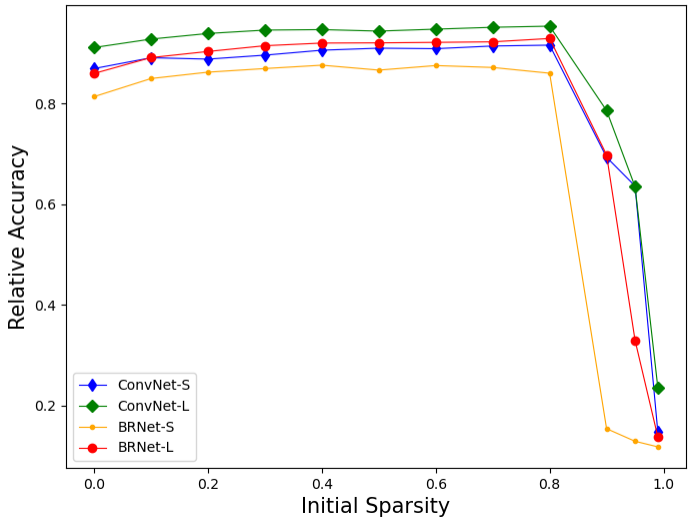


Figure 4.6. Relative accuracies for each pruned baseline for varying initial sparsity with the *PolynomialDecay* schedule. The final sparsity is set to 90%.

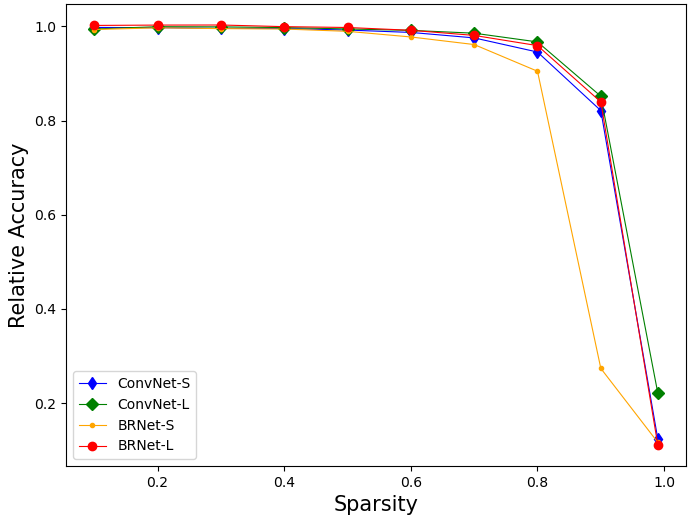


Figure 4.5. Relative accuracies for each pruned baseline for varying sparsity with the *ConstantSparsity* schedule.

In Figure 4.6, with the *PolynomialDecay* schedule, all models favor a large initial sparsity close to the final sparsity of 90%. However, when the initial sparsity exceeds the final sparsity, all model accuracies degrade quickly. Interestingly, even though the final sparsity is set to 90%, BRNet-S maintains acceptable accuracy, as opposed to the pronounced deterioration noted for 90% target sparsity with the *ConstantSparsity* schedule in Figure 4.5.

Finally, it can be seen in Figure 4.7 that all models seem to favor a polynomial exponent of 1. For all models except BRNet-L, increasing the exponent initially lowers the relative accuracy,

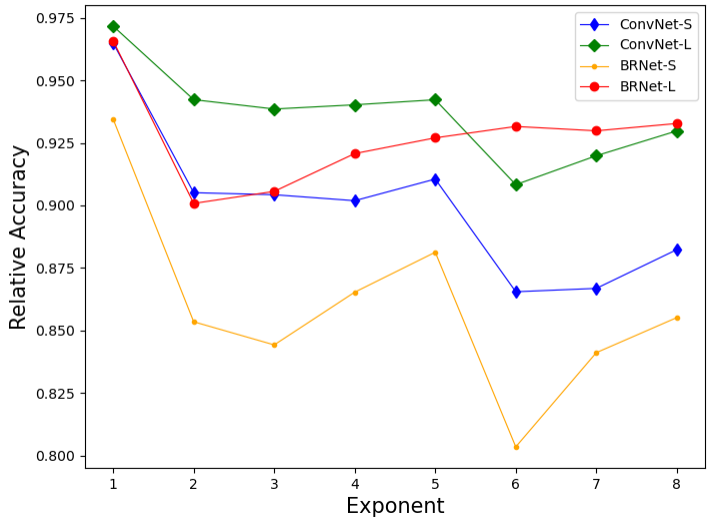


Figure 4.7. Relative accuracies for each pruned baseline for varying sparsity function exponent with the *PolynomialDecay* schedule. The final sparsity is set to 90%.

whereafter is gradually increases for high exponent values, though it is still lower than the accuracy when the exponent is 1.

