TensorQuant - A Simulation Toolbox for Deep Neural Network Quantization

Dominik Marek Loroch
Fraunhofer ITWM
loroch@itwm.fhg.de

Norbert Wehn
TU Kaiserslautern
wehn@eit.uni-kl.de

Franz-Josef Pfreundt
Fraunhofer ITWM
pfreundt@itwm.fhg.de

Janis Keuper
Fraunhofer ITWM
keuper@itwm.fhg.de

Abstract

Recent research implies that training and inference of deep neural networks (DNN) can be computed with low precision numerical representations of the training/test data, weights and gradients without a general loss in accuracy. The benefit of such compact representations is twofold: they allow a significant reduction of the communication bottleneck in distributed DNN training and faster neural network implementations on hardware accelerators like FPGAs. Several quantization methods have been proposed to map the original 32-bit floating point problem to low-bit representations. While most related publications validate the proposed approach on a single DNN topology, it appears to be evident, that the optimal choice of the quantization method and number of coding bits is topology dependent. To this end, there is no general theory available, which would allow users to derive the optimal quantization during the design of a DNN topology.
In this paper, we present a quantization tool box for the *TensorFlow* framework. *TensorQuant* allows a transparent quantization simulation of existing DNN topologies during training and inference. *TensorQuant* supports generic quantization methods and allows experimental evaluation of the impact of the quantization on single layers as well as on the full topology. In a first series of experiments with *TensorQuant* , we show an analysis of fix-point quantizations of popular CNN topologies.

1 Introduction

Deep Neural Networks suffer from the big amount of data which needs to be stored or transferred during training and inference. The data is usually represented by floating point numbers, because they are the most convenient to handle on standard hardware. However, the required memory, energy and achieved throughput of hardware depend approximately linearly on the number of bits necessary to represent the data. Several publications suggest that the floating point representation is taking more resources than it would be necessary to successfully train networks and perform inference [9, 20].
There are two important use cases where smaller data representations are particularly interesting: The first one is custom hardware such as FPGAs [8] and ASICs [11], where not only data storage and transfer can be carried out in a customized format, but also the computation. The second case is distributed systems [13, 7], where the communication between the different nodes is becoming the main bottleneck. A custom data representation can largely reduce the amount of data which needs to be communicated and thus reduce energy consumption and increase throughput.
A common approach to reduce the amount of data is quantization, which is a mapping from a large, continuous or discrete set of values to a discrete, smaller set.

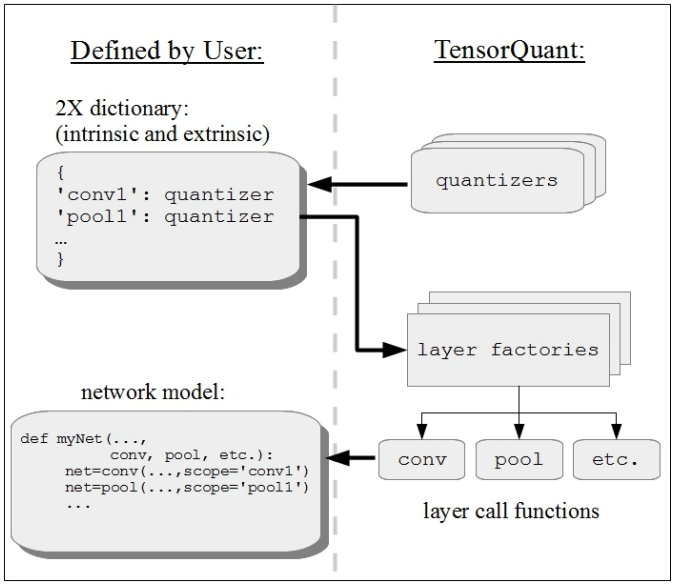

Figure 1: Overview of the *TensorQuant* toolbox.

1.1 Related Work

There are several quantization methods which can reduce the amount of stored and transferred data in neural networks. One common approach is to quantize the data with clustering, which means confining the data representation to a discrete set of values [9, 26, 6]. A special case of quantization is the fixed point representation, where all numbers of the

discrete set have the same distance to their two nearest neighbours [17, 18, 7, 12]. Fixed point numbers are a very popular representation in custom hardware like FPGAs and ASICs. An extreme case of quantization are binary [3, 20] and ternary quantization [16, 27]. Only one or two bits, respectively, are used to represent a value. Although the compression for the parameters is very high, these methods need gradients represented in floating point while training. There are various other methods like using the hashing trick [1], logarithmic quantization [19], etc.
Among these methods, the fixed point representation has gained attention because of the emergence of powerful custom hardware in datacenters, like Google's Tensor Processing Unit [11], Amazon Web Services' F1 instances and Microsoft Azure's FPGA-based cloud services.
Many publications claim that the rounding, which inevitably happens after every operation in custom hardware using the fixed point format, can be modelled with a single rounding step applied after a DNN layer has been computed in floating point precision [12, 7, 18]. The argument in [18] is that the quantization noise introduced after every rounding step can be transformed into a single source of noise at the end of the layer, since all operations are linear, if the activation function is a rectifying linear unit (ReLU). In other words, it does not matter at which point the noise level is increased, thus rounding at the end is a sufficient approximation.

