MASTER'S THESIS 2020

Quantization Profiler for Artificial Neural Networks

Martin Lindström, Jakob Hök

ISSN 1650-2884
LU-CS-EX: 2020-20

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2020-20

Quantization Profiler for Artificial Neural Networks

Martin Lindström, Jakob Hök

Quantization Profiler for Artificial Neural Networks

Martin Lindström
dat15mli@student.lu.se

Jakob Hök
dat15jh1@student.lu.se

June 18, 2020

Master's thesis work carried out at ARM Sweden AB.

Supervisors: Jörn Janneck, jorn.janneck@cs.lth.se (LTH)
Axel Berg, axel.berg@arm.com (ARM Sweden AB)
Kevin Wohnrade, kevin.wohnrade@arm.com (ARM Sweden AB)

Examiner: Flavius Gruian, flavius.gruian@cs.lth.se

Abstract

We develop a software framework that is able to modify implementations of operators within any artificial neural network (ANN). The framework is able to import a trained *TensorFlow* model and target a subset of its network layers, to provide them with custom operator implementations. Furthermore, the framework uses *signal-to-quantization-noise ratio* (SQNR) as a metric to identify potential layer implementations that are bottlenecks for prediction accuracy. With the use of the framework, we test various custom operator implementations for the *MobileNetV2* neural-network architecture, which was developed by researchers Google. Specifically, we carry out experiments that benchmark operators that are well adapted for low memory usage and execution time, e.g. 8-bit quantization, but have a potential cost in prediction accuracy. With our results, we prove that this tool can be useful for industries where running ANNs on devices with limited hardware, like mobile phones, are of interest.

Keywords: quantization, inference, MobileNet, SQNR, artificial neural network, convolution, TensorFlow

Acknowledgements

We would like to thank Arm Sweden AB for giving us such a great opportunity with this thesis and supplying us with resources. In particular, big thanks to Axel Berg and Kevin Wohnrade who have guided and helped us from start to finish. Finally, special thanks to Jörn Janneck at LTH who provided continuous and valuable feedback.

Contents

Chapter 1
Introduction

The ongoing research of *Artificial Neural Networks* (ANNs) is proceeding rapidly. Researchers, from e.g. *Google*, are publishing multiple papers each year regarding both *network architectures* and optimizations for speedup and accuracy for ANNs.

Today, there is a high demand for ANN applications within personal mobile devices, performing tasks such as image classification and natural language processing. Even though today's hardware is not usually a limiting factor for software projects in general, any applicable ANN model is usually an exception because it is large in both memory footprint and in the amount of computations. Consequently, there is a demand for optimizing ANN software in both computation speed and memory in order to be able to run *inference* (prediction) on limited hardware. This is the central motive behind this Master's Thesis project and why it is carried out under supervision of the company Arm, which is a supplier of microprocessor technology [2].

A common method to make ANN models feasible within mobile devices is to make use of *quantization* schemes to make the trained model both faster to execute and smaller in memory footprint. Often, an ANN model is first trained on a powerful machine with 32-bit floating-point precision; the trained model is then converted to the *quantized* format and deployed to a device; this is called *post training quantization* [11]. The goal is therefore to do as much computation and optimization *offline* as possible before freezing a model into its fully-trained state and deploying it. The deployed model can then, for example, work with *integer*-only arithmetic to perform inference.

Ultimately, optimizing an ANN model for speed, size, energy consumption or whichever relevant metric, will most likely result in an accuracy drop. After all, accuracy is an important matter when running inference for an ANN; hence, it is not hard to motivate an investigation of how a model's accuracy changes with implementation modifications. A *software framework* tailored for investigations like this does currently, to our knowledge, not exist; the purpose of the thesis is to develop such a framework from the ground up to fill this space; we have named the framework: *Quantization Profiler for Artificial Neural Networks* (QPANN).

A modern and widely used API for training, testing and deploying ANNs is *TensorFlow*,

which was first released as an *open-source* project in *November 2015* [1]; it has since then been regularly maintained and is still under development. *TensorFlow* has a software stack called *TensorFlow Lite* which is used for the creation and deployment of quantized models on limited hardware. Naturally, when such models are exported, there will be a cost in terms of prediction performance on the resulting neural network. Since *TensorFlow Lite* converts an entire ANN model to a quantized 8-bit counterpart, one can not choose which layers to quantize or specify the bit width. QPANN is therefore developed in order to help any user profile the consequences of quantization.

QPANN enables its user to select specific layers, of a pretrained image-classifying network, and quantize them to *8-bit integer*, following *TensorFlow Lite's* quantization scheme [6], while the rest of the layers remain in their default *32-bit floating-point* format, i.e., creating a *partially quantized* ANN. QPANN thus opens up the possibility to easily evaluate the layers of a neural network individually, in terms of how much they contribute to the overall accuracy loss. If there are layers that cause more accuracy drop than others, they can be identified by the user.

One can not only modify the underlying data type, but also analyze how SQNR (*Signal-to-quantization-noise*, covered in Section 2.4) may affect accuracy, or how rounding of multiply-accumulate operations might change accuracy.

The framework will simultaneously run inference on a model with floating-point precision, a fully quantized *TensorFlow-Lite* model and a partially quantized model, i.e. the user-specified model. The accuracy for the three models can then be directly compared to one another. With repeated iterations of different setups of the partially quantized network, the user can learn where quantization can be done more freely and where the model should remain in a higher precision mode. The framework will export accuracy for the models, together with SQNR metrics.

The thesis is divided into four parts. Chapter 2 explains relevant theory regarding ANN and quantization. Chapter 3 goes through the methodology for the project. Chapter 4 describes the software framework developed in the thesis. Chapter 5 evaluates the framework and show potential use cases. Finally, in Chapter 6, we conclude the evaluation and discuss future work.

1.1 Division of Work

The labor of this thesis, and how it was distributed between the two authors, is presented in this section. The abbreviations **ML** and **JH** refers to the authors, *Martin Lindström* and *Jakob Hök* respectively.

The first weeks of work was put into getting a better understanding of how quantization works and how it is applied within neural networks. This means both to study related work within the field, and to understand how *TensorFlow Lite* implements quantization for its models. This was done by both the authors because it is necessary to understand these central aspects within the work.

While still researching related work, the authors proceeded to getting a better understanding of the functionality of *TensorFlow* and which parts of it can be used to help the implementation of the framework. Specifically, **JH** was investigating how one can implement their own custom operator in *TensorFlow*, and implemented a test operator in C++. At the

same time, **ML** was researching how quantization/dequantization conversion works and how one can compute simple operations, like a scalar product between two quantized vectors.

After this was complete, the authors felt confident enough to start the actual implementing the framework. **ML** implemented the first version of a quantized convolution operator and wrote thorough unit tests for it. **JH** developed the base of the actual framework itself in *Python*. At this point, the framework could run inference on a model that made all convolutional layers (Section 2.1.1) by using **ML**'s quantized implementation.

JH continued to extend the framework by letting the user select which layers to replace for a quantized model. **ML** made different implementations for the convolutional operator so that the framework can run different types of convolutions for every layer. More unit tests were written for both the operator and the framework.

ML and **JH** went over to finish the remaining quantized operators (explained in Section 4.3) in C++ together. The remaining work regarding development were smaller tasks that were taken on by whomever was unoccupied at the moment.

The distribution of the writing the chapters and sections of the thesis is shown in Table 1.1. In the leftmost column, a single number refers to a chapter's text before any section begins.

Table 1.1: Division of work for the thesis' sections.

Section	**ML**	**JH**
1	X	X
1.1	X	
1.2		X
1.3		X
2.1	X	X
2.2	X	
2.3	X	X
2.4		X
3	X	
3.1	X	
3.2	X	
4	X	
4.1	X	
4.2	X	
4.3	X	X
4.4	X	
4.5	X	
5.1	X	X
5.2	X	X
6.1	X	X
6.2	X	X

1.2 Research Questions

The introduction motivates why it is interesting to be able to study the effect of quantization, not only on the entire ANN but also on layers individually. More specifically, we will study the network model *MobileNetV2* (covered in Section 2.2). The thesis will investigate the following research questions, mentioned in no particular order:

- Can we build a framework that is able to partially quantize a network to *8-bit integer*?
- Where is *MobileNetV2* most sensitive to quantization; in the beginning or in the end?
- In *MobileNetV2*, are there one or multiple layers that may contribute more to the accuracy loss when quantized?
- In *MobileNetV2*, can SQNR be used to detect the most and the least sensitive layers to quantization, in terms of inference accuracy?

1.3 Related Work

In papers published by Google, there have been research on quantization's impact on accuracy [10] [11]. In paper [11] the author investigates different quantization schemes and bit widths. In paper [10] the researchers explain how quantized layers can be optimized during inference, such as optimized "integer-arithmetic-only matrix multiplication" (will be covered in Section 2.3.2) and "batch-normalization folding" (covered in Section 2.3.5). Both of these papers are tangled with *TensorFlow*, where one describe how quantization is done in *TensorFlow* ([10]), more precisely in *TensorFlow Lite*; and the other use the framework for experiments ([11]). None of these papers look at individual layers in a network model, they study quantization of entire models.

Similar to *TensorFlow*, an open-source machine-learning library called *PyTorch*, developed primarily by AI researchers at *FaceBook* [14], has begun developing a quantization API which currently is experimental [3]. Their API supports ANN models that "perform all or part of the computation in lower precision". The API is, as mentioned, experimental and only allows the developer to change parameters in limited ways; alas, the specific- and customizable parameters needed in the thesis would still need to be implemented, whether it being in *TensorFlow* or in *PyTorch*. Hence, *PyTorch* will not be used in the thesis.

As a metric to quantify effect of quantization, SQNR (covered in Section 2.4) looks promising. A paper that investigated the effect of SQNR for ANNs, showed how one can optimize the bit widths of layers using fixed-point implementation [12]. Even though they optimize per layer, it does not generalize well when one is limited to a set of bit widths, or for example want to change specific layers' implementations. However, the paper gives a sound indication that SQNR is a viable metric; which is why we have chosen to work with SQNR.

Chapter 2
Background

2.1 Artificial Neural Networks (ANN)

Deep Learning is a field that has been researched thoroughly during the latest decade. The term "*deep*" comes from the fact that input data is propagated through a network, which is called ANN, that consists of multiple *hidden layers* [5]. The name comes from the fact that it is a model inspired from the human brain. Layers in an ANN are in practice functions that transform their input data to outputs that are based on what the network has learned during its *training stage*. These layers are trained on data samples and compute their outputs based on that; together they form the ANN and the pipeline that the input data propagates through. With this in mind, see Figure 2.2, where a set of images are taken through a *trained neural network*, starting with a convolutional layer and ending with a final, classifying layer. After that, the neural network outputs its prediction; in the case of the example, it is an array that contains the probability of the image belonging to each respective class that the network has been trained to recognize. In this example, the input to the classifying network is an image of a dog. The output is an array that contains 0.9 at the index for the class dog, with the rest of the 0.1 probability being spread out throughout the rest of the indices; this example is shown in Figure 2.1.

Dog	Cat	Pig	Cow	Horse
0.90	0.02	0.03	0.01	0.04

Figure 2.1: Output array from a simple classifier that is 90% certain that the input image shows a dog.

There are many architectures and application areas of ANNs that stretch beyond the field of classifying images; however, in this work, the scope will be limited to only cover *image-classifying* networks.

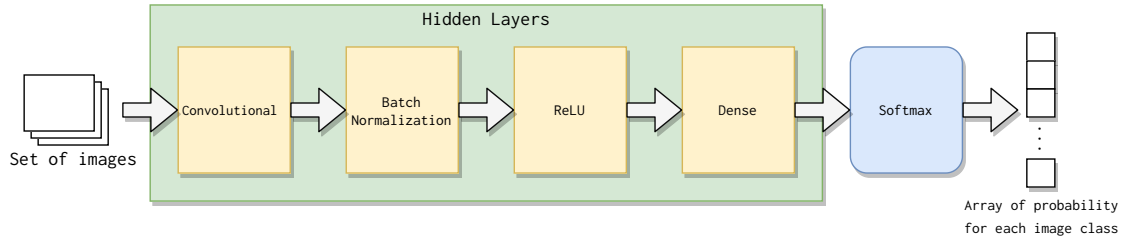


Figure 2.2: A simple example ANN with a few hidden layers that classifies images.

Training ANNs is an area that is central to the performance of the network and is carried out by defining a *cost function* and applying, for example, a method called *gradient descent* on the *weights* of the network [5]. The values of the *weights* within the hidden layers are what represent a training state for the network. This thesis is carried out with ANNs with *pretrained weights*, i.e., the assumption that a network has been sufficiently trained in *floating point* format will be made. Therefore, no theory regarding the training will be covered; instead focus will be held on the *forward propagation* of data, the process of classifying images.