4.4 Embedded Inference

4.4.1 C++ Implementation

During implementation it was noted that changing Eigen's scalar data type to integer from float for all C++ code would require an amount of work that was beyond the scope of the thesis. Instead, latency experiments were made for a single layer network, specifically a depthwise convolutional layer, as its pre-existing implementation in the codebase was easier to modify than for instance the convolutional layer. A network with three randomly initiated float or integer $3 \times 3 \times 1$ filters and 3×1 biases was created, and inferred with a random $512 \times 512 \times 3$ input. Each data point was averaged over 1000 observations. The results can be viewed in Table 4.3.

Table 4.3. *Inference latency for a single depthwise convolutional layer in C++, on an ARM Cortex-A72.*

Scalar data type	Inference latency [μs]
float	17 665
int8_t	19 544

It was noted that fixed-point 8-bit integer arithmetic was suprisingly slower than floating point. To ensure that the ARM processor's SIMD functionality was utilized, further tests were conducted. This time, instead of initializing a CNN, the Hadamard product (element-wise product) of two vectors of size 100000×1 was explored. Firstly, two C++ arrays were multiplied by using a for-loop through each element. Secondly, two Eigen matrices (of dimension 100000×1) were multiplied in the same manner. And finally, two Eigen matrices were multiplied by Eigen's `cwiseProduct()` function. Again, each latency was averaged over 1000 runs. The results can be viewed in Table 4.4.

Table 4.4. *Computation time for the Hadamard product of two* 100000×1 *vectors in C++, on an ARM Cortex-A72.*

Vector and data type	Time [μs]
float array (for-loop)	1 232
int8_t array (for-loop)	282
Eigen float matrix (for-loop)	1 405
Eigen int8_t matrix (for-loop)	1 009
Eigen float matrix (`cwiseProduct()`)	315
Eigen int8_t matrix (`cwiseProduct()`)	44

For standard C++ arrays, 8-bit integer computations are approximately $4.4\times$ faster than 32-bit floats. When using Eigen matrices and for-loops, integers are only $1.4\times$ faster, and when using Eigen's `cwiseProduct()` function, integers are $7.2\times$ faster. The latter Eigen approach is faster than standard C++ arrays for both the floating- and fixed-point cases.

4.4.2 TensorFlow Lite Implementation

The TensorFlow Lite Inference API was also used for each *full integer* post-training quantized baseline model. The results were again averaged over 1000 observations, and can be viewed in Table 4.5. The BRNet architectures required float fallback (FF) during quantization, and their latency speedup might hence not be as great as for the integer-only case.

For this implementation, speedup gains up to $5.5\times$ were achieved. The larger models present larger speedups than their smaller architectural counterpars.

Table 4.5. *Inference latency for each 8-bit post-training quantized baseline model, with the TFLite Inference Python API on an ARM Cortex-A72.*

Model	Baseline [μs]	PTQ (full integer) [μs]	Speedup
ConvNet-S	15 518	5 789	$2.7\times$
ConvNet-L	102 854	18 870	$5.5\times$
BRNet-S	16 228	11 464 (FF)	$1.4\times$
BRNet-L	137 572	42 950 (FF)	$3.2\times$

5. Discussion

5.1 Post-Training Quantization

No matter which baseline architecture is used, applying PTQ with 8-bit weights and activations has practically no effect on accuracy. The results are difficult to compare to QAT, as both methods perform equally well for 8-bits, but it would still be reasonable to assume that quantization without fine-tuning would perform worse at lower bit-widths. While this might be the case, latency on most hardware would not benefit from using weights or activations below 8-bits, which would mean that PTQ for these use cases is sufficient. However, as TensorFlow's fixed-point PTQ still uses data to calibrate the activations' ranges, there is no reason not to use QAT and fine-tune the model instead, as this most likely means a lower risk of accuracy degradation. Lastly, recent papers have demonstrated PTQ methods which do not use any data (Cai et al., 2020), (Nagel et al., 2019). These methods also preserve accuracy well, and could hence be more desirable than QAT for the 8-bit case, as dropping the requirement of calibration data is a clear advantage to both QAT and TensorFlow's PTQ.