1.2 Contribution

In this paper, the question whether or not the quantization noise introduced after each operation is close to a single quantization step at the end of a layer is answered by directly simulating those two cases and comparing them with each other. In addition to the location of the rounding, the rounding method is also investigated. Even simple rounding methods, such as nearest rounding and rounding down (defined in section 2.1), have an impact on the network's accuracy.
Quantization methods are often tested on simple datasets and small topologies, which can be quickly trained and run, such as LeNet and AlexNet with MNIST or CIFAR10. The results from these experiments are generalized to bigger networks and datasets. Here, the results are directly obtained from simulating big, state-of-the-art topologies, such as Inception V3 and ResNet 152 .
In order to investigate quantization in DNNs, a toolbox for *TensorFlow* called *TensorQuant* is introduced[1]. The full spectrum of functions offered by *TensorFlow* can be utilized, augmented with the ability to quantize each layer and to fully emulate fixed point format data processing. Up until now, there was no implementation which could emulate custom size fixed point computation in a common neural network simulation framework. In short, the main contributions are:

- A toolbox for *TensorFlow* , which can quantize the user's network using any user defined quantization method.
- Emulation of fixed point operations in addition to layer-wise quantization.
- A systematic investigation of the impact of the rounding location and method (the "where" and "how") on the DNN accuracy.
- A demonstration of the capabilities of the toolbox on large, state-of-the-art networks.

This paper is structured as follows: Chapter 2 introduces the used methods and terms. Extrinsic and intrinsic rounding are introduced and explained, which have a central role. Chapter 3 presents the *TensorQuant* toolbox and explains its features in detail. It gives an overview on how much effort the user needs to put into applying the toolbox to his or her own projects. The toolbox is used to investigate fixed point quantization in chapter 4. Experiments are carried out on large topologies such as Inception V3 and ResNet 152 .

2 Terms and Methods

2.1 Fixed Point Representation and Rounding:

A fixed point number (W, F) is an integer with word size W, where the F least significant bits are interpreted as the fractional portion of the number. The word size or width W is defined as the number of bits which are used to store a single numerical value. Negative numbers are saved in two's complement, thus the range of a fixed point number (W, F) is

$$[-2^{W-F-1},\, 2^{W-F-1} - 2^{-F}]. \tag{1}$$

The resolution Δ of the fixed point number is 2^{-F}.
Converting a number from floating point representation to fixed point causes loss in accuracy. If the original number does not fit into the fixed point range determined by equation 1, the usual approach is to saturate the number, that is to use the positive or negative marginal value, respectively. The fractional part is chosen using a rounding method. Commonly known is nearest rounding

$$Q_{\text{nearest}}(x) = \text{sgn}(x) \lfloor \frac{|x|}{\Delta} + 0.5 \rfloor \Delta, \tag{2}$$

[1] available at: `https://github.com/cc-hpc-itwm/TensorQuant`

where the number x is rounded to the closest element in the finite set of fixed point numbers with the resolution of Δ.
Another way is to round towards zero, that is to throw away the fractional part of the number after the given resolution:

$$Q_{\text{zero}}(x) = \text{sgn}(x)\lfloor\frac{|x|}{\Delta}\rfloor\Delta. \quad (3)$$

The easiest way to implement rounding in custom hardware is down rounding

$$Q_{\text{down}}(x) = \lfloor\frac{x}{\Delta}\rfloor\Delta, \quad (4)$$

which cuts off the binary representation of the number after as many bits as corresponds to Δ, regardless of the sign. For example, if Δ =0.25, the binary number is cut off after the second fractional bit.
A rounding method which has been investigated often in literature is stochastic rounding [7, 17]

$$Q_{\text{stochastic}}(x) = \begin{cases} \lceil\frac{x}{\Delta}\rceil\Delta & \text{if} \quad \frac{x-\lfloor x\rfloor}{\Delta} >= t_{\text{random}} \\ \lfloor\frac{x}{\Delta}\rfloor\Delta & \text{otherwise} \end{cases}, \quad (5)$$

where the number x is rounded up or down depending on the random, uniformly distributed threshold $t_{\text{random}} \in [0, 1]$. The probability to be rounded towards either of the neighbouring values increases with the proximity to that value.

