TECHNISCHE UNIVERSITÄT WIEN

Christian Doppler Laboratory
Embedded Machine Learning

A TECHNICAL REPORT ON

FPGA optimized dynamic post-training Quantization of Tiny-YoloV3

Bachelor programme Computer Engineering 033 535

by

Dominik Dallinger

01529357

Supervisor(s):

Projektass. Dipl.-Ing. Matthias Wess

Vienna, Austria

May 2021

Abstract

Nowadays Deep Neural Networks (DNNs) are getting more and more a part of our everyday life. DNNs are often used for sophisticated tasks such as speech recognition or computer vision. Recent state-of-the-art DNNs are usually getting bigger and bigger, which require a significant memory bandwidth, memory storage and more computational power. This development makes it difficult to fit these state-of-the-art DNNs on Embedded Devices. Various processes have been developed to counteract this development. On the one hand there are custom ASIC designs which accelerate the DNNs, on the other hand there are procedures that change the network. The most common procedure to optimize a neural network for Embedded Devices is quantization. Quantization is a technique where, typically, a 32bit floating point network is transformed to a different data type. The most common quantization is a transformation to 8bit fixed point integer. There are two main techniques to quantize a Network.

- **Quantized-Aware-Training**: A model definition and training data-set is needed to alter the back-propagation algorithm to include the quantization error and retrain the whole network. Therefore also information about the training algorithm is needed. A big disadvantage of this technique is, in most cases, the very long training time of complex data-sets.
- **Post-Training Quantization**: For this type of quantization, there is no need for information about the training algorithms or the whole data-set, only a small calibration data set and the model definition is needed. The model will be quantized through stochastical or statistical methods. Since there is no retraining needed this method works only in just a few minutes computing time.

In this work different techniques for Post-Training Quantization DNNs are compared, approaches for realization on real hardware are being discussed and a technique for Field Programmable Gate Array (FPGA) optimization is proposed.

Kurzfassung

Heutzutage werden DNNs mehr und mehr zu einem Teil unseres täglichen Lebens. DNNs werden oft für anspruchsvolle Aufgaben wie z.B. Spracherkennung oder Computer Vision verwendet. DNNs auf dem neuesten Stand der Technik werden in der Regel immer größer, was eine erheblich größere Bandbreite, einen größeren Speicherplatz und mehr Rechenleistung erfordert. Diese Entwicklung macht es schwierig, diese State-of-the-Art DNNs auf integrierten Geräten unterzubringen. Es wurden verschiedene Verfahren entwickelt, um dieser Entwicklung entgegenzuwirken. Zum einen gibt es eigene ASIC-Designs, die DNNs beschleunigen, zum anderen gibt es Verfahren, die das Netzwerk verändern. Das gängigste Verfahren zur Optimierung eines neuronalen Netzes für Embedded Devices ist die Quantisierung. Die Quantisierung ist eine Technik, bei der typischerweise ein 32-Bit-Fließkommanetzwerk in einen anderen Datentyp transformiert wird. Die gebräuchlichste Quantisierung ist eine Transformation in eine 8-Bit-Festkommazahl. Es gibt zwei Haupttechniken zur Quantisierung eines Netzwerks:

- **Quantized-Aware-Training**: Es werden eine Modelldefinition und ein Trainingsdatensatz benötigt, um den Backpropagation-Algorithmus so zu ändern, dass er den Quantisierungsfehler einbezieht und das gesamte Netzwerk neu trainiert. Daher werden auch Informationen über den Trainingsalgorithmus benötigt. Ein großer Nachteil dieser Technik ist die zum Teil sehr lange Trainingszeit bei komplexen Datensätzen.
- **Post-Training Quantization**: Für diese Art der Quantisierung werden keine Informationen über die Trainingsalgorithmen oder den gesamten Datensatz benötigt, sondern nur ein kleiner Kalibrierungsdatensatz und die Modelldefinition. Das Modell wird durch stochastische oder statistische Methoden quantisiert. Da kein Nachtrainieren erforderlich ist, lässt sich dieses Optimierungsverfahren in nur wenigen Minuten ausführen.

In dieser Arbeit werden verschiedene Techniken zur Post-Training-Quantisierung von DNNs verglichen, Ansätze zur Realisierung auf realer Hardware diskutiert und eine Technik zur FPGA-Optimierung vorgeschlagen.