2.1.1 Network Layers

Layers are what makes the ANN in its entirety; they are entities that operate on the *output activations* of their predecessors and computes their own *output activations*. The term *activations* refers to the temporary data that is passed between the layers.

For this work, a set of *neural-network* layers are included; specifically, the layers are taken from what is included in the network architecture of *MobileNetV2* [15], which will later be covered in Section 2.2. Each of these layers will be covered in this section individually by explaining their mathematical definitions and computational costs; they have all been implemented or to some extent been dealt with during the implementation of the software framework.

Fully Connected

The most fundamental layer is the fully-connected layer, where each of the n nodes in layer h have a connection to the m nodes of the following layer g. In other words, a node x_k^g in layer g will receive an input from each n nodes $x_i^h, i \in \{1 \dots n\}$. Every connection is associated with a weight w_{ik}^h, i.e., a factor of how much impact a node will have on x_k^g. This can be expressed as a scalar product between the column vectors $\mathbf{x}^h$ and $\mathbf{w}_k^h$, as shown in Equation (2.1). The fully connected layer is visualized in Figure 2.3.

$$x_k^g = (\mathbf{x}^h)^T \mathbf{w}_k^h = \sum_{i=1}^{n} x_i^h w_{ik}^h \tag{2.1}$$

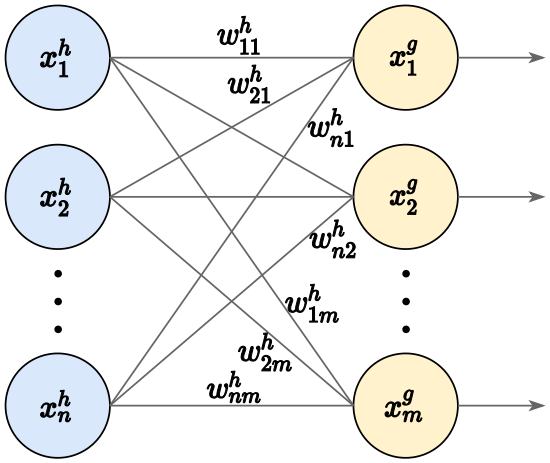


Figure 2.3: A fully-connected layer that takes its input $\mathbf{x}^h$ to perform matrix multiplication with its weight matrix $\mathbf{W}$, to compute the output $\mathbf{x}^g$

With $\mathbf{x}^g$ as a column vector and $\mathbf{W}$ as an $n \times m$ matrix, which represent input activations and weights respectively; the expression can be compactly expressed as:

$$\begin{bmatrix} x_1^g \\ x_2^g \\ \vdots \\ x_{m-1}^g \\ x_m^g \end{bmatrix} = \begin{bmatrix} w_{11}^h & w_{12}^h & \dots & w_{1m}^h \\ w_{21}^h & w_{22}^h & \dots & w_{2m}^h \\ \vdots & \vdots & \ddots & \vdots \\ w_{n1}^h & w_{n2}^h & \dots & w_{nm}^h \end{bmatrix}^T \cdot \begin{bmatrix} x_1^h \\ x_2^h \\ \vdots \\ x_{n-1}^h \\ x_n^h \end{bmatrix} \tag{2.2}$$

$$\mathbf{x}^g = \begin{bmatrix} | & | & & | & | \\ \mathbf{w}_1^h & \mathbf{w}_2^h & \dots & \mathbf{w}_{m-1}^h & \mathbf{w}_m^h \\ | & | & & | & |\cdot \end{bmatrix}^T \mathbf{x}^h$$

$$\mathbf{x}^g = \mathbf{W}^T \cdot \mathbf{x}^h$$

Given the Equations 2.1 and 2.2 one can easily calculate the computational cost in terms of number of operations. For each output x_k^g there are n multiplications and $n-1$ additions, hence the computational cost is: $m*(n+(n-1)) = m*(2n-1)$. The number of parameters to store is simply the weight matrix $\mathbf{W}$, which is: $m * n$.

Convolution

One can take advantage of the fact that an input is an image and use the *convolutional* layer. Pixels in close proximity are tightly coupled, and the pixels further away are less important. Thus, given a pixel p_{ij} on ith row and jth column, the pixels surrounding it (i.e. $\{p_{kl} : k \in [i-D, i+D], l \in [j-D, j+D]\}$ for some constant D) are only relevant for the corresponding node in the next layer. This can be interpreted as setting the rest of the input pixels' weights to 0, i.e. a *sparse* fully-connected layer.

In image processing this is more commonly called a *convolution* (or filtering) with a kernel of dimensions $K \times K$ where $K = 2D + 1$, hence the name *convolution layer*. Assuming a gray-scale image and ignoring color channels for now, the result X^g from a single convolution

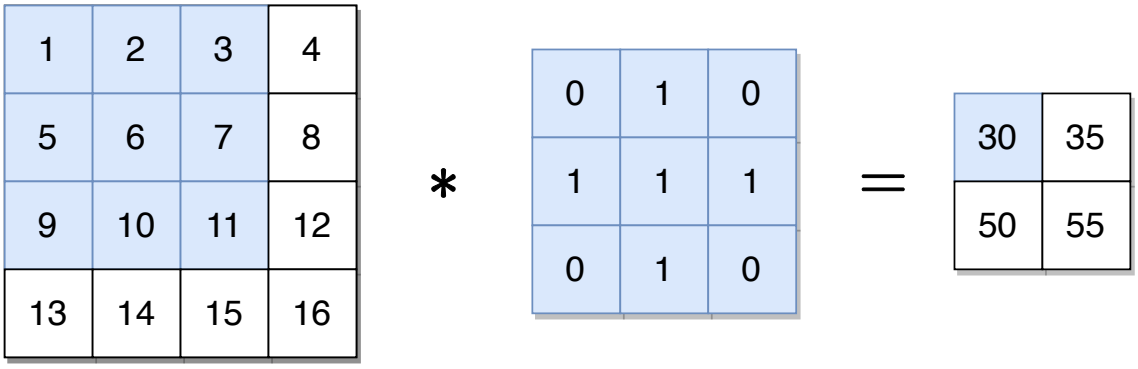


Figure 2.4: Demonstrating a simple two-dimensional convolution, with $N = 4$ and $K = 3$. On the left-hand side is the input image, in the middle is the kernel and on the right-hand side is the output image.

on image X^h can be expressed as:

$$X^g_{k,l} = \sum_{i=1}^{K}\sum_{j=1}^{K} W_{i,j} \cdot X^h_{(k+i-1),(l+j-1)} \tag{2.3}$$

where k is the row index and l is the column index. The single convolution in Equation 2.3 can be seen in Figure 2.4. If $K = 1$ then the output from the convolution will have the same dimensions as the input image, the convolution will simply be an affine mapping. However, if $K > 1$ then the resulting dimensions will be reduced. The relationship between input size N, kernel size K and output dimension D can be expressed as:

$$D = N - (K - 1) \tag{2.4}$$

Equation 2.4 assumes that the stride of the kernel is $S = 1$ and that there is no padding. Stride refers to the size of the steps that the kernel takes between each output index, whereas padding means to extend the input matrix with, e.g. zeros, around its border. Taking stride S and padding P into account, we can rewrite Equation 2.4 into:

$$D = 1 + \frac{N - K + 2P}{S} \tag{2.5}$$

As an example, with $N = 3$, $K = 3$, $P = 1$ and $S = 2$ the output dimension will be $D = 1 + \frac{3-3+2\cdot1}{2} = 1 + \frac{2}{2} = 1 + 1 = 2$, see Figure 2.5. The correctness of Equation 2.5 is left as an exercise for the reader.

Introducing color channels, e.g., red, green and blue, the convolution has a third dimension, a depth. The only difference is that the kernels now have to span across the channels and thus have $K \times K \times C_{in}$ dimensions instead of the two dimensional $K \times K$. Note, the output image's dimension is not affected. However, it may be desired to have multiple output channels as well, this is solved by having one $K \times K \times C_{in}$ kernel per output channel, see Figure 2.6. Now Equation 2.3 is modified with the inclusion of input- and output channels:

$$X^g_{k,l,m} = \sum_{c=1}^{C_{in}}\sum_{i=1}^{K}\sum_{j=1}^{K} W_{m,i,j,c} \cdot X^h_{(k+i-1),(l+j-1),c}$$

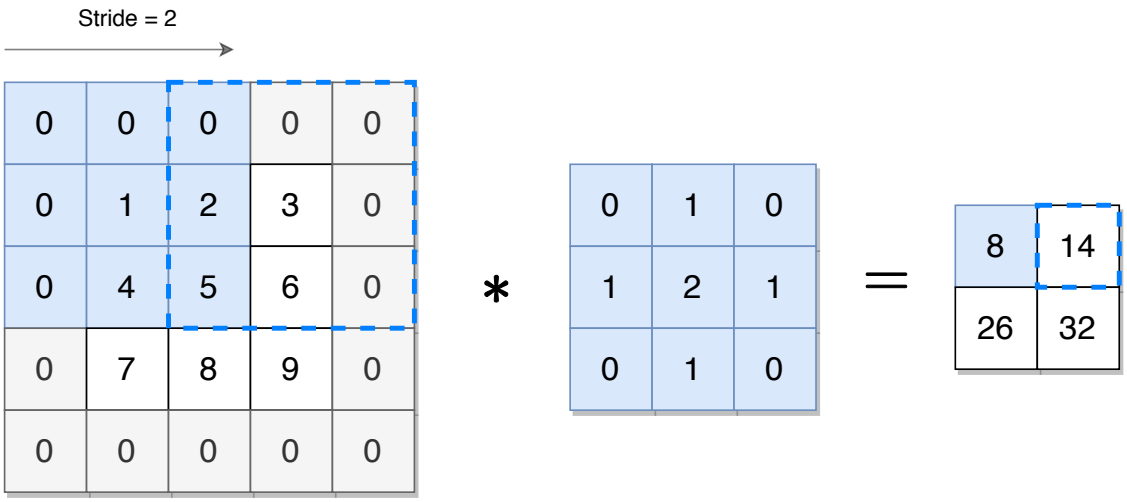


Figure 2.5: Two-dimensional convolution displaying the additional parameters *stride* and *padding*. The parameters in the example are: $N = 3$, $K = 3$, $P = 1$ and $S = 2$.

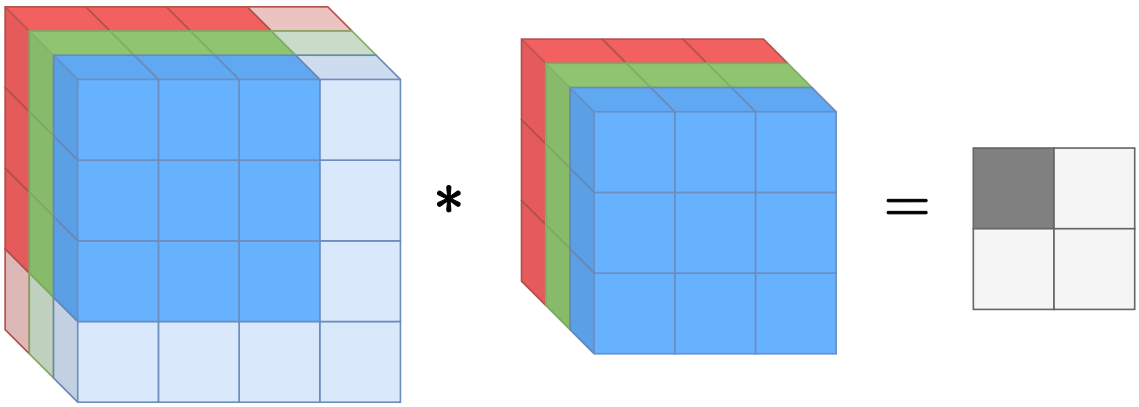


Figure 2.6: Convolution with three input channels; red, green and blue. Since there there is only one kernel, the output has only one output channel.

where m is the mth output channel.

To calculate the computational cost, consider that the convolution layer is a fully-connected layer with an input image of dimensions $N \times N \times C_{in}$ and an output image of dimensions $M \times M \times C_{out}$:

$$\begin{aligned} Cost_{\text{fully connected}} &= M \cdot M \cdot C_{out} \cdot (2 \cdot N \cdot N \cdot C_{in} - 1) \\ &= M^2 \cdot C_{out} \cdot (2 \cdot N^2 \cdot C_{in} - 1) \end{aligned}$$

Now, the fully-connected equation assumes there is a connection (non-zero weight) from every input to every output. In the convolution case, however, the amount of connections is dependant on the size of a kernel, which yields the following cost:

$$Cost_{\text{convolution}} = M^2 \cdot C_{out} \cdot (2 \cdot K^2 \cdot C_{in} - 1) \tag{2.6}$$

In terms of weights, there are C_{out} kernels of dimension $K \times K \times C_{in}$; hence the number of parameters stored is: $C_{out} \cdot K^2 \cdot C_{in}$.