2.2 Extrinsic, Intrinsic and Gradient Quantization:

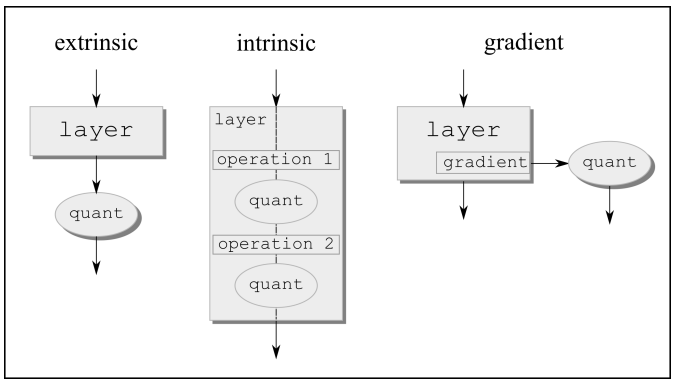

Figure 2: Overview of quantization types.

In order to handle the question of where the quantization is applied to, as it was described in the introduction, the terms "extrinsic" and "intrinsic" quantization are introduced in this paper (see figure 2 for an overview).

If quantization is applied at the end of a layer, it will be called extrinsic quantization. This is the case where a layer is computed in floating point precision, but the output is sent to the next layer in a reduced data format. An extrinsic quantization scenario can be encountered in a distributed system with many computation nodes, where the data is processed in floating-point precision within the nodes, but it is quantized before it is sent to another node in order to reduce the required bandwidth. For example, a convolution layer, which performs the operation $\circledast$ between the input x_{in} and the filter K, would be formally represented as

$$x_{\text{in}} \circledast K = Q(\sum_f \sum_k \sum_l x_{\text{in},f}(i+k, j+l) \times K_f(k,l)), \quad (6)$$

with Q the quantization function, f the number of features, k and l the coordinates within the filter and i and j the coordinates within a feature.
On the other hand, all intermediate results from every arithmetic operation within a layer can be quantized. This scenario can be found in custom hardware, where data is stored and processed in the fixed point format, thus the data is restricted to a certain precision at any point. Since this type of quantization is applied on a deeper level compared to the extrinsic one, it is called intrinsic quantization. A convolution layer for example is formally computed as

$$x_{\text{in}} \circledast K = Q(\sum_f \sum_k \sum_l Q(Q(x_{\text{in},f}(i+k, j+l)) \times Q(K_f(k,l)))). \quad (7)$$

Simulating intrinsic quantization needs more memory and time then the extrinsic case. It is only reasonable to use intrinsic quantization when trying to emulate the behaviour of hardware. Fixed point quantization is the most reasonable method to apply intrinsically, which is simply referred to as rounding in this paper.
During training, the gradient can be quantized before the update to the weights is applied [13]. Formally, this is can be described with

$$w_{\text{t+1}} = w_{\text{t}} + \eta Q(\nabla L), \quad (8)$$

with w the trained parameter at a time t, η the learning rate and ∇L the gradient of the loss function, which is to be minimized.

2.3 The Subunit Quantization Approach:

Intrinsic quantization can be unfeasible for big DNN topologies such as Inception V3 [22] and ResNet 152 [10], because it requires too much memory to run. A method to mitigate this

Listing 1: Fixed point quantizer code.

```
class FixedPointQuantizer(Quantizer_if):
    def __init__(self, fixed_size, fixed_prec):
        self.fixed_size=fixed_size
        self.fixed_prec=fixed_prec
    def quantize(self,tensor):
        return quant_kernel(tensor,self.fixed_size,
                                    self.fixed_prec)
```

problem is to apply rounding only to functional subunits of the topology. A good example for subunits are the inception modules occurring in the Inception type topologies. So instead of rounding all layers at once, only a few are rounded at the same time. There are good reasons justifying this approach. First, the accuracy loss which happens at one quantized layer cannot be regained in subsequent layers. Second, it is very likely that the various layers require different word sizes and fractional bits in order to keep the accuracy at the baseline value. Therefore, there exists one or several bottlenecks, which will determine the word size of the entire topology.
All subunits of the topology are rounded one after another and with different word sizes and fractional bits. The accuracy of the inference is recorded for each run. For each subunit, the best combination with the lowest word size and the least fractional bits is determined, for which the accuracy stays the same compared to the unquantized topology. Amongst all the best combinations, the subunit with the highest word size is identified as the bottleneck. If there are several bottleneck subunits with the same word size, the unit with the highest fractional bits is chosen as the bottleneck.

3 The TensorQuant Toolbox

The *TensorQuant* toolbox is able to quantize any neural network designed in *TensorFlow* intrinsically and extrinsically. Some changes to the user's original *TensorFlow* topology description file need to be made. Also, the user has to provide a specification of which layers shall be quantized. *TensorQuant* manages the quantization of the layers during the building and running process. An overview of the different components of *TensorQuant* is given in figure 1.