5.2 Quantization Aware Training

As with PTQ, quantization aware training works flawlessly for all baseline models at 8-bit weights and activations. Again, since typical hardware does not allow low-bit multiplications and perform all operations with at least 8-bit data types, there is often no reason to quantize models to bit-widths below this point. However, if custom hardware that enables faster low-bit multiplications is used, the accuracy degradation is evident below 4-bits, for both weights and activations. The larger baseline models handles weight quantization better, as expected due to their larger parameter count (Krishnamoorthi, 2018). However, contrary to expectations, they are more affected by activation quantization than their smaller counterparts. While the reason is not apparent, a hypothesis is that the amount of epochs needed for convergence is not the same for low-bit activations, and hence Figure 3.4 is not valid for quantization of lower bit-widths. More specifically, if the increase in needed epochs is not linear in terms of model parameters it is plausible that in these cases the smaller models do still converge, while the larger do not.

When looking at both weight and activation bit-widths at the same time, in Figures 4.3 and 4.4, the larger models are again, as expected, not as affected by weight quantization (as a larger portion of their area is dark green). However, they are simultaneously much more sensitive to low-bit activations. When either low-bit weights or activations cause significant accuracy degradation, virtually no accuracy is gained by increasing the other parameter. For instance, with 2-bit activations the relative accuracy is the same with either 2- or 16-bit weights (for all models). The BRNet architecture is more sensitive to low weight bit-widths than the ConvNet architecture. Further, BRNet-S is more sensitive to low activation bit-widths than ConvNet-S. While the more complex architecture might contribute to these observations, insufficient amount of epochs during fine-tuning could again be the reason. Finally, apart from low-bit activations the larger models are generally less sensitive to quantization. When both weight and activations are greater or equal to 5-bits, all models seem to preserve their accuracy without noticable degradation.

As for the other QAT parameters, it is difficult to confidently state which options generally work best. Asymmetric, per-channel, full range weights seem to be preferred in most cases. This is

also expected, as these are the values that yields a quantization scheme with maximum amount of flexibility; with asymmetric weights the zero-point variable is included, per-channel means that multiple quantizers will be used for each layer, and full range includes an extra bit to the quantization range. Still, symmetric and narrow range is apparently favorable in some cases. As for quantizer, the best performing options seem to be quite random. Many entries in Table 4.2 only differ some small percentage points per parameter value, and it is hence not certain that the highlighted choice is guaranteed to be the best, due to the stochastic nature of neural network training. Ultimately, these variables should most likely be viewed as hyperparameters, and tailored with caution for each model.

5.3 Pruning

For all models, pruning 70% of the weights with the *ConstantSparsity* schedule, in Figure 4.5, yields almost no accuracy deterioration. All models but BRNet-S can be pruned as much as 90% while still preserving 80% accuracy. Why BRNet-S is the most sensitive model is unclear. It is possible that the BRNet architecture is more difficult to fine-tune, and that BRNet-L does not show a similar trend due to its size and redundancy. As all models' accuracies decline at 99% sparsity, BRNet-L in fact performs worse than ConvNet-L, which again could be an indication of insufficient fine-tuning.

When pruning with *PolynomialDecay*, the initial sparsity does not seem matter much, as long as it is lower than the final sparsity, according to Figure 4.6. Interestingly, for these cases BRNet-S actually maintains its accuracy far better than with *ConstantSparsity*. Hence, it would seem that models sensitive to pruning should be pruned less aggressively at first, at least for large final target sparsities. Furthermore, according to Figure 4.7, all models seem to prefer a polynomial exponent equal to 1, in other words a linear polynomial function. When comparing the methods in the 90% sparsity case, there doesn't seem to be much of a difference (except for BRNet-S).

5.4 Embedded Inference

5.4.1 C++ Implementation

The ConvNet architecture was successfully implemented in C++ on the ARM processor, with approximately the same output as TensorFlow for both the quantized and non-quantized cases. However, unfortunately no speedup gains were obtained with quantization in this implementation. When comparing approaches to computing the Hadamard product in Table 4.4, it is evident that Eigen's matrix multiplication utilizes a very efficient technique, as computations with this method was much faster than normal C++ arrays. It is unclear why an inference speedup could be noted for the Hadamard product, but not the depthwise convolutional layer in Table 4.3. Obtaining inference speedups for the baseline architectures in the C++ implementation would require a thorough examination of Eigen's matrix-multiply operations in depth, which is beyond the scope of this thesis. For normal C++ arrays, however, it is clear that a latency reduction of at least $4\times$ is realizable on the ARM processor.

5.4.2 TensorFlow Lite Implementation

With the TFLite implementation, Table 4.5 shows that remarkable speedup gains can be obtained with the quantized models. The speedup seems to be related to model size, as both larger architectural variants have a far greater relative latency reduction. The BRNet models present a lower speedup than the ConvNet models. Whether this is due to a more complex architecture or because of float fallback is unclear.