If $K \ll N$, the cost will be greatly reduced as well as the memory footprint, since the number of weights that needs to be stored and loaded is decreased.

Depthwise Separable Convolution

Even with the addition of convolution layer, there is still room for improvement in terms of cost and memory footprint. Researchers at Google have showcased in their paper [8] a more efficient convolution, *Depthwise Separable Convolution*, which splits the conventional convolution into two separate operations. More specifically, the convolution is factorized into a depthwise convolution and a 1×1 convolution called pointwise convolution.

The idea behind the factorization is that standard convolution both filters *and* combines the input image in a single operation, whereas in the depthwise separable convolution, the depthwise convolution filters the input, and the pointwise convolution combines the result after the filtering.

Depthwise convolution uses a single two-dimensional kernel for each input channel, i.e. filters an $N \times N \times C_{in}$ image using a $K \times K$ kernel, to an intermediate image with size $M \times M \times C_{in}$. The pointwise convolution then combines the intermediate image using C_{out} kernels with size $1 \times 1 \times C_{in}$, see figure 2.7. The output image thus has the size $M \times M \times C_{out}$ which is the same as using standard convolution with C_{out} three-dimensional kernels of size $K \times K \times C_{in}$.

Starting with the depthwise convolution, the computational cost is:

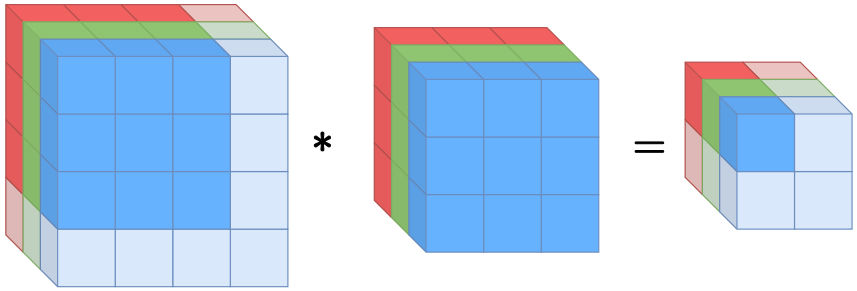
$$Cost_{\text{depthwise convolution}} = M^2 \cdot C_{in} \cdot (2 \cdot K^2 - 1)$$

Very similar to Equation 2.6 but with a kernel of only two dimensions. The pointwise convolution has the computational cost of:

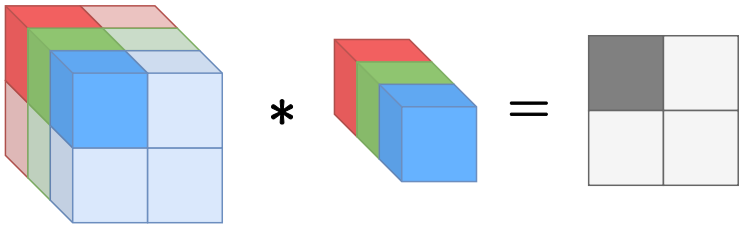
$$\begin{aligned} Cost_{\text{pointwise convolution}} &= M^2 \cdot C_{out} \cdot (2 \cdot 1 \cdot 1 \cdot C_{in} - 1) \\ &= M^2 \cdot C_{out} \cdot (2 \cdot C_{in} - 1) \end{aligned}$$

Finally, adding the cost of depthwise convolution and pointwise convolution, the total cost then becomes:

$$\begin{aligned} Cost_{\text{depthwise separable convolution}} &= Cost_{\text{depthwise convolution}} + Cost_{\text{pointwise convolution}} \\ &= M^2 \cdot C_{in} \cdot (2 \cdot K^2 - 1) + M^2 \cdot C_{out} \cdot (2 \cdot C_{in} - 1) \\ &= M^2 \cdot (C_{in} \cdot (2 \cdot K^2 - 1) + C_{out} \cdot (2 \cdot C_{in} - 1)) \end{aligned}$$



(a) Depthwise Convolution



(b) 1×1 convolution called pointwise convolution

Figure 2.7: The two operations of a depthwise separable convolution. First a depthwise convolution (a), followed by a 1×1 convolution (b).

In depthwise convolution, there are C_{in} kernels of dimension $K \times K$ and in pointwise convolution there are C_{out} kernels of dimension $1 \times 1 \times C_{in}$; thus the number of parameters stored is: $C_{in} \cdot K^2 + C_{out} \cdot C_{in} = C_{in} \cdot (K^2 + C_{out})$.

According to [8], with 3×3 kernels the amount of computations are reduced by a factor between 8 and 9, and since there are less weights to store, the model size is reduced. The downside is instead slightly worse accuracy, by the magnitude of 1%, which is impressive seeing the advantages.

Global Average Pooling

Global Average Pooling takes as input a three dimensional tensor, with height H, width W and number of channels C, and outputs the average of each channel. Thus the result is an array with C elements, see figure 2.8. When global average pooling was first introduced, as a replacement of the fully-connected layer at the end of the network, the results showed state-of-the-art performance [13].

Flattening the input we get N nodes, where $N = H \cdot W \cdot C$, and as output we get M nodes, where $M = C$. The computational cost of a fully-connected layer is, as mentioned in Section 2.1.1, $Cost_{\text{FC}} = M \cdot (2N - 1)$ whereas global average pooling will have $H \cdot W - 1$ additions and one division per output, resulting in a computational cost of: $Cost_{\text{GAP}} = (H \cdot W - 1 + 1) \cdot M = H \cdot W \cdot M = N$. As one can see, there is a cost reduction of about $2M$ and also no weights to store, thus removing the memory footprint $(N \cdot M)$ needed of a fully-connected layer.

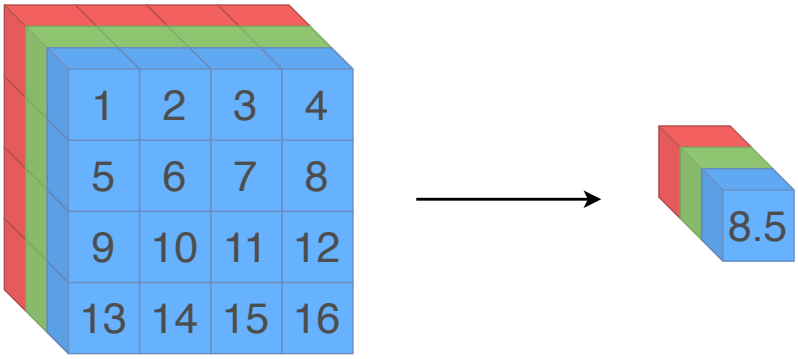


Figure 2.8: Global average pooling with three channels. The result is the mean for every channel.

Batch Normalization

When a neural network is trained, the data is often given to the network in so called *batches*, which simply means multiple samples are propagated through the network simultaneously. Batches are used because the network can adjust its trainable parameters according to data that is a better estimate of the entire training dataset, rather than just propagating one sample one at a time [9].

Since the values of samples and trainable weights change with each training iteration, the inputs to each layer changes distribution during training. This property causes long training times because of the amount of training iterations it requires; it is called *covariate shift*. To combat the long training times, researchers at Google came up with a solution by introducing a normalization layer that normalizes inputs for each *mini-batch* (small *batch*). The layer is called *Batch Normalization* and can be inserted after each layer [9].

The idea is to ensure the layers' inputs have distributions of zero means and unit variances, which in turn reduces the internal covariant shift. Thus, given an mini-batch B of size n with inputs $x_1, x_2, \ldots, x_n$ to the *batch normalization* layer, the output $\hat{x}_i$ is given as follows:

$$\hat{x}_i = \frac{x_i - \mu_B}{\sigma_B} \quad \text{where:} \quad \mu_B = \frac{1}{n}\sum_{i=1}^{n} x_i \,, \quad \sigma_B^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu_B)^2 \tag{2.7}$$

The *batch normalization* layer also adds two trainable parameters, γ and β, which linearly transforms $\hat{x}_i \rightarrow y_i$:

$$y_i = \gamma \cdot \hat{x}_i + \beta \tag{2.8}$$

and combining Equation (2.7) and (2.8):

$$\begin{aligned} y_i &= \gamma \cdot \hat{x}_i + \beta \\ &= \gamma \cdot \frac{x_i - \mu_B}{\sigma_B} + \beta \end{aligned} \tag{2.9}$$

The purpose of γ and β is to ensure that the *batch normalization* layer can produce the same output as input (if $\gamma = \sigma_B$ and $\beta = \mu_B$) which preserves the network capacity.

During inference the mean and variance will be fixed, thus the computation cost of calculating the mean and variance can be ignored. Given A activations, i.e. number of inputs to be normalized, for every input a_i where $i \in \{0, 1 \ldots A\}$ the number of arithmetic operations are 4 (disregarding the inefficiency of division for simplicity). Hence the total cost is: $4A$.

Following a convolutional layer, *batch normalization* will apply Equation 2.9 *per channel* meaning there will be batch-normalization parameters for every channel of the input. Thus, given C_{in} input channels; the number of parameters stored is: $4C_{in}$. Following a fully-connected layer, one can view it as having one channel.

ReLU

The *Rectified Linear Unit* (ReLU) is a function that applies a ramp function to its argument. ReLU is expressed mathematically as

$$ReLU(x) = max(0, x)$$

An extension of this function is the ReLU6, which sets an upper bound of the output as well (Equation (2.13) contains the clamp function):

$$ReLU6(x) = clamp(x, 0, 6)$$

This function is often called on the output from a convolutional layer; *MobileNet* makes use of this for a number of their convolutional layers [8]. In Section 2.3, it will be made clear why the ReLU6 function is well suited for quantized ANNs.

2.1.2 Bias Addition

A common method to use after a layer has computed its output, is to add every output activation with a constant value that has been determined during training. For a fully-connected layer, there is a unique bias for every output activation scalar, i.e., the output is added with the bias vector which is $\mathbf{b}$ in the equation below:

$$\mathbf{x}^g = \mathbf{W}^T \cdot \mathbf{x}^h + \mathbf{b}$$

For the specific case of *convolutional-* and *depthwise convolutional* layers, the bias is instead added per output channel, i.e., every output channel share a bias term.

The computational cost is simply one add operation per output. In terms of memory, for a fully-connected layer there are m parameters to be stored (one for every output); and for convolutional- and depthwise convolutional layers there are C_{out} parameters to be stored.

2.1.3 Top-N Accuracy

Often, a classifier is measured for how likely it is to predict the correct label, or rather, how often the correct label is included within the top N predictions from the classifier. An array that contains certainties (probabilities) for every label is computed from the classifier. The *top-N accuracy* is simply the following: Given a dataset of samples, how many times are the correct label included in the N topmost predictions if they are sorted by certainty, in descending order. This number is divided by the size of the dataset to show it as a percentage.

2.2 MobileNet and ImageNet

Even though QPANN is intended to function on any classifier, the scope of this project has been set to only experiment on *MobileNetV2* [15], trained on a large image dataset from the database *ImageNet* [4].

MobileNetV2 is a *neural network* architecture that was developed by researchers at *Google* and presented in a paper in 2019. The intension of *MobileNet* is to make it feasible to run inference on mobile devices or hardware with limited computational resources in general. The architecture contains layers that are a compromise to the abilities of the hardware, such as using *depthwise separable convolutions* instead of ordinary *convolutional layers* in order to reduce the amount of total computation.

To further reduce memory footprint and execution time, a possible solution is to *quantize* all *weights* together with all the activations and operations within the ANN. The motivation for choosing *MobileNetV2* is that it is a modern architecture that is already aimed towards making accuracy trade-offs in its structure.

As mentioned, the *MobileNetV2* implementation used in this project was trained on a large dataset taken from *ImageNet*, which is a database containing labeled images. This motive behind *ImageNet* is to provide data to relevant application areas, such as training machine-learning models [4]. For the experiments carried out while testing and developing QPANN, *ImageNet* has been used as the *only* source.

2.3 Quantization Design

This section will cover background theory related to quantization in order to familiarize with the concept as well as to understand the work that has been done in the thesis. First *uniform quantization* will be explained, together with how *quantized matrix multiplication* can be made more efficiently. Then the difference between *per-layer quantization* and *per-channel quantization* will be described. Then an optimization technique called *folding* will be explained. Finally, the section will explain the main metric, in the thesis, to measure quantization error called *SQNR*.

2.3.1 Uniform Quantization

Consider a floating-point based interval (f_{min}, f_{max}) which is required to contain the number 0. The uniform quantizer maps a floating-point number to an *signed/unsigned* integer of a desired *bit width*. Two parameters, scale Δ and zero-point Z is computed according to Equations (2.10) and (2.11) to finally construct the parameterized quantizer $Quant(x)$, described in Equation (2.14). This section will derive this construction and explain its steps.