The Quantizers The core of the toolbox are the quantizer objects, which carry out the quantization of the tensors. Their simple interface takes a tensor and outputs the quantized version. An example is given in listing 1. The Quantizer

Listing 2: Example topology file (original).

```
def lenet(images, ...):
    ...
    with tf.variable_scope(scope, 'LeNet',
    [images, num_classes], ...):
    net = slim.conv2d(images, 32, [5, 5], scope='conv1')
    net = slim.max_pool2d(net, [2, 2], 2, scope='pool1')
    net = slim.conv2d(net, 64, [5, 5], scope='conv2')
    ...
    logits = slim.fully_connected(net, num_classes,
    activation_fn=None, scope='fc4')
    ...
    return logits, end_points
```

Listing 3: Example topology file (for quantization).

```
def lenet(images, ...,
    conv2d=slim.conv2d,  # added layer types
    max_pool2d=slim.max_pool2d,
    fully_connected = slim.fully_connected):
    ...
    with tf.variable_scope(scope, 'LeNet',
    [images, num_classes],...):
    # removed slim. before layer calls
    net = conv2d(images, 32, [5, 5], scope='conv1')
    net = max_pool2d(net, [2, 2], 2, scope='pool1')
    net = conv2d(net, 64, [5, 5], scope='conv2')
    ...
    logits = fully_connected(net, num_classes,
    activation_fn=None, scope='fc4')
    ...
    return logits, end_points
```

interface forces a "quantize" method, which invokes the quantization kernel. The quantization process is carried out by the kernel, which is written in C++. It is possible to write the quantizers entirely in Python, although in the case of intrinsic quantization, this utilizes prohibitively many resources.

Changes to the User's Topology File Quantizers can be applied to any node of the topology, but it would be very laborious to assign them by hand. For the *TensorFlow Slim* framework, a series of convenience functions is implemented in *TensorQuant* , which automate the application of quantizers to the topology. An example of a file describing a topology is given in listing 2. The changes to the topology file when prepared for quantization are shown in listing 3.

Assigning Layers for Quantization The locations where quantization is applied are controlled with the *TensorFlow* variable namespaces. The user has to specify the entire variable name for a single, or matching substrings for a set of layers where the quantization should be applied to. For example, in listing 3 there is the variable scope "LeNet", which contains the layers "conv1", "pool1" and so on. The first convolution layer can be accessed with the identifier "LeNet/conv1". All

Listing 4: Pseudocode for layer factory.

```
def layer_factory(intrinsic_dictionary,
                  extrinsic_dictionary):
  def func(*args,**kwargs):
    current_layer=get_current_layer()
    if current_layer in intrinsic_dictionary:
        kwargs['quantizer']=
              intrinsic_dictionary[current_layer]
        net=quantized_conv(*args,**kwargs)
    else:
        net=standard_conv(*args,**kwargs)

    if current_layer in extrinsic_dictionary:
        return extrinsic_dictionary[current_layer].
                                        quantize(net)
    else:
        return net
  return func
```

layers in the "LeNet" scope can be accessed at once by the identifier "LeNet".
The user specifies two dictionaries, one for intrinsic and one for extrinsic rounding. The keys are the identifiers, and the values are quantizer objects. The dictionaries are passed to the layer factories.

The Layer Factories The layers are built by factory functions. Each layer type has its own factory that returns a function, which has the same interface as the standard *TensorFlow* layers. The factory takes two dictionaries as input arguments, one for intrinsic and extrinsic quantization. The pseudocode of a factory function is given in listing 4.

Implementation of Extrinsic and Intrinsic Quantization Implementing extrinsic quantization is straight forward, because the quantization is applied directly to the layer output. The quantization is independent of what is computed in the layer, thus it can be applied directly to all layer types.
In the intrinsic case, however, there is no such straight forward approach since the specific operation of the layer needs to be considered. Unfortunately, there is no other way than to re-implement the layer type, where quantization is applied to the desired calculation steps.
The aim is to use the same quantizer objects as previously shown in listing 1, thus no additional C++ kernels need to be written. In the re-design, the standard *TensorFlow* framework is used as much as possible. The downside is, that a lot of intermediate results need to be stored in tensors, which means an increase in required memory. To mitigate this problem, the batch size can be reduced.
Adding extrinsic quantization to the model increases the time to build the model by less than 2 % and the runtime by 20 % (estimated on Inception V1). Intrinsic quantization increases the build time by a factor of approximately $\times 70$ and the runtime by $\times 20$.

Some layers contain trained parameters, e.g. filter weights. For a proper representation of fixed point operations, those values are automatically quantized with the intrinsic quantizer before passed to the calculation. Adding quantization does not interfere with the *TensorFlow* namespaces. Therefore, the model parameters can be loaded from save files where the model was trained without quantization.
Extrinsic and gradient quantization are independent from the *TensorFlow* version. Intrinsic quantization needs to re-implement layers, therefore it can be incompatible to other versions than 1.0 and 1.2.

Gradient Quantization The gradient quantization as described in equation 8 is implemented easily. In the file controlling the training, the gradients are intercepted and the quantizer is applied, before they are passed to the optimizer function.