Contents

List of Tables

List of Figures

Acronyms

AI Artificial Intelligence. 1

BN Batch Normalization. 3, 9, 10, 18, 29

DNN Deep Neural Network. iii, iv, 1, 5, 18

FPGA Field Programmable Gate Array. iii, iv, 2, 5, 9, 17, 22, 24, 25

mAP Mean Average Precision. 22, 23

QNN Quantized Neural Network. 2, 3, 5

YOLO You Only Look Once. 18

YOLOv3 You Only Look Once Version 3. v, vii, ix, 2, 11, 12, 13, 14, 15, 16, 17, 18, 19, 21, 22, 23, 24, 25

Chapter 1

Introduction

Deep Neural Networks (DNNs) are nowadays state-of-the-art when it comes to computer vision and Artificial Intelligence (AI) since Alex Krizhevsky [1] created the foundation for modern AI applications. However these DNNs need large amounts of memory bandwidth, memory storage and computational power. This characteristic makes it very unsuitable for both power limited and computational power limited embedded devices. One approach to solve this problem are custom ASIC designs or external low power accelerators for DNNs. But with extra devices the costs increase. Therefore, there is a great need for techniques to optimize the DNNs that reduce size, computing power and thus power consumption. A solution to this problem is quantization. As a result, the proposed solution is a tradeoff between accuracy and computational effort. Quantization transforms the floating-point network definition to a more suitable data-type like 8bit integer. Defined by Nagel and Baalen et al. [2] Quantization can be split into four levels.

- **Level 1** Calibration data-set nor retraining is required. Only the model definition, weights and bias are transformed.
- **Level 2** Quantization needs a calibration data-set to gather statistics to minimize the quantization error of weight and bias. No retraining is needed. The model definition will be also transformed.
- **Level 3 & 4** Requires full data set for training and testing.The model has to be retrained. This two levels are also known as Quantized aware training. Due to the fact that this work deals only with level one and two, this topic will not be discussed in detail here.

A standard solution for level 3 & 4 is quantized-aware-training as shown by Jacob et al. [3], where the weights and bias are quantized during each training epoch to minimize the quantization error. A continuation of this work is a paper by Fan et al. [4], where the approach is extended to approch beyond 8bit fixed-point quantization with a further development during the forwarding method. Kim et al. [5]

introduced an approach for a better 8bit fixed-point quantization through statistic and stochastic methods. And a Field Programmable Gate Array (FPGA) optimized approach as seen by Jain et al. [6] where the training algorithm is adapted to make it optimal for hardware implementations. The quantizer is constrained to use a power of 2 scaling methods for adjustment and minimizes the quantization error through retraining. This thesis is about the quantization of level 1 & level 2 with focus on optimization for execution on an FPGA.

1.1 Problem Statement

Most integrated devices have a very limited power capacity, therefore it is advantageous to use networks that require only a very low power. To fit these requirements for object detection networks a low power state-of-the-art realtime object detection network named Tiny You Only Look Once Version 3 (YOLOv3) is used and quantized to a FPGA optimized Quantized Neural Network (QNN). Recent research has shown that quantization is mostly done for image detection networks and not for Real-Time Object Detection neural networks. For this reason this thesis deals with the FPGA optimized quantization of Tiny YOLOv3 and compares different quantization methods.

Chapter 2

Related Work

2.1 Surveys

In the whitepaper from Krishnamorrthi [7] a comparison between symmetric and asymmetric quantization is done. Moreover he compares the impact of Batch Normalization (BN) to quantization and makes a comparison on different quantization types and bit widths. He shows that he can reduce a network by a factor of 4 by quantizing weights to 8bit with a accuracy drop of about 2%. As a recommendation, he states, for hardware optimized networks, a per channel quantization of weights and activations is preferable.
In the survey and tutorial from Sze et al. [8] a overview about different types of neural networks, frameworks and implementations on hardware is given. Moreover he shows different hardware platforms that are used to process a QNN and a discussion about mixed signal circuits and new memory technologies for neural networks to optimized throughput and energy consumption.
The survey of Guo [9] gives an background on QNNs. He compares different quantization techniques and discusses these implementations.

2.2 VitisAI

VitisAI [1] is a framework from Xilinx which allows to perform several optimization algorithms on neural networks. One part of these algorithms is quantization. VitisAI provides a framework for post training with Level 2 quantization. It uses the quantization optimization algorithms of [2] and [10]. For adjustment after each layer a $x \quad 2^n$ adjustment is done as described in section 3.6.