5.5 Generalizability of the Results

The fact that all results are gathered for only two CNN architecture types, one dataset, and a single problem formulation (only classification was examined, not regression), must be taken into consideration. The baselines were based on the popular VGG and embedded-friendly MobileNet architectures in order to reflect the current state of the art models, but whether the results are generalizable to other problem formulations is still difficult to conclude. However, as all baselines are quite small compared to models used for more complex datasets, it is likely that the observed quantization and pruning limits are at least not too generous, but the opposite.

Similarly, only one hardware type was explored. The ARM Cortex-A72 was chosen partially due to the ubiquity of ARM processors in embedded systems, however all inference latency results are greatly tied to the baseline models. It is difficult to state whether similar computational reductions are achievable for other networks or hardware, but the results give an idea of what order of latency speedups are obtainable by the means of 8-bit integer quantization.

5.6 Future Work

An interesting line of future research is to examine multiple datasets, and additional CNN architectures. This would yield a more general guideline as to how and when quantization should be performed on a given problem formulation. As binary quantization has been implemented with excellent accuracy (Courbariaux et al., 2016), a further development of the results presented in this thesis is to learn why low-bit quantization showed enormous accuracy degradations. An especially interesting question is also why large models seem to be more sensitive to low-bit activations than their smaller architectural counterparts.

As for pruning, a future topic is to examine how pruning only certain layers affects the accuracy, as this is supposedly preferable compared to pruning the entire model (Blalock et al., 2020). Further, achieving speedup with structured pruning would give an idea of how pruning can affect latency, and not only how it affects relative accuracy for different sparsities. Finally, combining pruning and quantization to observe effects on model compression, accuracy and inference latency is a most profitable potential future work, in terms of understanding interplay between CNN optimization methods.

Finally, concerning embedded inference, further exploring Eigen's built-in matrix operations could yield an understanding of why no speedups were obtained with the C++ inference implementation. An addition to the presented hardware results is to also explore inference latency on multiple hardware types. Investigating for example FPGAs, SoCs and focusing on custom hardware that enable speedups with uncommon bit-widths below 8 could give a thorough trade-off between inference latency and model accuracy not only for 8-bits, as presented in this thesis, but for a large range of bit-widths.

6. Conclusion

Convolutional Neural Networks are arguably one of the most interesting technical advancements today, due to their enormous success in multiple computer vision problems. Implementing them on embedded systems is however a difficult task; one that often requires the networks to be optimized in terms of either size or computational efficiency. This thesis has successfully presented results that provide a guideline for how, and when, the optimization methods quantization and pruning can be applied in order to achieve inference latency speedup or model compression.

Four convolutional neural networks, based on two relevant and popular architectures, were implemented and trained on the CIFAR-10 dataset with almost human accuracy on the test set. Both quantization and pruning were thereafter implemented and examined on these baseline CNN models. For quantization aware training, the larger networks are generally more robust to quantization but suprisingly more sensitive to low activation bit-widths, which could be due to insufficient fine-tuning. All models can be quantized with both 5-bit weight and activations, and still maintain almost full test accuracy. As for pruning, approximately 70% of the weights can be removed with barely any effect on accuracy.

For 8-bit quantization, which is an upper limit of what is favorable on most hardware, both post-training quantization and quantization aware training works without any accuracy loss. For this case, the networks' inference latency has been successfully explored on an ARM Cortex-A72 microprocessor, with up to 5.5$\times$ speedup compared to the baseline models with floating-point parameters. Larger models seem to gain more latency reduction from quantization than small models.

The findings of this thesis relate to multiple potential future research topics, such as further examining large models' low-bit activation sensitivity, latency reduction with structured pruning or quantized embedded inference with bit-widths lower than 8 on custom hardware. While the choice of baseline models and dataset were made as to reflect today's research and industry, whether the results are generalizable or not is difficult to state. Still, the valuable insights obtained in this thesis can be used as starting points for investigating how other popular CNN optimization methods affect model precision, as well as how to improve inference latency on other architectures or hardware types.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mane, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viegas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems.

Alexandrov, O. (2004). File:gradient descent.png. `https://commons.wikimedia.org/wiki/File:Gradient_descent.png`. Downloaded 2021-05-03.

Anwar, S., Hwang, K., and Sung, W. (2015). Structured pruning of deep convolutional neural networks.

arm (n.d.). Arm neon. `https://www.arm.com/why-arm/technologies/neon`. Downloaded 2021-04-27.