4 Experiments

The *TensorQuant* toolbox is used to apply fixed point quantization to DNNs. The simulations are focused on popular CNN topologies, such as Inception V1 (GoogleNet) [21], Inception V3 [22], ResNet 50 and ResNet 152 [10]. The networks are trained on ImageNet 2012 [4, 23]. For reference, we also provide results for LeNet [15] with the MNIST dataset. In the learning experiments, AlexNet [14] is trained on ImageNet. The networks and trained parameters are taken from the *TensorFlow* model library[25], specifically the *Slim GitHub* web page [24].
The main metric is the test accuracy. An unquantized version of each topology serves as the baseline, to which all accuracies are given to as relative values. About 1% of the validation set is used for inference, simply to perform design space exploration in reasonable time, especially in the case of intrinsic quantization. Using a smaller validation set is valid, because it is only interesting whether or not the baseline accuracy is reached, plus quantization without any form of retraining cannot improve accuracy.

4.1 Inference with Quantization

The following experiments perform inference on pre-trained networks. The files with the pre-trained parameters are downloaded from the *Slim* web page [24].

4.1.1 Rounding Method

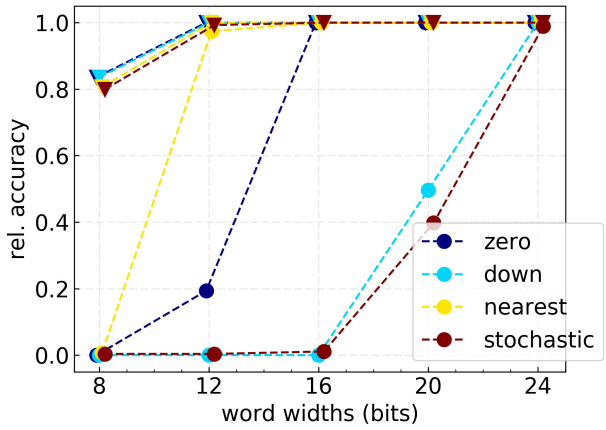

Figure 3: Zero, down, nearest and stochastic rounding methods applied to the Inception V1 topology. Dots mark the relative accuracy when the rounding is applied intrinsically and triangles mark the extrinsic rounding.

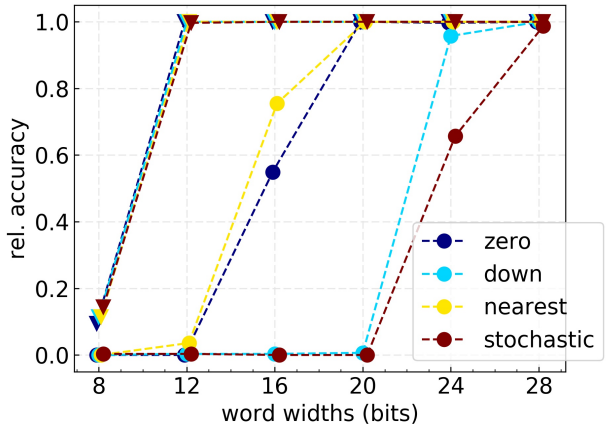

Figure 4: Rounding methods applied to the ResNet 50 topology, as in figure 3.

The influence of the different rounding methods introduced in section 2.1 is investigated on the Inception V1 and ResNet 50 topologies. Figures 3 and 4 show the relative accuracy against the word width used for rounding. The fractional bits are half of the word size. Intrinsic and extrinsic rounding are plotted in the same figure with different markers.
Extrinsic rounding is almost not affected by the rounding method. The only deviation from the baseline accuracy is coming from the word size. All rounding methods lead to equally good accuracies.
For the intrinsic case, the choice of the rounding method has an impact on the accuracy. Using down or stochastic rounding in the Inception V1 topology doubles the required word size compared to nearest rounding, whereas zero rounding is between those. For the ResNet 50 topology, nearest and zero rounding lead to similar accuracies, but down ans stochastic rounding are still worse than the other two methods.
Nearest rounding is the method which requires the least amount of bits, therefore it is the best method of the investigated ones. It is used in the following experiments. Down and stochastic rounding, on the other hand, are the most demanding methods.

Table 1: Comparison of rounding all layers at once (all) against the subunit method (single). The entries are (relative accuracy, fractional bits).

Topology	8 bit	12 bit	16 bit	20 bit
Inception V1 (all)	0.01, 4	0.99, 6	1.00, 8	1.00, 8
Inception V1 (single)	0.46, 2	1.00, 8	1.00, 8	1.00, 8
ResNet 50 (all)	0.00, 0	0.04, 6	0.91, 6	1.00, 10
ResNet 50 (single)	0.02, 2	0.54, 4	0.99, 6	1.00, 10

Table 2: Comparison of different topologies and word sizes for intrinsic rounding. The entries are (relative accuracy, fractional bits).