[1] VitisAI `https://github.com/Xilinx/Vitis-AI`

2.3 Level 1 Quantization

A very good example for Level 1 Quantization is shown by Nagel et al. [2] where a data-free quantization method for 8bit fixed-point is introduced. Their approach is about weight equalization and Bias correction which gives a significant boost in accuracy compared to a non optimized quantization. The great advantage of this method is that it does not require any information about the training algorithm or the dataset. The quantization of the network is as simple as an API call. For common networks like MobileNet they achieve state-of-the-art quantized model performance.

2.4 Level 2 Quantization

Banner et al. [11] introduces a level 2 quantization with the first practical 4bit post training quantization approach. They suggest three complementary methods to minimize the quantization error. These three methods are analytical clipping for integer quantization, per-channel bit allocation an bias correction. With a combination of these methods they achieve a significant improvement for 4bit quantization in an image detection net. The knowledge is used in an optimization tool by Intel Labs [2]

[2] IntelLabs Distiller `https://github.com/IntelLabs/distiller`

Chapter 3

Quantization

QNNs are a subset of DNNs which have the particularity to use other data-types instead of floating point 32bit for weight, bias and activation representation. The goal of quantization is to compact the models and optimize the computational effort without accuracy or performance degradation. The arithmetic operations in QNNs are typically bitwise operations that match the respective target hardware. For example, in an FPGA we choose fixed-point arithmetic to replace the inefficient floating-point arithmetic, which often requires 100 as many gates compared to fixed-point arithmetic.

3.1 Fixed-Point Quantization

Fixed-point quantization can be split into two subsections.

Asymmetric Quantization In asymmetric quantization the min/max value of the float range is not directly mapped to the integer range, in order to prevent unused areas in the number range the zero point of the float range is moved by using a quantization bias (or offset) in addition to the scale factor.

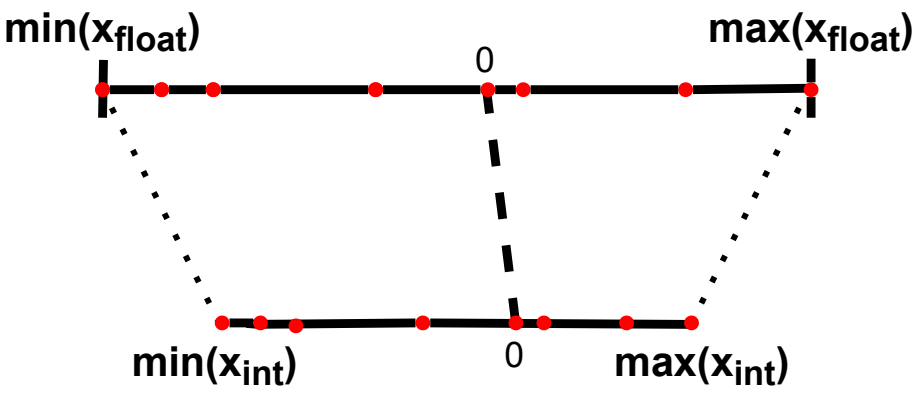


Figure 3.1: Asymmetric Quantization

Symmetric Quantization In symmetric quantization the absolute max value of the float range is directly mapped to the fixed-point range instead of the exact min/max of the float range. The quantization bias is zero, this means the zero point is directly mapped to the fixed-point range. To represent these mapping from floating point to fixed point a scaling factor is needed.

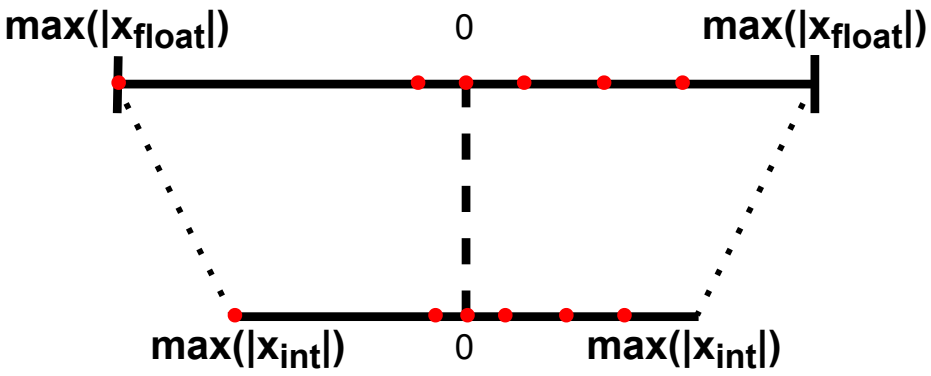


Figure 3.2: Symmetric Quantization

Formula 3.1 q_x defines this scaling factor, which is calculated with the absolute maximum value of the floating point range. 2^n - 1 describes with n as bit width the range of the fixed-point data type.

$$q_x = \frac{(2^n \quad 1) = 2}{\max(abs(x_f))} \tag{3.1}$$

To receive the actual value in the integer range we have to multiply the scaling factor to the associated float value and round i to a fixed point value.