Δ is a floating-point number and represents the distance in the floating point domain between each quantized step; therefore, to represent wide intervals, the Δ will become large and reduce precision for computations. The type of Z is the same as the quantized result and is equal to the quantized value where the floating point counterpart is equal to zero. As an additional note, Section 4.2.1 explains how to retrieve f_{min} and f_{max} for every activation

and weight within a neural network.

$$\Delta = \frac{f_{max} - f_{min}}{2^{bit\ width} - 1} \tag{2.10}$$

$$Z = round\Big(- \frac{f_{min}}{\Delta}\Big) \tag{2.11}$$

The quantization scheme can equivalently be applied for both unsigned and signed formats; they only differ in the *clamp*-ing part of the quantization formula. The advantage of signed quantization, is that Z is 0 for symmetric quantization, rather than 128 which is the case for the unsigned format [6]; $Z = 0$ means that, in practice, the addition in Equation (2.12) can be omitted.

After these quantization parameters have been found, a *real* number can be quantized to a *integer* number with the following formula:

$$Q(x) = round\Big(\frac{x}{\Delta}\Big) + Z \tag{2.12}$$

However, the *integer* needs to fit within the range given by the *bit width* of the quantized format. If the operand $x \notin (f_{min}, f_{max})$; then the quantized output lies outside this range, it needs to be moved into the interval with the *clamp* function below:

$$clamp(x, a, b) = \begin{cases} a, & \text{if } x < a \\ b, & \text{if } x > b \\ x, & \text{otherwise} \end{cases} \tag{2.13}$$

With these parameters and functions, the final quantization function can be expressed:

$$Quant(x) = \begin{cases} clamp(Q(x),\ 0,\ 2^{bit\_width} - 1), & \text{for unsigned} \\ clamp(Q(x),\ -2^{bit\_width-1},\ 2^{bit\_width-1} - 1), & \text{for signed} \end{cases} \tag{2.14}$$

To dequantize a number back to *floating point*:

$$Dequant(q) = \Delta(q - Z) \tag{2.15}$$

2.3.2 Quantized Matrix Multiplication

With the quantization scheme from Section (2.3.1) the formula for quantized matrix multiplication can be derived. Equation (2.16) shows a multiplication matrices A and B:

$$C_{ij} = \sum_{k=1}^{N} A_{ik} B_{kj} \tag{2.16}$$

The quantized formula is derived from Equation (2.15) and is shown below in Equation (2.17). The quantization parameters Δ and Z are defined per matrix and not per matrix element; therefore, a reasonable selection to retrieve Δ_A and Z_A would be to substitute in $f_{min} = min(0, min(A))$ and $f_{max} = max(0, max(A))$ into Equations (2.10) and (2.11). Then, the value zero, from the non-quantized format is always perfectly representable in the

quantized environment; this is similarly done in the paper [11], where the author has chosen to use the method of selecting quantization parameters from the minimum- and maximum values of weight tensors.

$$\Delta_C(C_{ij}^q - Z_C) = \sum_{k=1}^{N} \Delta_A(A_{ik}^q - Z_A)\Delta_B(B_{kj}^q - Z_B) \tag{2.17}$$

By isolating C_{ij}^q, as Equation (2.18), a method evaluating matrix multiplications for quantized matrices and their respective parameters has been found.

$$C_{ij}^q = Z_C + M\sum_{k=1}^{N}(A_{ik}^q - Z_A)(B_{kj}^q - Z_B), \text{ where } M = \frac{\Delta_A\Delta_B}{\Delta_C} \tag{2.18}$$

There is more to the formula however. Firstly, the factor M is a real value and has to be explicitly dealt with. The paper [10] states that this number has empirically been found to always be between 0 and 1 and it can be expressed in a *fixed-point* format to perform the multiplication with the sum. In practice, this would mean to perform one integer multiplication followed by a bit shift operation, which makes the this data-type conversion desirable.

Secondly, the sum can be separated into four parts, as Equation (2.19) displays, in order to optimize the computation in terms of speed.

$$C_{ij}^q = Z_C + M\left(\sum_{k=1}^{N} A_{ik}^q B_{kj}^q - Z_A\sum_{k=1}^{N} B_{kj}^q - Z_B\sum_{k=1}^{N} A_{ik}^q + NZ_AZ_B\right) \tag{2.19}$$

NZ_AZ_B can be computed *offline* (before inference) so it becomes a constant. The first sum is a scalar product and has to be computed for every index (i, j). The two following sums can be reused between rows and columns so they do not need to be evaluated for every index. Therefore, the sums of rows and columns leads to $O(N^2)$, whereas the bottleneck of the computation is the scalar product with $O(N^3)$ in time complexity for full matrices.

Note that the computations in the parenthesis of Equation (2.19) are all *signed*-integer arithmetic but are likely to overflow within the specified *bit width*. *TensorFlow Lite* has chosen to accumulate the products with the sums in a 32-bit *signed* integer while keeping all operands in a integer 8-bit format [10].

This computation algorithm was described to be used for matrix multiplications; naturally, it can be used for convolutions as well since every output index is practically a scalar product of two operands. The final part of the algorithm is to add the *bias* term *b*, if such one exists, which is defined to have $\Delta_b = \Delta_A\Delta_B$ and $Z_b = 0$. With these constraints, the bias term can be included in the parenthesis of Equation (2.19), such that:

$$C_{ij}^q = Z_C + M\left(\sum_{k=1}^{N} A_{ik}^q B_{kj}^q - Z_A\sum_{k=1}^{N} B_{kj}^q - Z_B\sum_{k=1}^{N} A_{ik}^q + NZ_AZ_B + b^q\right)$$

TensorFlow Lite uses 32-bit *signed* integer numbers rather than 8-bit for *bias* terms [6]. The reason for this, as [10] explains it, is that if 8-bit biases are used, there will be an overall biased error which continues to propagate through the network. That is, all nodes which the 8-bit bias are added onto, will receive the same biased error with a non-zero mean. If instead a

32-bit format is used, this error will become much smaller and is consequently a must for achieving accuracy close to a network's floating-point counterpart. Since the scalar-product also uses 32-bit *signed* integer, the bias addition becomes simple.

To summarize, the quantized inference at a convolution layer can be described in four steps:

1. Multiply-accumulate operation of input activation and weights:

```
int32 += int8 * int8
```

2. Add the bias term with the result from multiply-accumulate operation:

```
int32 = int32 + int32
```

3. Perform the *fixed-point* multiplication with the factor M to rescale the accumulator:

```
int32 = rescale(int32)
```

4. Saturate the resulting 32-bit integer down to signed 8-bit integer which is the activation in the next layer:

```
int8 = saturate(int32)
```

2.3.3 Per-Layer and Per-Channel Quantization

Section 2.3.2 discusses how a quantized matrix multiplication can be performed if the operands have their own separate ranges (*per-layer quantization*). An extension of this that fits the domain specific properties of a *convolutional layer* can be used to improve prediction accuracy. This extension is called *per-channel quantization* [11]. *TensorFlow Lite* has enabled this by default for *convolutional layers* [6].

Per-channel quantization means that the weight kernel uses separate scale and zero point parameters for each output channel. As explained in Section 2.1.1, there exists C_{out} number of weight kernels, each with the dimension $K \times K \times C_{in}$; therefore, C_{out} pairs of quantization parameters are required for each convolutional layer. This arrangement issues greater memory footprint, but has a positive effect on prediction accuracy [11].

2.3.4 ReLU6 Within Quantized ANNs

With the established theory from 2.3, one can note that $(Quant \circ ReLU6)(x) = Quant(x)$, given that $f_{min} = 0$, $f_{max} = 6$. The $Quant$ function will *clamp* its output into the specified interval which is equivalent to the behaviour of $ReLU6$; therefore, in a quantized ANN, all $ReLU6$ layers can be disregarded and instead set quantization parameters (Δ, Z) for the output of the previous layer accordingly. To clarify, it is known that the output after a $ReLU6$ is contained in the interval (0, 6), and is also just an identity function ($f(x) = x$) within the interval; the quantization parameters is then simply based on this interval. The act of merging in the $ReLU$ with the output quantization parameters is called *ReLU folding*.

Figure 2.9 shows an illustration of a quantized $ReLU6$; the x-axis shows the input to the function, as a real number; the y-axis represents values that the quantized *ReLU6* can output.

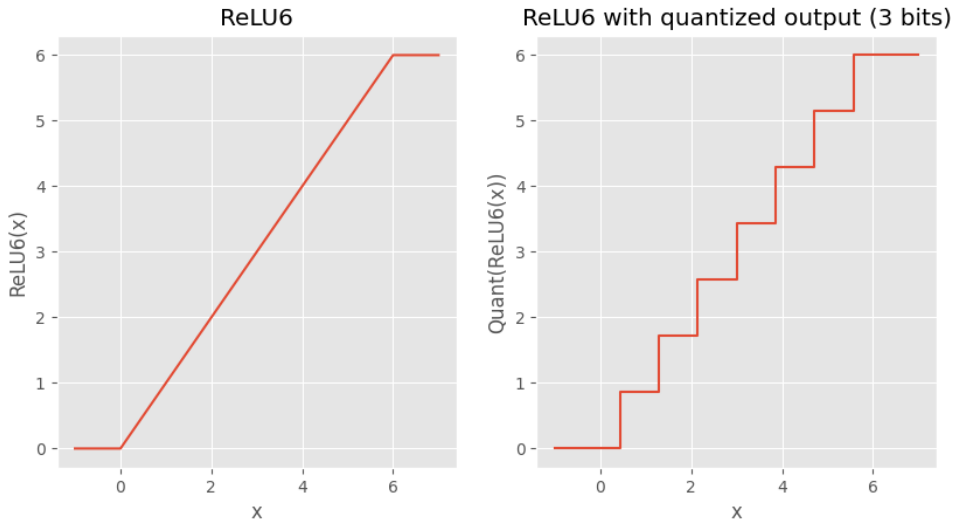


Figure 2.9: A side-by-side comparison between a regular ReLU6 and a ReLU6 which output is quantized with 3 bits and $f_{min} = 0$, $f_{max} = 6$.

2.3.5 Batch Normalization Within Quantized ANNs

During inference of a quantized ANN, the trainable parameters are frozen, i.e. fixed. The same goes for batch normalization. One can see in Equation (2.9) that the layer is just an affine transformation with constants. The constants are subject to offline optimizations with what is called *Batch-normalization folding* when batch normalization is following a convolution or fully-connected layer [10]. Starting with Equation (2.9), removing subscripts for brevity, we can split it into two parts, one constant and one variable part:

$$y = \gamma \cdot \frac{x - \mu}{\sigma} + \beta = \frac{\gamma}{\sigma}x + \beta - \frac{\gamma \cdot \mu}{\sigma}$$

$$\Rightarrow Const = \beta - \frac{\gamma \cdot \mu}{\sigma},\ Var = \frac{\gamma}{\sigma}x$$

Since the batch normalization is after a convolution or fully-connected layer, the variable x in this case is the scalar product of activation and weights ($x = a \cdot w$). Now $\frac{\gamma}{\sigma}$ is just a constant scale, thus it can be optimized by folding it with the weights:

$$Var = \frac{\gamma}{\sigma} \cdot x = \frac{\gamma}{\sigma} \cdot a \cdot w = a \cdot \frac{\gamma}{\sigma} \cdot w = a \cdot \tilde{w}$$

In more detail, following the scheme in [10], the folded weights $\tilde{w}$ are defined as:

$$\tilde{w} = \frac{\gamma \cdot w}{\sqrt{\sigma^2 + \epsilon}}$$

where ϵ is a small constant added for numerical stability, to prevent division with zero. The constant offset *Const* can simply be added in conjunction with the bias, hence the folded

bias will be the following:

$$\begin{aligned}\tilde{b} &= \frac{\gamma \cdot b}{\sqrt{\sigma^2 + \epsilon}} + Const \\ &= \frac{\gamma \cdot b}{\sqrt{\sigma^2 + \epsilon}} + \beta - \frac{\gamma \cdot \mu}{\sigma}\end{aligned}$$

To summarize, the optimization removes the computational cost and memory footprint during inference since the parameters are incorporated into the layer.