Banner, R., Nahshan, Y., Hoffer, E., and Soudry, D. (2019). Post-training 4-bit quantization of convolution networks for rapid-deployment.

Basha, S. S., Dubey, S. R., Pulabaigari, V., and Mukherjee, S. (2020). Impact of fully connected layers on performance of convolutional neural networks for image classification.

Bengio, Y., Léonard, N., and Courville, A. (2013). Estimating or propagating gradients through stochastic neurons for conditional computation.

Berrar, D. (2018). Cross-validation.

Bianco, S., Cadene, R., Celona, L., and Napoletano, P. (2018). Benchmark analysis of representative deep neural network architectures. *IEEE Access*, 6:64270â64277.

Blalock, D., Ortiz, J. J. G., Frankle, J., and Guttag, J. (2020). What is the state of neural network pruning?

Cai, Y., Yao, Z., Dong, Z., Gholami, A., Mahoney, M. W., and Keutzer, K. (2020). Zeroq: A novel zero shot quantization framework.

Choi, J., Wang, Z., Venkataramani, S., Chuang, P. I.-J., Srinivasan, V., and Gopalakrishnan, K. (2018). Pact: Parameterized clipping activation for quantized neural networks.

Chrabaszcz, P., Loshchilov, I., and Hutter, F. (2017). A downsampled variant of imagenet as an alternative to the cifar datasets.

Courbariaux, M., Hubara, I., Soudry, D., El-Yaniv, R., and Bengio, Y. (2016). Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

Guennebaud, G., Jacob, B., et al. (2010). Eigen v3. http://eigen.tuxfamily.org. Downloaded 2021-05-05.

Han, S., Mao, H., and Dally, W. J. (2016). Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding.

Han, S., Pool, J., Tran, J., and Dally, W. J. (2015). Learning both weights and connections for efficient neural networks.

He, K., Zhang, X., Ren, S., and Sun, J. (2015). Deep residual learning for image recognition.

Hinton, G., Vinyals, O., and Dean, J. (2015). Distilling the knowledge in a neural network.

Ho-Phuoc, T. (2019). CIFAR10 to compare visual recognition performance between deep neural networks and humans.

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. (2017). Mobilenets: Efficient convolutional neural networks for mobile vision applications.

Huang, G., Liu, Z., van der Maaten, L., and Weinberger, K. Q. (2016). Densely connected convolutional networks.

Hughes, C. (2015). Single-instruction multiple-data execution.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift.

Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H., and Kalenichenko, D. (2017). Quantization and training of neural networks for efficient integer-arithmetic-only inference.

Jaderberg, M., Vedaldi, A., and Zisserman, A. (2014). Speeding up convolutional neural networks with low rank expansions.
Jain, S. R., Gural, A., Wu, M., and Dick, C. (2019). Trained uniform quantization for accurate and efficient neural network inference on fixed-point hardware.
Jin, J., Dundar, A., and Culurciello, E. (2015). Flattened convolutional neural networks for feedforward acceleration.
Khan, A., Sohail, A., Zahoora, U., and Qureshi, A. S. (2020). A survey of the recent architectures of deep convolutional neural networks. *Artificial Intelligence Review*, 53(8):5455–5516.
Kiranyaz, S., Avci, O., Abdeljaber, O., Ince, T., Gabbouj, M., and Inman, D. (2019). 1d convolutional neural networks and applications: A survey.
Koelling, T. (n.d.). User-customizable arm-based socs for next-generation embedded systems.
Krishnamoorthi, R. (2018). Quantizing deep convolutional networks for efficient inference: A whitepaper.
Krishnan, G., Du, X., and Cao, Y. (2019). Structural pruning in deep neural networks: A small-world approach.
Krizhevsky, A. (2012). Learning multiple layers of features from tiny images.
Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks.
Lam, M., Yedidia, Z., Banbury, C., and Reddi, V. J. (2020). Quantized neural network inference with precision batching.
Le Maire, J., Brunie, N., de Dinechin, F., and Muller, J.-M. (2016). Computing floating-point logarithms with fixed-point operations.
Le Tan, D.-K., Le, H., Hoang, T., Do, T.-T., and Cheung, N.-M. (2018). Deepvq: A deep network architecture for vector quantization.
LeNail, A. (2019). Nn-svg: Publication-ready neural network architecture schematics.
Li, H., Kadav, A., Durdanovic, I., Samet, H., and Graf, H. P. (2017). Pruning filters for efficient convnets.
Lindholm, A., Wahlström, N., Lindsten, F., and Schön, T. B. (2021). *Supervised Machine Learning*. `https://smlbook.org`.
Meenakshi, M. (2020). Machine learning algorithms and their real-life applications: A survey.
Meng, F., Cheng, H., Li, K., Luo, H., Guo, X., Lu, G., and Sun, X. (2020). Pruning filter in filter.
Moons, B., Bankman, D., and Verhelst, M. (2018). Embedded deep learning: Algorithms, architectures and circuits for always-on neural network processing.
Moons, B., Goetschalckx, K., Berckelaer, N. V., and Verhelst, M. (2017). Minimum energy quantized neural networks.
Nagel, M., van Baalen, M., Blankevoort, T., and Welling, M. (2019). Data-free quantization through weight equalization and bias correction.
Narang, S., Elsen, E., Diamos, G., and Sengupta, S. (2017). Exploring sparsity in recurrent neural networks.
Nash, W., Drummond, T., and Birbilis, N. (2018). A review of deep learning in the study of materials degradation.
Nayak, P., Zhang, D., and Chai, S. (2019). Bit efficient quantization for deep neural networks.
Papers With Code (n.d). Image classification on CIFAR-10. `https://paperswithcode.com/sota/image-classification-on-cifar-10`. Downloaded 2021-04-07.
Paupamah, K., James, S., and Klein, R. (2020). Quantisation and pruning for neural network compression and regularisation.
Roth, W., Schindler, G., Zöhrer, M., Pfeifenberger, L., Peharz, R., Tschiatschek, S., Fröning, H., Pernkopf, F., and Ghahramani, Z. (2020). Resource-efficient neural networks for embedded systems.
Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. (2015). ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252.
Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C. (2019). Mobilenetv2: Inverted residuals and linear bottlenecks.
Shanchen, P., Shuo, W., Alfonso, R.-P., Pibao, L., and Xun, W. (2019). An artificial intelligent diagnostic system on mobile android terminals for cholelithiasis by lightweight convolutional neural network.
Simonyan, K. and Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2014). Going deeper with convolutions.