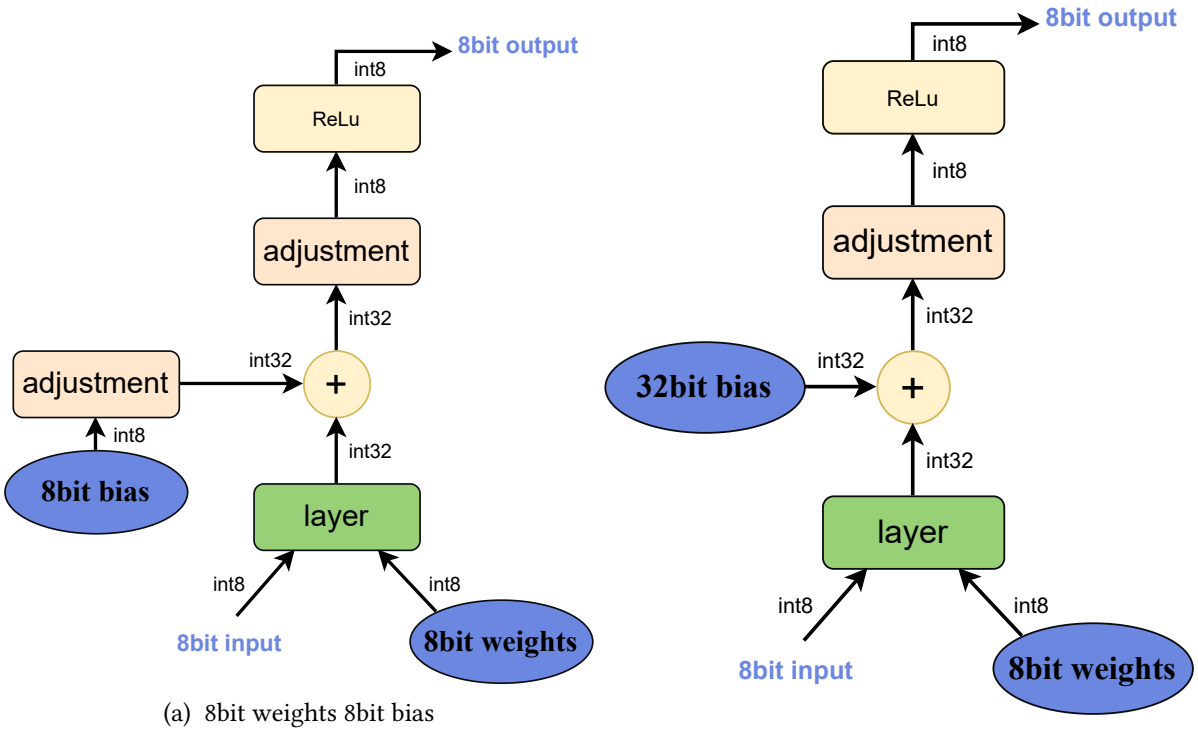
$$x_q = \text{round}(q_x x_f) \tag{3.2}$$

As an example a convolutional layer is considered as in formula 3.3. Here both weights and bias are quantized with formula 3.2 to 8bit. Since this calculation generates an output of maximum 32bit an adjustment has to be done to shorten the output to 8bit again. Furthermore the bias value has to be adapted to 32bit too with an additional factor as seen on figure 3.3b. This factor is called adjustment factor and is typically a floating point value $adjustment < 0$.

$$\text{round} \quad \frac{q_y}{q_x q_w} \quad \text{X} \quad x_q w_q + \frac{q_x q_w}{q_b} b_q \tag{3.3}$$

An other approach is to scale the bias directly with 32bit to avoid the additional adjustment as seen on figure 3.3a. Formula 3.5 shows how to quantize the bias to fit into 32bit floating point. The convolutional layer is considered as in formula 3.4.