2.4 Signal-to-Quantization-Noise Ratio

When quantizing weights from a higher to lower precision such as from 64-bit floating point to an 8-bit unsigned integer, rounding error, also called quantization error, is introduced. To see the effect of the error, one can view it as adding noise to the given signal:

$$\tilde{w} = w + \eta_w \tag{2.20}$$

where w is the weight, $\tilde{w}$ is the value of the weight after quantization and finally, η_w is the noise. Using the insight from Equation (2.20) we arrive at the formula for *signal-to-quantization-noise ratio* (SQNR):

$$\xi = \frac{E[w^2]}{E[\eta_w^2]} \tag{2.21}$$

Commonly, it is desired to express the SQNR in decibel (dB):

$$\xi_{dB} = 10 \log_{10}(\xi) \tag{2.22}$$

Chapter 3
Method

The work of this is summarized with the steps below, where each step had to be finished before proceeding to the next one.

1. Research *TensorFlow*, quantization and related work within the field
2. Develop the core functionality of the framework
3. Extend the framework with new features as far as time allows
4. Evaluate the framework with smaller experiments

After the authors were familiar with the concept of quantization within neural networks and related work that are presented in Section 1.3, adequate knowledge were in place to proceed on to starting the actual implementation.

3.1 Implementation

3.1.1 Core Functionality

Because this project both focuses on carrying out experiments, motivated from the research questions listed in Section 1.2, and providing a software framework that enables future experiments to be carried out, there was a need for a structured implementation that is extendable with additional features. Specifically, the features that are part of this implementation are:

- a quantized, custom implementation of the convolutional layer, in Section 2.1.1, and unit testing it thoroughly
- batch-norm- and ReLU folding, as explained in 2.3.4 and 2.3.5

- making the network compute all its quantization parameters (Section 2.3), which is called activation calibration and is explained in Section 4.2.1
- main program that runs inference on the partially quantized model and compares it to a non-quantized counterpart, and measures them both for classification accuracy

With these features, a user can start to investigate how the accuracy changes when it is quantized for all or some of its convolutional layers in comparison to a non-quantized model; however, this implementation does not provide any metric beyond classification accuracy which opens up for extensions. Also, to widen the possible use of the framework, more network layers could be implemented.

3.1.2 Extended Functionality

The authors decided to extend the framework with features as time allowed, which were the following, in no particular order:

- quantized, custom implementations of the rest of the network layers in Section 2.1.1 and unit tests to each one of them
- hyper-parameters to the implemented layers that controls which quantized implementation to use; some compromise precision for computation speed and/or memory usage (these are described in Section 4.3)
- run inference on a *TensorFlow-Lite* model, in parallel to the other two, that act as a fully quantized model, that opens up for more accuracy comparison
- SQNR metric as a measure of noise, as explained in Section 4.4
- make the framework export a graph displaying quantization error (example graph in Figure (4.3)), that act as a visual representation for where weaknesses, with respect to noise, are found

3.2 Conducting Experiments

As a final step, several experiments were carried out, both to evaluate the implementation of the framework in its whole, and to answer the research questions from Section 1.2. With these experiments, a decision can be made on whether extending the framework even further is a reasonable idea. Should the experiments instead not lead any further from their results, this can be concluded as well. In summary, the experiments are focused on evaluating quantization on different combinations of the convolutional layers within *MobileNetV2* by using a dataset from the database *ImageNet*.

Chapter 4
QPANN - The Software Framework

Quantization Profiler for Artificial Neural Networks (QPANN) is the central part of this thesis; it has been developed from scratch with the support of the *TensorFlow* library. The purpose of this framework is to evaluate implementations of ANN operators; it is therefore important to verify that every operator works as expected so there are not any flaws in potential hardware translations of these operators. This section will go through how QPANN works on a high abstraction level, what the intention of its modules are, how one can use it to evaluate a quantized model.

4.1 Primary Functionality

The main idea behind QPANN is to select layers within a ANN (the reference model), and replace them with other implementations, such as a quantized counterpart. In fact, the framework accepts any custom operations to be carried out within the replacing layer; it is up to the user running an experiment to know and verify that the replacing layer has the same intention as the replaced layer.

When executing QPANN once, the reference model, which ideally is a 32- or 64-bit floating point model, will be reconstructed to create the *recreated model*, which is a copy of the *reference model* with the replaced layers which were specified by the user. In addition to this, the program will export the *reference model* to a 8-bit *TensorFlow Lite Model* with *TensorFlow*'s own *Lite converter* [11]. QPANN then runs inference for the three models and gathers top-1 and top-5 accuracy metrics for them. After the program terminates, the user can compare the accuracy between the three models in order to evaluate the *recreated model*. By doing multiple runs of the program, the user can then investigate what the bottlenecks are of the *recreated model* when it comes to classification accuracy.

Currently, QPANN only accepts models used for image classification, i.e., models that takes in images and predicts probabilities for every possible label; therefore, this section is limited to cover only such models, even though it should be possible to extend QPANN with

other types of models. The framework has been written with the primary intention of being able to run on *MobileNetV2* with the widely used dataset of 1000 labels called *ImageNet* [4].

Custom layers are implemented as *Keras* layers within *Tensorflow* (*Keras* is a nested high-level API within *TensorFlow* for creation and training of deep-learning models [1]). Their implementations are responsible for performing the *three* steps described in Section 4.2, i.e., quantize the input, perform the operations and then dequantize back to floating point. During planning of the project, *Keras* alone was declared to not be sufficient because there is a need to do operations on bit level; therefore, the low-level operators are programmed in C++ for full control of the implementations. Figure 4.1 displays how the modules are interfaced; the C++ implementation is hidden from the rest of the program and only visible from within the wrapping *Keras* layer.

The labels shown in Figure 4.1 are summarized below:

Data samples and labels is a dataset of validation images, that the **Reference Model** has trained to classify, and the corresponding ground-truth labels. The **Reference Model** is a trained, 32-bit float model.

Model Recreation is the module that is responsible for copying the reference model, but also replacing the specified layers that are stated in the **Recreation Config**. After this operation, the **Recreated Model** should be a copy of the reference model, but some layers contain custom-implemented operators so that one can study how these operators affect accuracy.

The **TF-Lite Model** is exported from the **Reference Model** with *TensorFlow*'s native method.

QPANN is the main module that delegates instructions to other modules. The main module will run inference on the **Recreated Model**, **TF-Lite Model** and **Reference Model** with the same validation set. After running through the entire dataset, the program returns with the accuracy for the three models.

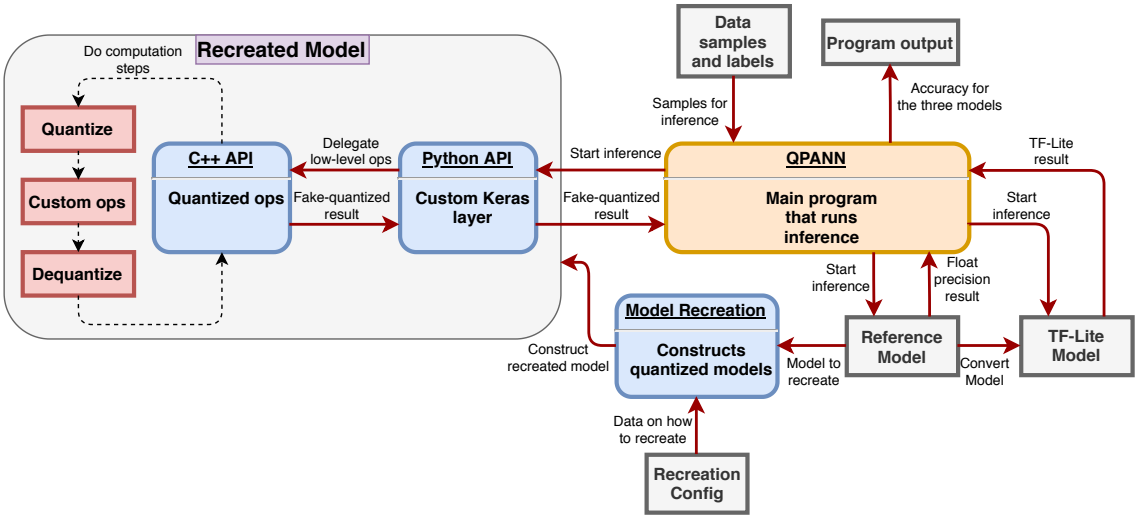


Figure 4.1: A diagram of QPANNs high-level architecture.

4.2 Fake quantization within QPANN

Early in the project's lifetime, the choice to use *TensorFlow Lite* was considered because it is *TensorFlow's* software for building and deploying quantized models on e.g., mobile devices. This idea was dropped because there was a need of using the extended functionality of regular *TensorFlow*. Instead of using a fully quantized model, a *32-bit floating point* model is used. For the layers that are to be quantized, *fake quantization* is used for simulating the effect of real quantization. In practice, this means that the following steps are carried out for a quantized layer within QPANN:

1. quantize the input of the layer from *32-bit float* to *8-bit signed integer*
2. perform the operation within the layer in its quantized form
3. dequantize the output back to *32-bit float* and propagate it to the next layer

A question that might arise then is: For two adjacent fake-quantized layers, Will additional error be introduced when output activations are dequantized in the first layer, and then quantized again in the second layer, as opposed to just propagate the activations in the quantized format to the second layer? Actually, the results are the same for the both cases, if the dequantization and quantization use the same scale Δ and zero point Z, i.e., $(Quant \circ Dequant)(q) = q$, for all quantized numbers q. This can be confirmed by studying Equations 2.14 and 2.15. This means that if all layers are fake quantized, then the model would behave as real quantized model.

The process of quantizing and dequantizing multiple times in the network is naturally slower than a fully quantized model; but fits this application because there is not a critical need of performance in terms of speed.

4.2.1 Determining Quantization Parameters

As mentioned, the quantization scheme requires the parameters Δ and Z for every operand, output vector inclusive. For weights, there are two supported methods in QPANN to decide them. The first way enforces *symmetrical* quantization and defines the following:

$$\begin{cases} f_{min}(w) = -r \\ f_{max}(w) = r \end{cases} \quad \text{where } r = max(max(w), -min(w))$$

A motivation for choosing the *symmetric* approach is to make operations like the scalar product simpler for the hardware. The *8-bit integer* format in QPANN is *signed* which implies that the zero point Z becomes 0; one sum in the scalar product (Equation (2.19)) does not have to computed as a result of this, because it is guaranteed to be zero, along with NZ_AZ_B, which is also zero.

The second method relaxes the requirement of *symmetric* quantization, but still requires the number zero to be representable, by setting:

$$\begin{cases} f_{min}(w) = min(0, min(w)) \\ f_{max}(w) = max(0, max(w)) \end{cases}$$

The resulting floating-point range are then substituted into the quantization-converting Equations (2.10) and (2.11) to find Δ_w and Z_w.

For input and output to a layer, there are no minimum and maximum values in the same manner as for the weights. Instead, a method called *activation calibration* is used, which is also supported by *TensorFlow Lite* [1]. This means that the full precision model that is to be quantized are run with a desired amount of samples, e.g., 100. Then the minimum and maximum values across all samples are found for the input and output tensors. These values then retrieves Δ and Z in the same way as for the weights.

4.3 Supported Operators

For any *Keras* layer that one intends to replace, an implementation of a *quantized* version needs to exist. These quantized counterparts have been implemented from the ground up together with unit tests that run the implementations in isolation; this section lists the replacement operators that have been implemented.

Additionally, some of these layer implementations have hyper-parameters attached to them which are there to adjust specific parts of the layer, e.g., whether the layer should have *symmetric* or *asymmetric* quantization for weights. QPANN is written in a such a way that it is extendable with respect to these hyper-parameters. The ones implemented during the work of this thesis are listed together with the corresponding layer in the subsections below. The parameters are named the same as they are in the source code.

4.3.1 Hyper-Parameters

This section will describe the supported hyper-parameters in general, whereas in Section 4.3.2 will list which hyper-parameters are attached to which layer. Every hyper-parameter in this list are boolean flags, i.e., they can be toggled on and off for an implementation.

- `rounding_div`: Some operators make *fixed-point* multiplications, i.e., integer multiplication followed by a *bit-shift division*. This bit-shift division can be chosen to round its output to the nearest whole number. This rounding is expensive for the hardware because it contains a conditional branch. This flag controls whether to do *truncating* or *rounding* division.

- `symmetric_weights`: This flag toggles whether to force *symmetrically quantized* weights or not. Section 4.2.1 explains this property.

- `per_channel_quantization`: As covered in Section 2.3.3, this controls whether to apply *per-channel* (output channel) or *per-layer* quantization for the *weights*. Enabling this flag increases memory requirement but may lead to prediction-accuracy increase.

- `merge_batch_norm`: Toggles batch-norm folding, as explained in Section 2.3.5. This means to merge the batch-norm layer into this layer.

- `merge_relu`: If the layer is followed by a ReLU6, this layer can fold the ReLU6 into the output quantization parameters. Since the range of a ReLU6 function is $[0, 6]$, this range defines $f_{min} = 0$ and $f_{max} = 6$ for the output scale and zero point in order

to construct a uniform quantizer for that range. In practice, this means to remove the ReLU6 layer/activation completely, as explained in 2.3.4.