Topology	8 bit	10 bit	12 bit	14 bit	16 bit
Inception V1	0.46, 2	0.98, 5	1.00, 8	1.00, 9	1.00, 8
Inception V3	0.08, 2	0.92, 7	0.97, 6	0.99, 7	0.98, 6
ResNet 50	0.02, 2	0.14, 3	0.54, 4	0.93, 5	0.99, 6
ResNet 152	0.03, 2	0.16, 3	0.55, 4	0.95, 5	1.00, 6

4.1.2 Intrinsic vs. Extrinsic Rounding

For intrinsic rounding, the subunit approach explained in section 2.3 is used. Table 1 compares the accuracies from rounding all layers at once against the subunit approach for Inception V1 and ResNet 50 . If the relative accuracy is very close to 100 % in the subunit approach, then the method is equivalent to rounding the entire network at once, as the results are the same.
Intrinsic rounding is applied to DNN topologies in table 2.

Table 3: Comparison of different topologies and word sizes for extrinsic rounding, similar to table 2.

Topology	4 bit	6 bit	8 bit	10 bit	12 bit
Inception V1	0.32, 0	0.76, 2	0.98, 2	1.00, 4	1.00, 4
Inception V3	0.08, 0	0.93, 2	0.97, 2	0.99, 4	1.00, 6
ResNet 50	0.06, 0	0.62, 0	0.99, 2	1.00, 4	1.00, 4
ResNet 152	0.07, 0	0.55, 0	1.00, 2	1.00, 2	1.00, 2

Table 4: LeNet, Inception V1 and ResNet 50 rounded intrinsically for different word sizes. The entries are (word size, fractional bits).

Topology	4 bit	8 bit	12 bit	16 bit	20 bit
LeNet	0.40, 3	0.90, 4	1.00, 6	1.00, 8	1.00, 8
Inception V1	-	0.01, 4	0.99, 6	1.00, 8	1.00, 8
ResNet 50	-	0.00, 0	0.04, 6	0.91, 6	1.00, 10

The word size is fixed in each column. The entries show the maximum achievable relative accuracy and the used fractional bits.
Inception type topologies require less word size than ResNet type ones to achieve full benchmark accuracy. 12 bits are enough for Inception, whereas ResNets need 16 bits. This is a hint that the different topology types have different bottlenecks, even though they use the same layer types. The bottlenecks will be located later in section 4.1.3.
The conclusion from comparing the Inception and ResNet type topologies amongst themselves is that the word size does not depend on the depth of the topology. This can be attributed to the batch normalization layers, which normalize the activations before they leave a layer. The range of the activation values is kept the same, therefore fixed point quantization is working well.

The results from rounding the topologies **extrinsically** are shown in table 3, which is structured as in the intrinsic case. The subunit method is not needed here, because extrinsic rounding does not require as much memory to run, therefore all layers can be quantized at once without problems. Notice that the word sizes in the columns are different. Extrinsic rounding achieves baseline accuracy with less word size than in the intrinsic case. 8 to 10 bits are already enough to stay close to full accuracy, regardless of the topology type. Also, the portion of fractional bits is lower than in the intrinsic case, meaning the output values of the layers can be transferred in low precision.

Last, LeNet is compared to the Inception V1 and ResNet 50 topologies. All networks are quantized completely, so the subunit method is not used. In table 4, the results are presented as in table 2 before. However, LeNet was trained on MNIST and the other topologies on ImageNet.
At 8 bits word width, LeNet achieves 90 % relative accuracy, but the other topologies are close to zero. As from the previous results, ResNet 50 needs more word width than Inception V1 . This illustrates that even though all three topologies are CNNs and utilize the same layer types, results cannot be generalized from one topology to another for the intrinsic case.

4.1.3 Layerwise Intrinsic Rounding

After seeing the results from the subunit approach in table 2, the actual per subunit requirements regarding word size and fractional bits are shown in the tables 5 and 6 for the Inception V3 and ResNet 50 topology, respectively. Each column represents a threshold for the relative accuracy. The entries state the minimum required word size and fractional bits to be above the threshold.

From this perspective, one can identify the bottleneck

Table 5: Subunits and relative accuracies for the Inception V3 topology. The entries show the required word size and fractional bits to achieve the relative accuracy of the column.

Subunit	100%	99%	90%
Conv2d_1a_3x3	24,6	16,12	12,6
Conv2d_2a_3x3	16,6	12,6	8,2
Conv2d_2b_3x3	12,6	12,6	8,4
MaxPool_3a_3x3	8,4	8,4	8,0
Conv2d_3b_1x1	12,6	12,6	8,4
Conv2d_4a_3x3	8,4	8,4	8,4
MaxPool_5a_3x3	8,0	8,0	8,0
Mixed_5b	8,4	8,4	8,4
Mixed_5c	12,4	12,4	8,4
Mixed_5d	16,10	16,10	8,4
Mixed_6a	8,4	8,4	8,4
Mixed_6b	12,6	12,6	8,4
Mixed_6c	12,6	12,6	8,6
Mixed_6d	12,8	12,8	12,6
Mixed_6e	12,8	12,8	12,6
Mixed_7a	24,20	16,8	8,4
Mixed_7b	8,6	8,6	8,6
Mixed_7c	8,6	8,6	8,6
AuxLogits	8,0	8,0	8,0
PreLogits	8,0	8,0	8,0
Logits	12,4	12,4	12,4

Table 6: Subunits and relative accuracies for the ResNet 50 topology, as in table 5.