Figure 3.3: Quantization schematic of convolutional layer



(a) 8bit weights 8bit bias

(b) 8bit weights 32bit bias

$$\text{round}\left(\frac{q_y}{q_x q_w}\right) \times x_q w_q + b_q \tag{3.4}$$

$$b_q = b_f \cdot (q_w \cdot q_x) \cdot \frac{q_b}{q_b} \tag{3.5}$$

3.2 Adjustment factor

The previous described adjustment factor has to be added after every layer which has been quantized. Observations have shown that this value is typically $adjustment < 0$, which makes it very unsuitable for hardware oriented solutions, because we originally wanted to depart from the floating point calculation. There are several algorithms to solve this problem.

3.2.1 Floating point adjustment

The easiest way is to simply ignore the problem and accept the extra computational effort as seen in formula 3.6.

$$y_{int8} = y_{int32} \cdot adjustment \tag{3.6}$$

3.2.2 Integer multiplication and shift

Another approach is to split the problem into two sections. The first part is a integer multiplication and the second a shift operation to match the right range as seen in formula 3.7.

$$\begin{aligned} y_{int} &= y_{int32} \quad adjustment_{int} \\ y_{int8} &= y_{int} << adjustment_{shift} \end{aligned} \tag{3.7}$$

The process of finding these values is shown in the pseudocode.

```
function findshiftscale(adjustment):

    e = ceil(log2(adjustment)
    x = 1
    approx = x  2^e
    delta = adjustment   approx
    oldloss = delta^2

    while:
        approx = x  2^e
        delta = adjustment   approx
        loss = delta^2

        if loss < oldloss and delta > 0
            scale lifo append x
            shift lifo append e

        oldloss = loss

        if delta < 0  # approximation to big - make it smaller
            e = e  1
            x = x  2
            x = x  1
        else
            x = x + 1
```

```
if x > 2^n or e < 40
    return shift and scale lifo second last
```

3.2.3 Power of 2 adjustment

Another step in that direction is to omit the integer multiplication value. This results in just an bit shifting operation which makes it very suitable for an embedded device like FPGA's. As seen in formula 3.8 only one operation has to be done to scale the value back to the targeted range. This makes it very efficient for FPGA's because this operator has the least operational power.

$$y_{int8} = y_{int32} << adjustment_{shift} \tag{3.8}$$

The process of finding the shifting value is shown in the pseudocode.

```
function findshift(adjustment):
    shift = nearestround(log2(adjustment))
    return shift
```

In order to apply this method, an important step in the quantization is necessary to avoid an high quantization error when calculating the shift value. This topic will be discussed further in chapter 3.6.

3.3 Batch Normalization Fusing

BN is a common method used in training of neural networks. It enables a faster training time and better generalization [12]. Due to these advantages this technique is widely used for training, however a disadvantage of these BN is that it causes problems with quantization. There are solutions for this problems, for example in paper by Lin et al. [13] where a new method for quantizing the BN is presented by converting a transformation of two floating-point numbers into a fixed-point operation with a common quantized scale. Another easier method is to calculate the BN into an following convolutional layer. For this reason we use a mathematical method called Batch Normalization Fusing.
Let's start with equation 3.9 where x defines the input activation and y the output activation given in a batch of size n received from computing samples over a batch. $\mathrm{Var}[x]$ and $\mathrm{E}[x]$ are defined as variance and mean computed over the batch, is the scaling factor, the shift factor and is a small constant included for numerical stability. [14]

$$y_i = \frac{x_i \quad \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] +}} \quad + \tag{3.9}$$

$\mathrm{E}[x]$ (3.11) and $\mathrm{Var}[x]$ (3.10) are recalculated with each batch.

$$\mathrm{E}[x] = \frac{1}{n} \sum x_i \tag{3.10}$$

$$\mathrm{Var}[x]^2 = \frac{1}{n} \sum (x_i \quad \mathrm{E}[x])^2 \tag{3.11}$$

Batch Normalization Fusing

To solve the equation, the BN layer is included in the convolutional layer. w_{conv} indicates the weight and b_{conv} indicates the bias of the convolutional layer. and indicate the weight and the bias of the BN layer.

$$w_{new} = w_{conv} \quad \frac{}{\sqrt{\mathrm{Var}[x] +}} \tag{3.12}$$

$$b_{new} = \frac{b_{conv} \quad \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] +}} \quad + \tag{3.13}$$