4.3.2 Layers

Convolutional Layer

The *quantized* convolutional layer applies the convolution formula described in Section 2.1.1 and applies it in a quantized format as described in Section 2.3.2 (This section covers matrix multiplication, but it is analogous to convolution because they are both composed of a set of scalar products). This layer also support bias weights that are added in the quantized domain.

Hyper-parameters:

- `rounding_div`
- `symmetric_weights`
- `per_channel_quantization`
- `merge_batch_norm`
- `merge_relu`

Depthwise Convolution Layer

With the theory of *depthwise convolution*, described in Section 2.1.1, a quantized layer has been included into this framework. This step is the first of two for *depthwise separable convolution*. The implementation is close to identical to the convolution layer; it is likewise a set of scalar products, which makes Equation 2.17 also the underlying formula. The layer supports quantized bias addition as well.

Hyper-parameters

- `rounding_div`
- `symmetric_weights`
- `per_channel_quantization` The *per-channel* quantization refers to output and input channel simultaneously because one output channel is dependent on a single input channel; there is a 1-to-1 mapping.
- `merge_batch_norm`
- `merge_relu`

Add Layer

This layer performs simple addition between two tensors; the two tensors need to have the same shape and the output tensor inherits this shape as well. This layer has no weights and supports different quantization parameters for its input and output.

Hyper-parameters:

- `rounding_div`

Fully Connected (Dense) Layer

The fully connected layer is a simple matrix multiplication between the input and the weights. This quantized implementation uses Equation (2.17) to achieve the multiplication and applies bias addition to the output if desired.

Hyper-parameters:

- `rounding_div`
- `symmetric_weights`

Global Average Pooling

Global Average Pooling inputs a three-dimensional tensor (height, width, channel) and outputs the mean of each channel. The layer has no weights and the input and output have the same quantization parameters.

4.4 The SQNR-Recorder Module

Since the framework is a profiler for quantized ANNs, there is supported functionality for quantifying the noise that each replaced layer contributes with. The metric used is *signal-to-quantization-noise ratio* (SQNR), which is introduced in Section 2.4. When inference is started for the validation samples (after activation calibration), the framework will use the module called *SQNR Recorder* to accumulate the sum of the reference activations and the sum of activation noise for each layer. After inference is complete and the data has been recorded, Equation (2.21) is used for computing the final SQNR for all layer outputs. By comparing the outputs from the *replaced-* and *reference* layers, the *mean* SQNR across all their activations can be computed. This computation can be expressed with the equations below.

First, define λ_i as the sum of quadratic activation values for the activation with index i of the *reference model*. In a similar way let μ_i be the sum of the quadratic difference between the *reference model*'s activations and the *recreated model*'s activations. The sum is computed from every validation sample n.

$$\lambda_i = \sum_n (a_{ni}^{ref})^2 \tag{4.1}$$

$$\mu_i = \sum_n (a_{ni}^{ref} - a_{ni}^{rep})^2 \tag{4.2}$$

Use Equation (2.21) to compute the SQNR for that layer activation:

$$\xi_i = \frac{\lambda_i}{\mu_i}$$

Finally, compute the *mean* of this ratio and apply the logarithmic transformation. N is the number of activations for the layer.

$$SQNR(layer) = 10 \log_{10}\left(\frac{1}{N}\sum_{i=1}^{N} \xi_i\right) \tag{4.3}$$

This computation is done in the *SQNR-recorder* module, where it accumulates the quadratic sums from Equation (4.2) and (4.1) in memory. Figure 4.2 shows a diagram of how the data is propagated during inference.

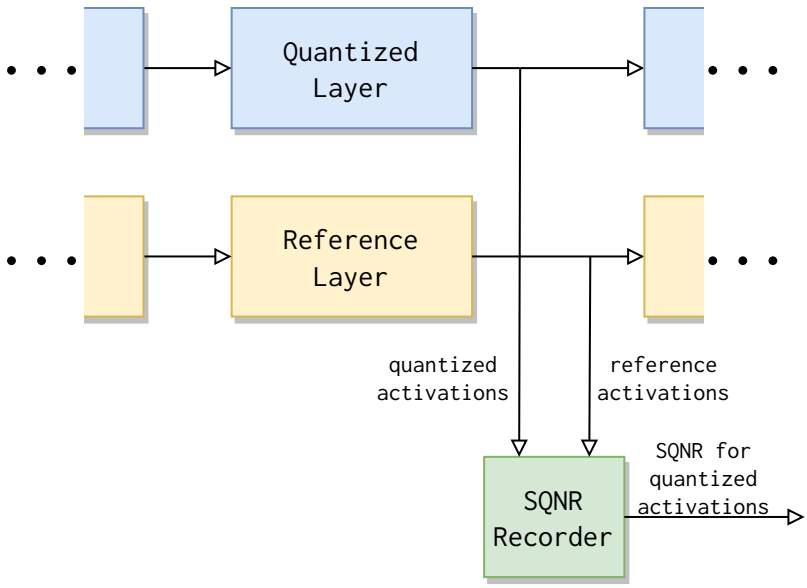


Figure 4.2: The output activations from all layer pairs are accumulated into the *SQNR Recorder*. The SQNR can then be computed after inference is complete.

After the *recorder* has computed the SQNR for all layers, QPANN exports the results of these computations in a viewable graph; Figure 4.3 shows an example of such a graph. Each directed edge display the output SQNR from layers, which act as vertices in the graph. The SQNR values are displayed in dB, computed from Equation (4.3). With this graph, the user can detect which layers might reduce the accuracy of the model, and optimize operators accordingly. Note, if the output SQNR of a layer is lower than its input SQNR, then it would mean that the layer is introducing more noise than it received during inference, and could then benefit from a precision optimization.

The layer's noise contribution is compared to the reference model, which then in turn does not necessarily have to be better in accuracy. Quantization noise does not directly mean worse accuracy; it is just a measure of deviation from the reference model. The noise can also be greater on activations that are less important for the output prediction. The SQNR measure does not take that into account, which is why the user should not believe in a monotonous relationship between output SQNR and prediction accuracy.

A paper [12] showed an approach to optimize *convolutional neural networks* with the use of SQNR, which was the inspiration for including the metric. The paper also showed that the SQNR for the output layer is approximately equal to the *harmonic mean* of the output of the preceding layers (Equation 4.4). This approximation does work for convolutional layers, but not as well for fully-connected layers, according to the paper.

$$\frac{1}{\xi_{out}} = \frac{1}{\xi_{L1}} + \frac{1}{\xi_{L2}} + \ldots + \frac{1}{\xi_{LN}} \tag{4.4}$$

Whether this formula is accurate or not, there is inevitably a relationship between output SQNR and prediction accuracy. The user might have to investigate how the SQNR measurements contributes to the performance for that specific network architecture. Exactly how the relation between SQNR and accuracy is manifested in practice for other types of layers is not examined within this work; it is instead left as potential future work. In paper [12], the authors optimized for high SQNR in order to achieve better prediction accuracy which could mean that this type of optimization can be carried out with QPANN for an arbitrary classifying network.

A further note on SQNR metrics during inference is that measures for any layer is dependant on the noise produced from preceding layers. If the user desires to measure the SQNR of one layer in isolation, a model with just that layer replaced can be created.

4.5 Configuring Inference

An important matter of this work is to enable the user to easily switch and try out different implementations of layers and operators; therefore, a *config-file* specification has been established. Any user can then change parameters easily within the *two* config files that QPANN needs in order to execute. The two config files are named *network config* and *recreation config*; they are regular *python* files. See Appendix A for examples of how the files are structured.

The *recreation config* relates to parameters that affect the construction of the *recreated model*. This is where the user chooses which layers to replace and provide hyper-parameters (specified in section 4.3) to them.

The *network config* file relates to the reference model, activation-calibration batch-size and the validation dataset. Here one can specify how many samples the inference will be carried out on and where to load the image data from.

To ensure that a carried out experiment can be investigated and/or reconstructed any time in the future, QPANN exports the config files and places them together with the accuracy result for the models.

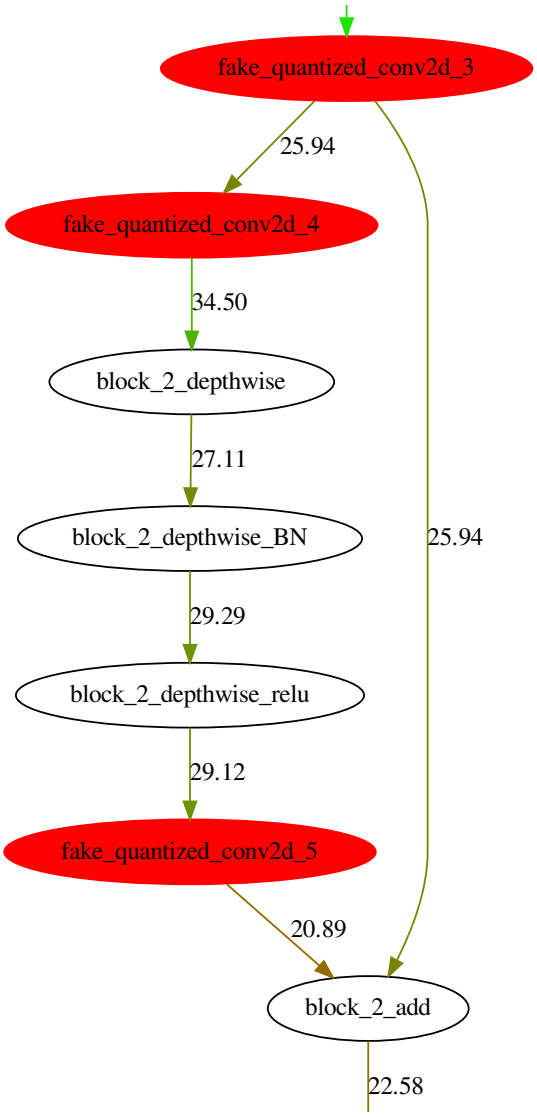


Figure 4.3: An example of a SQNR graph, of the recreated model, that the framework exports after inference. This graph can be used to find potential weaknesses in the layer structure for the model to find which layers need to remain in higher precision and which can be quantized harder. The red vertices represent replaced layers.

Chapter 5
Evaluation

5.1 Experimental Setup

The following chapter will go through several experiments that were done with QPANN and the results of these experiments. The experiments will not be aimed towards finding optimal implementations of a quantized model, but rather to show examples of what QPANN can find with experiments and how one can proceed investigating to find a balanced implementation of a quantized model. A balanced model in this case would be a model that has a good standing in accuracy versus speed and memory usage. The experiments are limited to only replace convolutional layers with different quantized implementations, since, at the time of experimentation, these were the only layers that had a complete implementation.

The reference model used in all experiments is *MobileNetV2*, which is pretrained on *ImageNet*. In *MobileNetV2* there are 35 convolutional layers in total, evenly distributed across the network, which creates a large space of implementation permutations to discover.

The results from the experiments on our custom quantized model, will be compared with *TensorFlow*'s 32-bit floating-point model (the reference model) and *TensorFlow Lite*'s 8-bit integer model (converted from the reference model using the converter embedded in *TensorFlow Lite*). All experiments were done by validating the result on 50000 images of *ImageNet*. The *activation-calibration* batch (explained in Section 4.2.1) were 100 samples selected randomly for each new experiment, inspired by paper [11]. All SQNR values in the results are in dB.

The first experiment to be carried out is to replace all convolutional layers in *MobileNetV2* with *quantized* layers. Following *TensorFlow Lite*'s quantization specification [6]; for *all* layers, *rounding division*, *symmetric weights*, *per-channel quantization* and *batch-norm folding* were present; furthermore, the layers that also could fold a *ReLU6* did so, which were 18 of the 35 convolutional layers.

In the second experiment we will identify five *low* and five *high* data points with respect to how they affect SQNR, given the result from the first experiment. As previously explained with Equation (4.4), the SQNR of the model should be mostly affected when modifying its

weakest links; therefore, we expect the accuracy to differ more if the five convolutional layers with *low* SQNR are modified rather than if the five with *high* SQNR are. The identified convolutional layers will vary their hyper-parameters; more specifically, bit width (*32-bit float precision*), type of division (*truncating division*) and finally the amount of quantization parameters used on a given layer (*per-layer quantization*).

To answer whether *MobileNetV2* is the most sensitive in the beginning or in the end, a third experiment will be carried out. In the experiment we will quantize the first half of the convolutional layers and compare it with the second half of the convolutional layers quantized, using the same scheme as the first experiment.

5.2 Experiments

For a comparison of how a custom quantized model compares to *TensorFlow*'s 32-bit floating-point model and *TensorFlow Lite*'s 8-bit integer model, the result of these two models are shown in Table 5.1.