Taylor, G., Burmeister, R., Xu, Z., Singh, B., Patel, A., and Goldstein, T. (2016). Training neural networks without gradients: A scalable admm approach.

TensorFlow (2021a). Pruning comprehensive guide. `https://www.tensorflow.org/model_optimization/guide/pruning/comprehensive_guide`. Downloaded 2021-03-31.

TensorFlow (2021b). Tensorflow lite 8-bit quantization specification. `https://www.tensorflow.org/lite/performance/quantization_spec`. Downloaded 2021-04-28.

TensorFlow (n.d.a). Deploy machine learning models on mobile and iot devices. `https://www.tensorflow.org/lite`. Downloaded 2021-04-23.

TensorFlow (n.d.b). Optimize machine learning models. `https://www.tensorflow.org/model_optimization`. Downloaded 2021-04-14.

TensorFlow Model Optimization team (2020). Quantization aware training with tensorflow model optimization toolkit - performance with accuracy. `https://blog.tensorflow.org/2020/04/quantization-aware-training-with-tensorflow-model-optimization-toolkit.html`. Downloaded 2021-03-31.

Wu, H., Judd, P., Zhang, X., Isaev, M., and Micikevicius, P. (2020). Integer quantization for deep learning inference: Principles and empirical evaluation.

Yang, J., Shen, X., Xing, J., Tian, X., Li, H., Deng, B., Huang, J., and Hua, X. (2019). Quantization networks.

Zeiler, M. D. and Fergus, R. (2013). Visualizing and understanding convolutional networks.

Zhang, D., Yang, J., Ye, D., and Hua, G. (2018). Lq-nets: Learned quantization for highly accurate and compact deep neural networks.

Zhu, C., Han, S., Mao, H., and Dally, W. J. (2017). Trained ternary quantization.

Zhu, M. and Gupta, S. (2017). To prune, or not to prune: exploring the efficacy of pruning for model compression.

Zmora, N., Jacob, G., Zlotnik, L., Elharar, B., and Novik, G. (2019). Neural network distiller: A python package for dnn compression research.

Appendix A.
Baseline Model Architectures

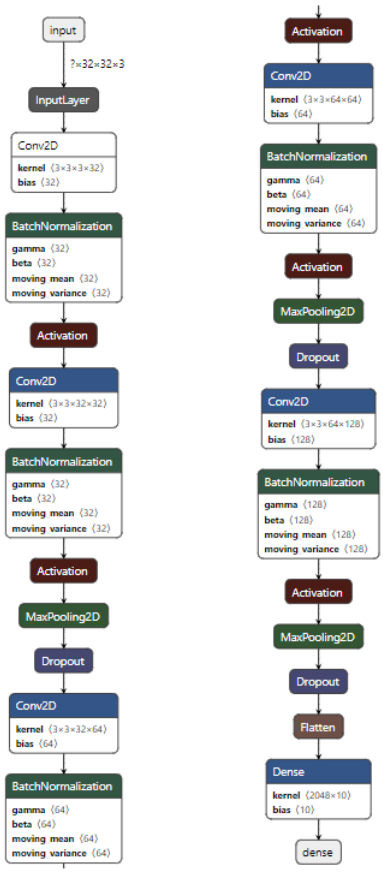


Figure A.1. ConvNet-S architecture.