Subunit	100%	99%	90%
resnet_v1_50/conv1	20,8	16,6	16,6
block1	16,10	12,6	12,6
block2	16,10	12,6	12,6
block3	12,8	12,8	12,6
block4	16,10	16,8	12,6
logits	16,8	12,4	12,4

Table 7: Relative accuracy achieved for LeNet when the gradient is quantized with different word sizes (W) and fractional bits (F).

(W,F)	2,1	4,2	6,3	8,4	10,5	12,6
rel. accuracy	0.87	0.97	0.99	0.99	1.00	1.00

units and how demanding they are. Most units in both Inception and ResNet topologies have the same requirement of 12 bits word width and 6 bits fractional part, because most of the layers within a subunit are convolution layers. However, the bottleneck layers of the network determine the required word width of the entire hardware architecture. The very first layer needs a high word size in both topologies. In Inception V3 , there are bottlenecks appearing in the middle of table 5. There is no general rule that bottlenecks appear at the beginning or the end of a topology.

4.2 Training with Quantization

For the case of distributed systems, it is more interesting to see if and how well a topology can be trained when the gradients, which are communicated between the computation nodes, are quantized. As before, the quantization method is fixed point with nearest rounding. The accuracies are relative to a network trained with exactly the same set up, but without gradient quantization.
Table 7 shows the relative test accuracies for different word size and fractional bits combinations for LeNet, and table 8 for AlexNet. The gradient can be quantized quite rigorously in LeNet. Only 4 bits are sufficient to train LeNet to 97 % relative accuracy on the MNIST dataset. AlexNet, on the other hand, needs at least 12 fractional bits to be trained to 92 %. The comparison between the two topologies shows that there is no point in generalizing results from simple topoligies like LeNet to larger and deeper networks.

Table 8: Relative accuracy achieved for AlexNet when the gradient is quantized with different word sizes, similar to table 7.

(W,F)	16,8	16,11	16,12	16,13	16,14	32,16
rel. accuracy	0.004	0.847	0.926	0.997	0.745	1.0

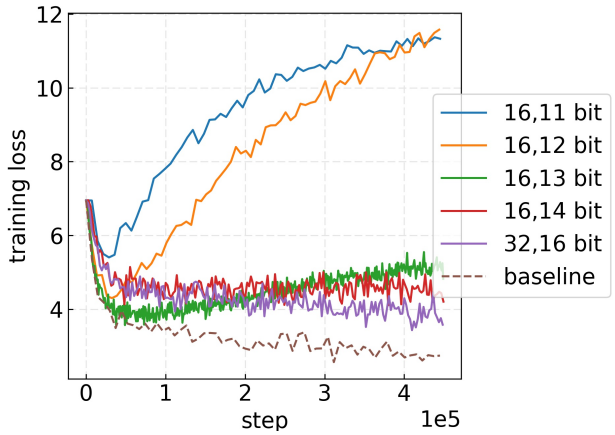

Figure 5: Training loss versus number of steps for gradient quantization and the AlexNet topology.

An interesting observation can be made when looking at the training loss for AlexNet. Figure 5 shows the training loss against the number of steps for the AlexNet topology, trained with the word sizes of table 8. The training loss is going up after an initial phase of descend for 11 and 12 fractional bits. Despite the training loss going up, the test accuracy is high. The reason for that is that the used function for the training loss comprises of the cross entropy and the L2-regularizer. The quantization renders the L2-regularizer ineffective during the computation of the gradient in case of low fractional bits, so only the cross entropy is minimized. Since the weights are not bound by the regularizer, their magnitudes can grow freely. However, then the regularizer contributes a high value to the overall training loss.

5 Discussion

A paper which has investigated the fixed point data format with similar thoroughness is [12]. The most complex network they investigated is Inception V1 . They suggest using at most 14 bits word length with 2 bits fractional part. This result somewhat coincides with our findings for extrinsic rounding, where 10 bits word size and 4 fractional bits are sufficient

for that topology. The difference in the results could come from the used framework and model parameters.
Designers of custom hardware have been using 16 bit word size for implementations of their topologies [8, 5, 2]. Our results suggest to be careful, because there is no general word size which guarantees to be sufficient for all topologies. But for the investigated cases, 16 bit is large enough for all layers. The results from the training coincide with [7], although they used extrinsic rounding, instead. The fractional part needs to be relatively high, whereas the integer part is not needed. The increase of the training loss when using 16 bit nearest rounding during training (figure 5) was also observed by [7], despite using extrinsic quantization. In their paper, stochastic rounding makes the training loss converge normally.