Table 5.1: Results for the *reference model* (*TensorFlow* 32-bit float) and *TensorFlow Lite* (8-bit integer) when validating on the 50000 samples of *ImageNet*.

Setting	Top-1 Accuracy	Top-5 Accuracy
Reference Model	68.05%	88.40%
TensorFlow-Lite Model	67.39%	87.82%

Experiment Suite 1

The result of quantizing all convolutional layers, got a top-1 accuracy of 63.81%, top-5 accuracy of 85.65% and 11.14 in the model's output SQNR (shown in the **A** setting in Table 5.2). The top-1 accuracy is therefore a decrease of over 4 percent points compared to the *reference model*, which means our implementation cannot match *TensorFlow Lite*'s 8-bit integer model.

To see how SQNR propagates through the model, Figure 5.1 shows how it accumulates for each layer in the topmost graph. The bottom graph instead shows the difference between the SQNR (computed from Equation (4.3)) for the layer output and its input, i.e.:

$$SQNR(layer[i+1]) - SQNR(layer[i])$$

Note: QPANN does not export Figure 5.1; it is included in this report for helping visualization for the specific case of *MobileNetV2*. If the graph is *not* completely sequential, i.e., every layer has only input and output, this type of plot is misleading and therefore not included as part of the framework.

A part of the SQNR graph exported for this experiment is shown in Figure 5.2. As seen in both in the graph and Figure 5.1, the SQNR is oscillating up and down when propagating through the layers; specifically some of the non-quantized *batch-normalization* layers have been observed to increase SQNR, like vertex *block16_depthwise_BN* does in Figure 5.2. The

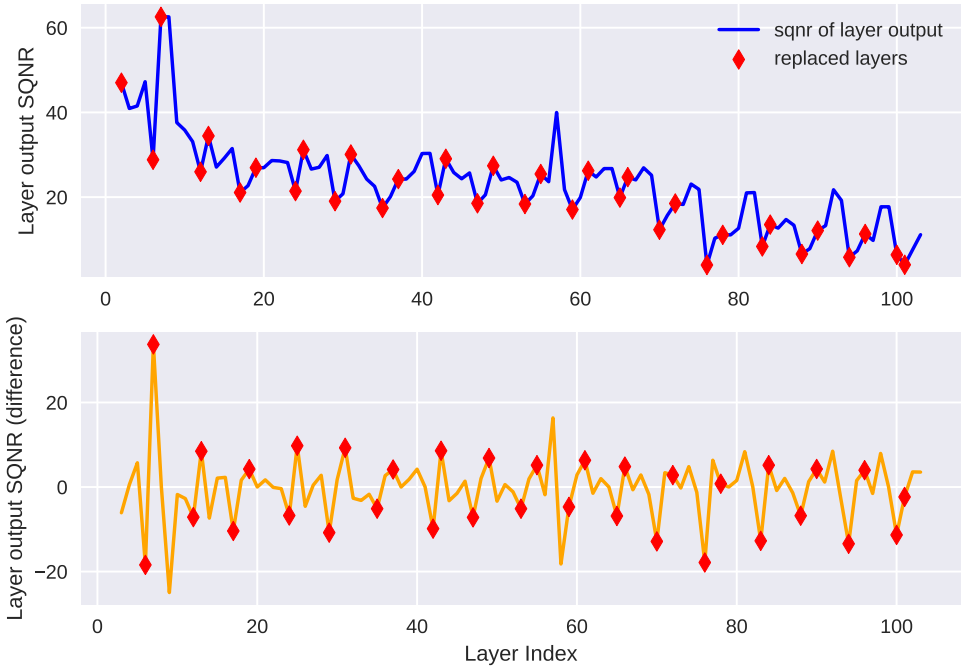


Figure 5.1: Mean, through 50000 validation samples, of SQNR (in dB) of output activations for every layer index of *MobileNetV2*. Starting from index 0, where the image first pass through, it propagates through all the layers until the output is computed at the last index 103.

reason for this unknown, but a theory of ours, from brief investigations, is that such *batch-normalization* layers are strongly affected by β, and not as much from $\gamma \cdot \hat{x}_i$, i.e., the output is dominated by β because it is, e.g. five times larger than the other term; refer to Equation (2.9) to see these parameters. The following equations clarifies our theory. Consider the input activation to a *batch-normalization* layer $\hat{x}_i$. Within the experiment, this activation has quantization noise so it can be expressed as $x_i^q + \eta_i$. The SQNR (Equation (2.21)) for the input is:

$$\xi_i^{in} = \frac{E[(\hat{x}_i)^2]}{E[\eta_i^2]}$$

The output of the *batch normalization* is defined as:

$$y_i = \gamma \hat{x}_i + \beta = \gamma x_i^q + \gamma \eta_i + \beta$$

The SQNR of the output then becomes:

$$\xi_i^{out} = \frac{E[y_i^2]}{E[\eta_{y_i}^2]} = \frac{E[(\gamma x_i^q + \gamma \eta_i + \beta)^2]}{E[(\gamma \eta_i)^2]}$$

As seen in the output's SQNR, large β's can potentially increase the SQNR; furthermore,

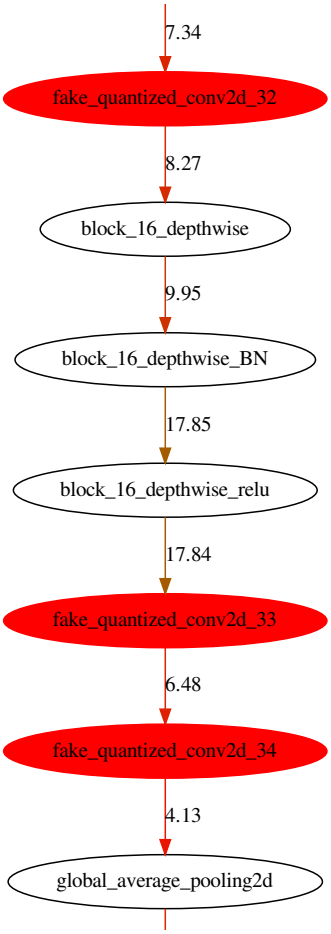


Figure 5.2: A part of the SQNR graph that QPANN exported for the experiments of quantizing all convolutional layers. The red vertices represent quantized layers.

this can be applied to *bias* additions in general. Note that this proof was based on a *batch-normalization*-layer without noise. For a fully quantized model, like one of *TensorFlow Lite*, every operation are in a quantized format; a *batch-normalization* layer is usually folded into the previous layer, as explained in Section 2.3.5. One should therefore be careful when making conclusions about how well a partially quantized model within an experiment generalizes to a fully quantized model. Because of this potential increase of SQNR when adding constant terms, like *bias* additions, one can investigate if the preceding layer can be optimized even further for performance rather for precision, because the *bias* term can potentially save the output SQNR and consequently even accuracy. The theory we have provided with this experiment is just an example of what one can discover when using the framework.

Experiment Suite 2

In the bottom plot of Figure 5.1, together with the SQNR graph that QPANN exports, we identified five *low* and five *high* data points with respect to how they affect SQNR. In Figure 5.1 it is the sets of indices {6, 70, 76, 82, 94} and {8, 13, 25, 31, 43} respectively. The motivation for running these experiments is to show that QPANN can help its user to identify operators

within a model where the accuracy is sensitive to modifications and where precision-reducing optimizations are allowed. As previously mentioned, we expect the accuracy to differ more if the five convolutional layers with *low* SQNR are modified rather than if the five with *high* SQNR are. With this information, seven experiments, labeled **A-G** were set up to see if the hypothesis is true.

For all the entries in the list below, a *quantized* layer with no further description means that it uses *TensorFlow*'s default quantization scheme, i.e., it is quantized *symmetrically* and *per channel*, have *rounding division* in its implementation, has *Batch-Normalization-* and *ReLU-*folding if applicable (Section 4.3 shows further information regarding these attributes).

- **A:** All convolutional layers are quantized
- **B:** Five layers with *high* output SQNR use *32-bit float precision*; the rest are *quantized*
- **C:** Five layers with *low* output SQNR use *32-bit float precision*; the rest are *quantized*
- **D:** Five layers with *high* output SQNR are *quantized per tensor*; the rest are *quantized per channel* (*TensorFlow*'s default quantization scheme)
- **E:** Five layers with *low* output SQNR are *quantized per tensor*; the rest are *quantized per channel* (*TensorFlow*'s default quantization scheme)
- **F:** Five layers with *high* SQNR implement *truncating division*; the rest use *rounding division* (*TensorFlow*'s default quantization scheme)
- **G:** Five layers with *low* SQNR implement *truncating division*; the rest use *rounding division* (*TensorFlow*'s default quantization scheme)

Table 5.2: SQNR and accuracy metrics for the different experiments listed above.

Setting	SQNR	Top-1 Accuracy	Top-5 Accuracy
Reference model	N/A	68.05%	88.40%
A	11.14	63.81%	85.65%
B	10.80	63.82%	85.69%
C	10.82	64.00%	85.69%
D	10.70	63.68%	85.58%
E	10.12	63.50%	85.18%
F	10.76	63.95%	85.65%
G	10.31	63.73%	85.28%

Indeed, the hypothesis, that *lower*-SQNR layers affects accuracy more when modified, conforms with the results in Table 5.2. We investigate each of the settings **B-G** pairwise (**A** behaves as the default reference setting).

B and **C** removes five layers of *high* and *low* SQNR respectively. According to our hypothesis, **C** should have higher accuracy than **B**, which is the case. Keep in mind, even though the accuracy increase may be small, when optimizing a model every small increase is important.

Removing quantization from five high layers (**B**) does barely raise the accuracy compared to **A**; however, when five lower layers are changed to full precision, the *top-1 accuracy* increases noticeably from 63.81% to 64.00%.

D and **E** introduce *per-tensor quantization* for five layers of *high* and *low* SQNR respectively. We expect the SQNR to drop more where the *lowest* layers are compromised (**E**), which is indeed the case for both the *top-1-* and *top-5 accuracy*.

F and **G** implement *truncating division* for five layers with *high* and *low* SQNR respectively. The same holds for this pair; **G** cause a greater accuracy drop than **F**, which we expected.

After the weak- and strong links are found in the model, we have showed that if the implementation of the weaker links are optimized for precision and the stronger links are optimized for computation speed and memory, we can reach a desired state of trade-off between accuracy, computation speed and memory usage. This experiment does not verify that this approach always works for any model, but the results are a solid indication that this strategy can be used when optimizing a model.

Experiment Suite 3

In the final experiment, the first and second half of the convolutional layers are quantized separately; the remaining half use *32-bit* precision layers. To clarify, there are 35 convolutional layers in total for *MobileNetV2*; starting from the *input layer* the 18 first layers (according to *breadth-first search* on the model's graph) corresponds to the first half and the rest makes the second half. From the topmost graph in Figure 5.1, one interesting property that can be seen is that the SQNR suddenly starts to drop after around layer index 70. With this in mind, the hypothesis is that *MobileNetV2* should be more sensitive in the second half.

The results of how SQNR accumulates for the two experiments are shown in Figure 5.3. Starting by observing the blue curve, where the first half of the convolutional layers are quantized, one can see that it drops until it reaches an SQNR of 20. After this point, the SQNR approximately oscillates between 20 and 30.

For the green curve, where the second half of convolutional layers are quantized, the SQNR instead starts to drop rapidly after quantization is introduced; it reaches below the blue curve at the same layer indices as Figure 5.1 showed. This indicates that the output noise is more affected by the *later* layers of inference. To complement the noise data with actual results of accuracy, Table 5.3 shows this along with the output SQNR. There is a significant accuracy difference between quantizing the first- and the second half of the network. The network with the first half quantized only drops 0.16 percentage points in *top-1 accuracy* from the reference model compared to the second half's drop of 3.82 percentage points. The reason for this difference was not investigated; it could be because of the fact that activations, for this specific model architecture, have more channels the closer they come to the network's output. This results in longer accumulation chains (explained in Section 2.3.2) which could amplify quantization errors.