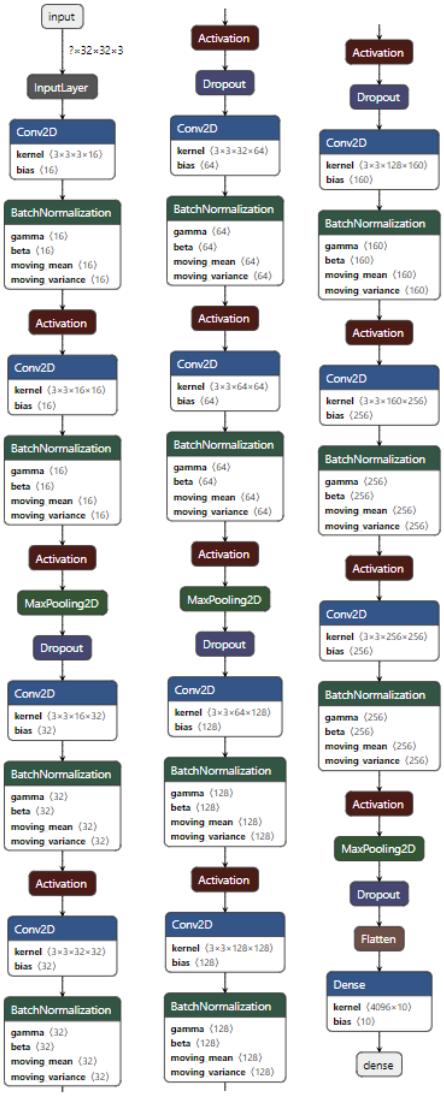


Figure A.2. ConvNet-L architecture.

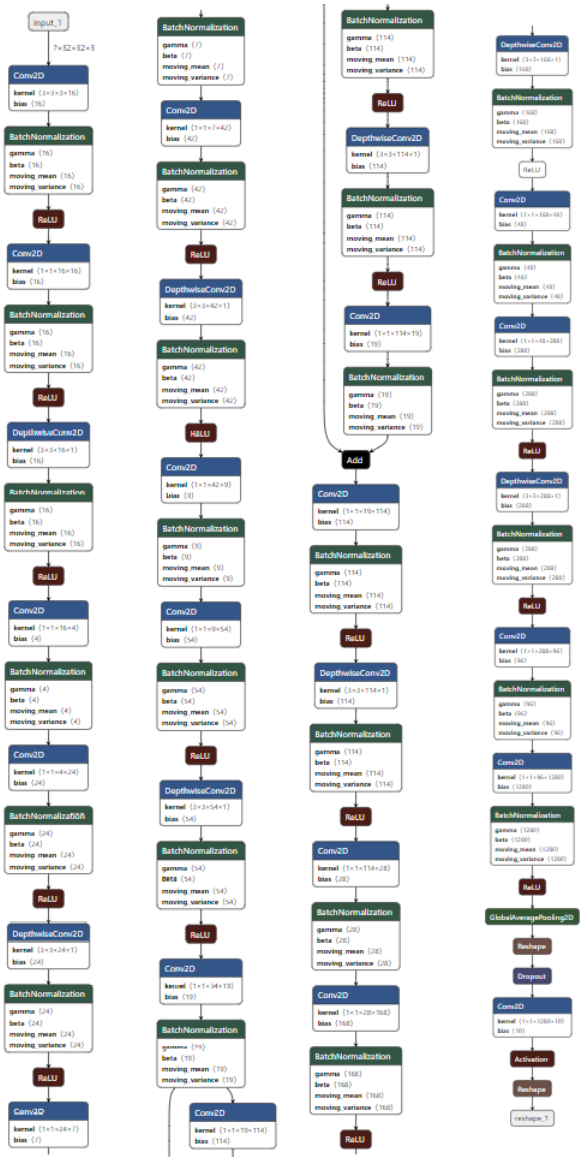


Figure A.3. BRNet-S architecture.

Figure A.4. BRNet-L architecture.