6 Conclusion

The *TensorQuant* toolbox allows to explore different quantization methods with the *TensorFlow* framework. The unique feature is that the fixed point data format can be emulated to the arithmetic level, thus allowing for the closest similarity to custom hardware yet presented in any framework.
The most important result from the experiments is that in the case of intrinsic rounding, the word size is more demanding than it was previously thought. Intrinsic rounding is very sensitive to the rounding method. It is not possible to generalize the required word width from one topology to another, as they have different bottleneck layers. The results from the training section show that gradient quantization allows even less to generalize between topologies.
It is planned to extend the *TensorQuant* toolbox much further, especially the functionality related to training. The goal is to fully emulate learning on a distributed system comprising of custom hardware, thus using the fixed point data format. Other quantization methods will be implemented to study possible strategies to reduce bandwidth in distributed systems. Also, *TensorQuant* will be applied to other topologies like recurrent neural networks. It is expected that the requirements regarding the data representation will differ from CNNs.

Acknowledgments

The authors thank the Leibniz Supercomputing Centre for providing compute time on their DGX-1 system. We also gratefully acknowledge the support of the NVIDIA Corporation with the donation of the Titan X Pascal GPU used for this research.

References

[1] Chen, W., Wilson, J., Tyree, S., Weinberger, K., and Chen, Y. Compressing neural networks with the hashing trick. In *International Conference on Machine Learning* (2015), pp. 2285–2294.

[2] Chen, Y.-H., Krishna, T., Emer, J. S., and Sze, V. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits 52*, 1 (2017), 127–138.

[3] Courbariaux, M., Hubara, I., Soudry, D., El-Yaniv, R., and Bengio, Y. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830* (2016).

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

[5] Gokhale, V., Zaidy, A., Chang, A. X. M., and Culurciello, E. Snowflake: A model agnostic accelerator for deep convolutional neural networks. *arXiv preprint arXiv:1708.02579* (2017).

[6] Gong, Y., Liu, L., Yang, M., and Bourdev, L. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115* (2014).

[7] Gupta, S., Agrawal, A., Gopalakrishnan, K., and Narayanan, P. Deep learning with limited numerical precision. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)* (2015), pp. 1737–1746.

[8] Han, S., Liu, X., Mao, H., Pu, J., Pedram, A., Horowitz, M. A., and Dally, W. J. Eie: efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture* (2016), IEEE Press, pp. 243–254.

[9] Han, S., Mao, H., and Dally, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[10] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016), pp. 770–778.

[11] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al. In-datacenter performance analysis of a tensor processing unit. *arXiv preprint arXiv:1704.04760* (2017).

[12] Judd, P., Albericio, J., Hetherington, T., Aamodt, T., Jerger, N. E., Urtasun, R., and Moshovos, A. Reduced-precision strategies for bounded memory in deep neural nets. *arXiv preprint arXiv:1511.05236* (2015).

[13] Keuper, J., and Pfreundt, F.-J. Distributed training of deep neural networks: Theoretical and practical limits of parallel scalability. In *Proceedings of the Workshop on Machine Learning in High Performance Computing Environments* (Piscataway, NJ, USA, 2016), MLHPC '16, IEEE Press, pp. 19–26.

[14] Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (2012), pp. 1097–1105.

[15] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE 86*, 11 (1998), 2278–2324.

[16] Li, F., Zhang, B., and Liu, B. Ternary weight networks. *arXiv preprint arXiv:1605.04711* (2016).

[17] Li, H., De, S., Xu, Z., Studer, C., Samet, H., and Goldstein, T. Training quantized nets: A deeper understanding. *arXiv preprint arXiv:1706.02379* (2017).

[18] Lin, D., Talathi, S., and Annapureddy, S. Fixed point quantization of deep convolutional networks. In *International Conference on Machine Learning* (2016), pp. 2849–2858.

[19] Miyashita, D., Lee, E. H., and Murmann, B. Convolutional neural networks using logarithmic data representation. *arXiv preprint arXiv:1603.01025* (2016).

[20] Rastegari, M., Ordonez, V., Redmon, J., and Farhadi, A. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision* (2016), Springer, pp. 525–542.

[21] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2015), pp. 1–9.

[22] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2016), pp. 2818–2826.

[23] Website. Imagenet. `http://www.image-net.org/challenges/LSVRC/2012/`. Access: 28.08.2017.

[24] Website. Slim. `https://github.com/tensorflow/models/tree/master/slim`. Access: 28.08.2017.

[25] Website. Tensorflow models. `https://github.com/tensorflow/models`. Access: 28.08.2017.

[26] Zhou, A., Yao, A., Guo, Y., Xu, L., and Chen, Y. Incremental network quantization: Towards lossless cnns with low-precision weights. *arXiv preprint arXiv:1702.03044* (2017).

[27] Zhu, C., Han, S., Mao, H., and Dally, W. J. Trained ternary quantization. *arXiv preprint arXiv:1612.01064* (2016).