The results of this experiment shows that, for this specific case of *MobileNetV2* trained on *ImageNet*, the first half of convolutional layers are more allowing for memory- and computation speed optimizations, while the second half will reduce accuracy even further if additional

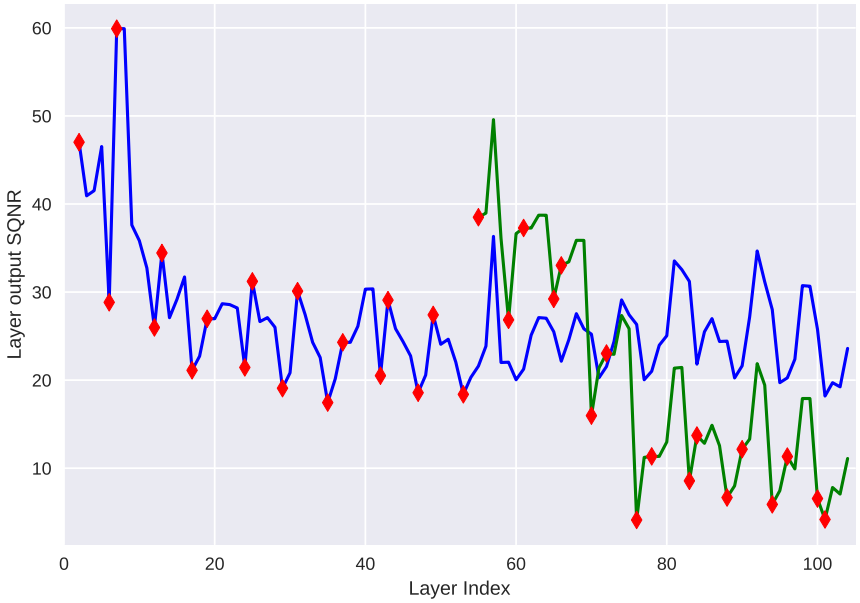


Figure 5.3: Accumulation of SQNR if the first (blue curve) and the second (green curve) half of the convolutional layers are quantized. The red markers denote quantized layers.

precision-compromising implementations are introduced for those layers. With QPANN, one could continue optimizing the operator implementations with these results in mind.

Table 5.3: Performance comparison between quantizing the first and the second half of the convolutional layers of *MobileNetV2*; the first half contains 18 layers and the second half 17 layers.

Setting	SQNR	Top-1 Accuracy	Top-5 Accuracy
First half	23.60	67.89%	88.18%
Second half	11.10	64.23%	85.89%
All *Conv2d* layers	11.14	63.81%	85.65%
Reference model	N/A	68.05%	88.40%

Chapter 6
Conclusion & Future Work

6.1 Conclusion

For a use case of running ANNs with limited memory, hardware capacity or similar, it is paramount to be able to analyze the relationship between computation precision and inference accuracy. Generally speaking, when compromising precision, a question that arises asks how much accuracy is sacrificed in order to conform to constraints of the domain where the model is deployed.

Our evaluation demonstrates how QPANN can be used quantitatively to investigate how inference accuracy is affected by degrading parameters, such as faster division or *per-tensor quantization*. The framework particularly allows for studying of how SQNR may or may not be able to model how accuracy is affected by altering parameters. A user can select layers to provide with custom implementations to investigate how they change the model's accuracy. The framework gives the developer full control of the implementation and can be extended in multiple ways to further research the relationship between precision and prediction accuracy.

As a method to evaluate QPANN and its potential future usage, *MobileNetV2* was selected as a neural network to carry out experiments on. For this specific architecture, we found layers where accuracy is sensitive for numerical-precision compromising implementations. The downside is that we do not know if our findings are generalizable to other architectures; one would need to use QPANN for every architecture to investigate. However, no software available to the public, that we are aware of, is specifically tailored for the type of post-training quantization analysis which QPANN enables. Thus, we conclude that the software framework is a great contribution to the ongoing research of machine learning.

As a final remark, we consider the societal impact the framework imposes. We think that our work is rather safe from potential malicious intentions of usage; of course, we are unable to control what Arm's customers do with the optimized hardware that their future chips have. However, we do not feel that this work promotes ethical violations or similar, more than any other research within machine learning does.

6.2 Future Work

The implementation of QPANN was more time demanding than what was anticipated when starting the project; therefore, the framework has parts of it that we feel are, to some extent, incomplete and could be starting points for a possible continuation.

The first recommendation from us would be to go through the custom quantized operators and aim for closer results compared to *TensorFlow Lite*'s implementations. Even though all layers of *MobileNetV2* can be replaced with our implementations, it is not on a par with the actual accuracy of a fully quantized model that uses *TensorFlow*'s own *Lite Converter*. Similar to the *batch-normalization-* and *ReLU* folding, *TensorFlow*'s quantized models might do optimizations in between operators that we are not aware of; this could use further investigation.

Once the operators' performance are adequate, one could continue by extending the functionality of QPANN. It has only been evaluated on *MobileNetV2*; verifying that it works on other architectures is advised. Implementing quantized versions of other operators, e.g. *Softmax*, *Tanh* and *Swish*, as they use in the successor, *MobileNetV3* [7], could also be a way forward.

Another matter that is worth investigating is whether the SQNR measure for output of layers is a sound method for estimating prediction accuracy. During the work of this thesis, there was no thorough verification of this; our results could just be coincidences that apply specifically for *MobileNetV2*. The choice of including SQNR measures originates from the work presented in [12], where they tuned quantization of layers based on bit width. In this work, more general SQNR affecting implementations, e.g. truncating division, have been tested which could introduce new and different characteristics for the prediction accuracy.

As a final mention, QPANN does currently not make any estimation of how an implementation affects the hardware in terms of speed and memory; it could therefore be useful to extend the framework with this. One can study the source code and attach performance estimations for every type of implementation. An example would be that the framework could estimate memory savings from using *per-tensor quantization* over *per-channel quantization* within a network to give the user an idea of the benefit of this optimization.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Arm. Official homepage. `https://www.arm.com`.

[3] Torch Contributors. Pytorch quantization. `https://pytorch.org/docs/stable/quantization.html`.

[4] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[5] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[6] Google. Tensorflow lite 8-bit quantization specification. `https://www.tensorflow.org/lite/performance/quantization_spec`.

[7] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Hartwig Adam. Searching for mobilenetv3. *CoRR*, abs/1905.02244, 2019.

[8] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

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

[10] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2704–2713, 2018.

[11] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv preprint arXiv:1806.08342*, 2018.

[12] Darryl Lin, Sachin Talathi, and Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks. In *International Conference on Machine Learning*, pages 2849–2858, 2016.

[13] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013.

[14] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[15] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation. *CoRR*, abs/1801.04381, 2018.

Appendices

Appendix A
Config Files Example

This *python* list is contained in the *recreation-config* file:

```
recreation_config = [
    {
        # The name of the keras layer to replace
        'name': 'conv2d',

        # The indices of replaced layers
        'indices': [2,46],

        'merge_batch_norm': True,
        'merge_relu': True,
        'quantized': True,
        'per_channel_quant': True,
        'rounding_div': False,
        'symmetric_weights': True,
    },
    {
        'name': 'conv2d',
        'indices': [108],
        'merge_batch_norm': True,
        'merge_relu': False,
        'quantized': True,
        'per_channel_quant': True,
        'rounding_div': True,
        'symmetric_weights': True,
    },
    {
        'name': 'depthwise_conv2d',
```

```
        'indices': [5,22,32],
        'merge_batch_norm': True,
        'merge_relu': True,
        'quantized': True,
        'per_channel_quant': True,
        'rounding_div': True,
        'symmetric_weights': True,
    },
    {
        'name': 'dense',
        'indices': [156],
        'quantized': True,
        'rounding_div': True,
        'symmetric_weights': True,
    },
]
```

This *python* dict is contained in the *network-config* file:

```
network_config = {
    # File path to a keras model
    'model_file': path/to/mobilenetv2.h5,

    # (height, width, channel) of input
    'input_shape': (224, 224, 3),

    # A sequence, e.g. list or range, containining the indices for the
    # samples used for validation.
    'image_indices': range(1, 50001),

    # batch size for activation calibration
    'calibration_batch_size' : 100,

    # The folder containing input samples (file type png or jpg)
    'validation_samples': 'path/to/validation_set/samples/'

    # The file containing ground-truth labels for each sample.
    'validation_labels': 'path/to/validation_set/ground_truth_labels.txt'

    # The file containing the name of each label (optional)
    'label_names': 'path/to/validation_set/label_names.txt'
}
```

Appendix B
Inference Accuracy Stats During Runtime

Below is an example of inference output when executing QPANN.

```
########## ILSVRC2012_val_00006899.JPEG                    ##########
> Recreated Model Prediction    : laptop                 (23.5890%) <
> TFLite Model Prediction       : gong                   (23.4375%) <
> Reference Model Prediction    : notebook               (29.7965%) <
> Ground Truth                  : notebook                          <
########## ILSVRC2012_val_00006900.JPEG                    ##########
> Recreated Model Prediction    : Rhodesian ridgeback    (54.0703%) <
> TFLite Model Prediction       : Rhodesian ridgeback    (69.1406%) <
> Reference Model Prediction    : Rhodesian ridgeback    (75.1466%) <
> Ground Truth                  : Rhodesian ridgeback               <
```

```
#######################################
| Batch Accuracy Score:  100 samples |
|    Recreated model                  |
|      > Top 1:  61.0000              |
|      > Top 5:  77.0000              |
|    TFLite model                     |
|      > Top 1:  57.0000              |
|      > Top 5:  79.0000              |
|    Reference model                  |
|      > Top 1:  59.0000              |
|      > Top 5:  81.0000              |
#######################################
#######################################
| Total Accuracy Score: 6900 samples |
|    Recreated model                  |
|      > Top 1:  63.9130              |
|      > Top 5:  85.4783              |
|    TFLite model                     |
|      > Top 1:  67.8841              |
|      > Top 5:  87.7536              |
|    Reference model                  |
|      > Top 1:  68.8116              |
|      > Top 5:  88.2174              |
#######################################
```

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2020-06-11

EXAMENSARBETE Quantization Profiler for Artificial Neural Networks
STUDENTER Martin Lindström, Jakob Hök
HANDLEDARE Jörn Janneck (LTH), Axel Berg (ARM Sweden AB), Kevin Wohnrade (ARM Sweden AB)
EXAMINATOR Flavius Gruian (LTH)

Optimeringsverktyg som effektiviserar AI på inbyggda system

POPULÄRVETENSKAPLIG SAMMANFATTNING **Martin Lindström, Jakob Hök**

Optimering av slutledning för artificiella neurala nätverk är ett måste vid exekvering på inbyggda system på grund av begränsad datorkraft. Examensarbetet har utvecklat ett verktyg som ska hjälpa utvecklaren att analysera avvägningen mellan prestandaoptimeringar och slutledningsförmåga.

Maskinlärning och AI etablerar sig allt mer i samhället, där de används till att ge videorekommendationer, vänförslag på sociala medier, självkörande bilar, röstassistenter och mycket mera. “Artificiella neurala nätverk“ (ANN) är ett vanligt förekommande uttryck som är en mjukvaruteknik som använder sig av nätverk med moduler. Dessa nätverk tränas för att kunna känna igen och identifiera domänspecifika mönster. Efter träning används de i flera användingsområden, däribland bildklassificering. Vid bildklassificering skickas en bild in till nätverket som sedan gör en kvalificerad gissning på vad som visas på bilden; om exempelvis nätverket har tränats för att kunna särskilja hundar ifrån katter så skulle en modul till exempel kunna ha lärt sig hur en nos ser ut och en annan modul hur en svans ser ut hos de två olika djuren. Efter denna typ av igenkänning tas ett beslut av nätverket om det faktiskt är en hund eller en katt. Fundera på hur du själv skiljer på en hund och en katt!

Att skicka in en bild till ett ANN för att få ett utsägande kallas *slutledning*. Eftersom forskningen kring ANN har ökat explosionsartat det senaste decenniet, mycket tack vare tillgången på stor mängd träningsdata och kraftfullare datorer, har efterfrågan på tekniken ökats. Dessutom finns det ett behov av att kunna behärska tekniken på maskiner med begränsad datorkraft, så som mobiltelefoner. Slutledning kräver mycket datorkraft och därav för att köra det på begränsad hårdvara, behöver utvecklare optimera nätverken till att bli snabbare och effektivare, och samtidigt bibehålla tillräckligt bra slutledningsförmåga.

Ett sätt att optimera ett ANN för en mobilprocessor är att utföra beräkningarna enbart med heltal istället för decimaltal, eftersom decimaltal är mer krävande att hantera för en dator. Vad som är intressant för en utvecklare är att enkelt kunna testa och analysera hur olika genvägar och optimeringar påverkar slutledningsförmågan, för att till slut kunna framställa en modell över avvägningen mellan prestandaoptimeringar och slutledningsförmåga.

Vårt examensarbete gick ut på att skapa ett mjukvaruhjälpmedel till utvecklare som vill kunna undersöka olika typer av optimeringar för ett ANN. Programmet kallas *QPANN* och låter en användare enkelt ändra på moduler i ett ANN för att köra enbart heltalsberäkning, m.m. för att sedan se hur slutledningsförmågan förändras. *QPANN* har stor utvecklingspotential, eftersom det går att programmera egna operatorer för att testa hur det påverkar slutledningsförmågan.