Appendix B.
Hyperparameter Tuning

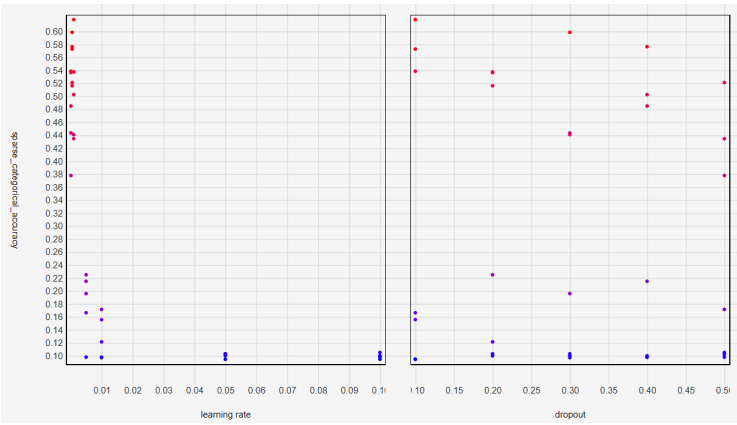


Figure B.1. Hyperparameter tuning for ConvNet-S.

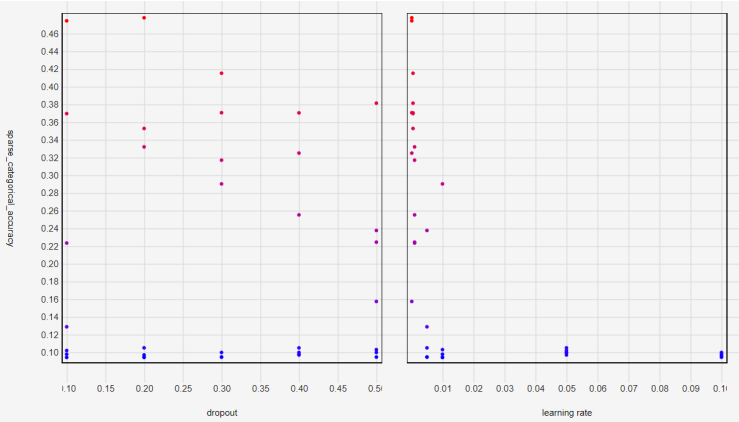


Figure B.2. Hyperparameter tuning for ConvNet-L.

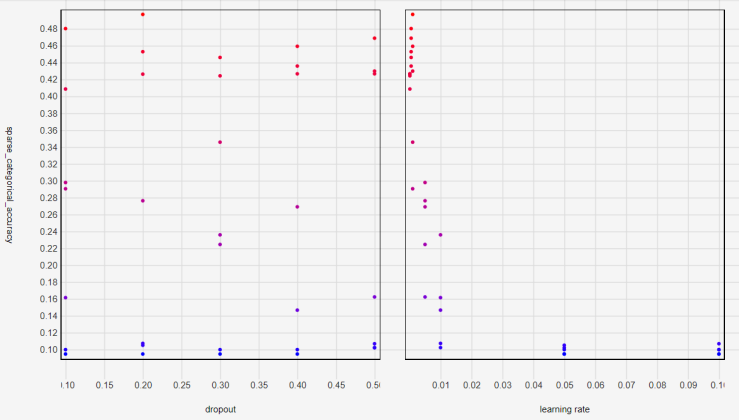


Figure B.3. Hyperparameter tuning for BRNet-S.

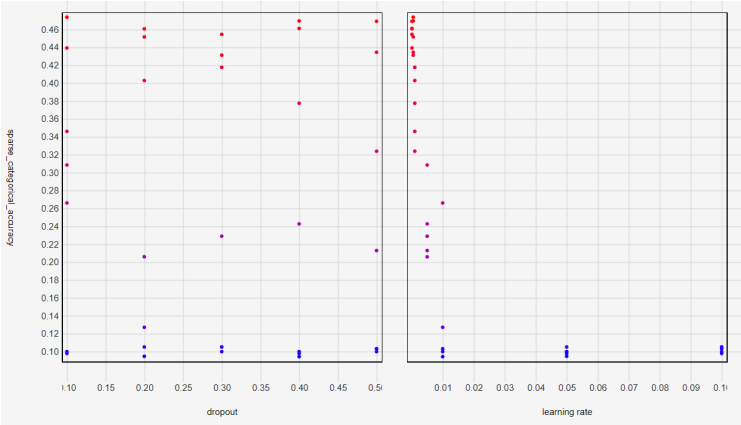


Figure B.4. Hyperparameter tuning for BRNet-L.