The following pseudocode shows an implementation of BN fusing. The input of the function is a convolutional layer and the BN layer which is going to be fused into the convolutional layer. The output is again a *nn.Conv2d* layer.

```
function fuse(conv layer, BN layer):
    mean = running mean of BN
    var_sqrt =sqrt(BN running_var + BN eps)
    beta = BN weight
    gamma = BN bias

    w = conv weight
    if conv has bias
        b = conv bias

    new weight = w \cdot (beta / var_sqrt)
    new bias = (b - mean)/var_sqrt \cdot beta + gamma

    return new conv layer
```

3.4 Symmetric Quantization

As described in section 3.1 symmetric quantization is a quantization method where the absolute max value of each layer is used to scale the floating point weights and bias to a different data-type. Here as example in figure 3.4 the second layer of a pretrained Tiny YOLOv3 is taken to show the impact of a 8bit integer symmetric quantization. The diagrams are shortened to 100 subsamples to increase visibility.

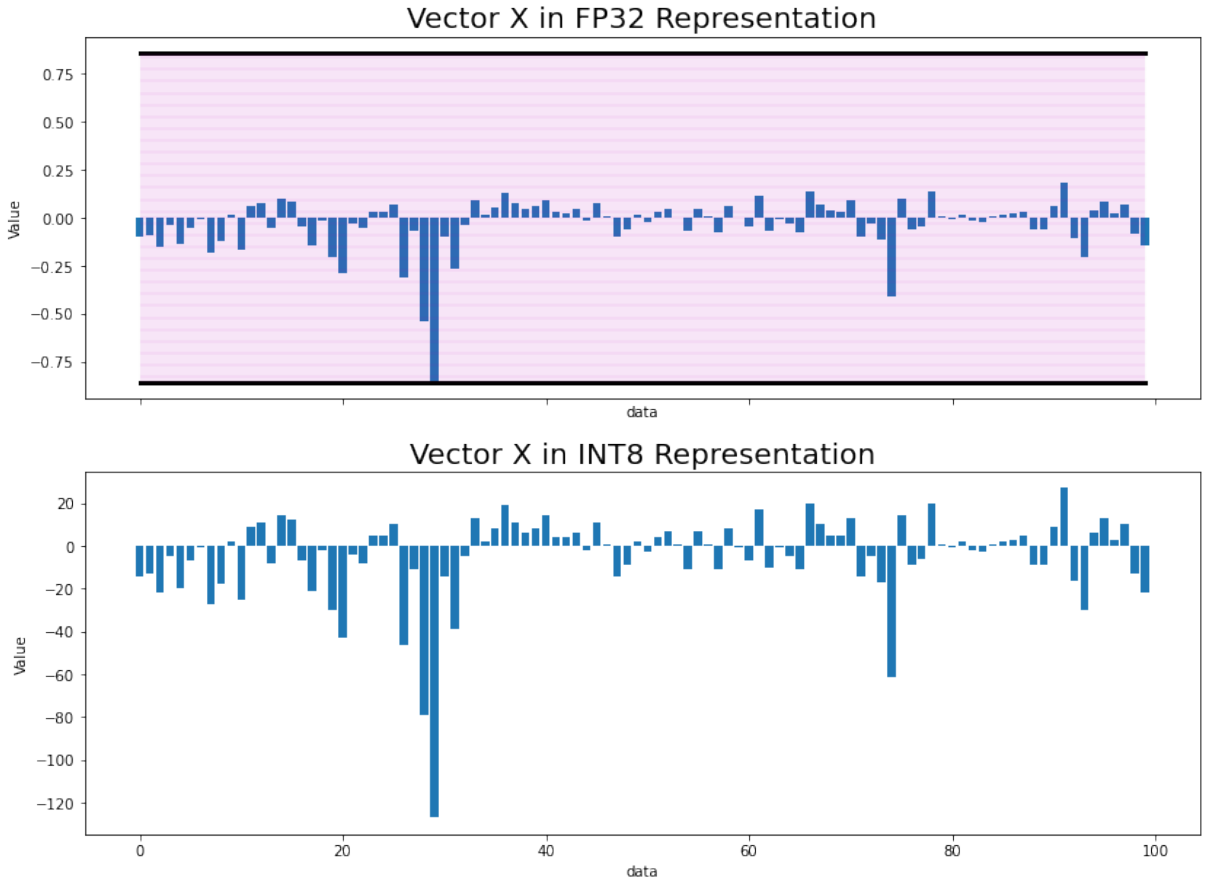


Figure 3.4: Min/Max quantization of Tiny YOLOv3 Conv2

As seen in figure 3.4 and figure 3.5 the range of the quantization is very unbalanced with a few outliers which leads to a high quantization error for values nearby zero as seen in 3.6.

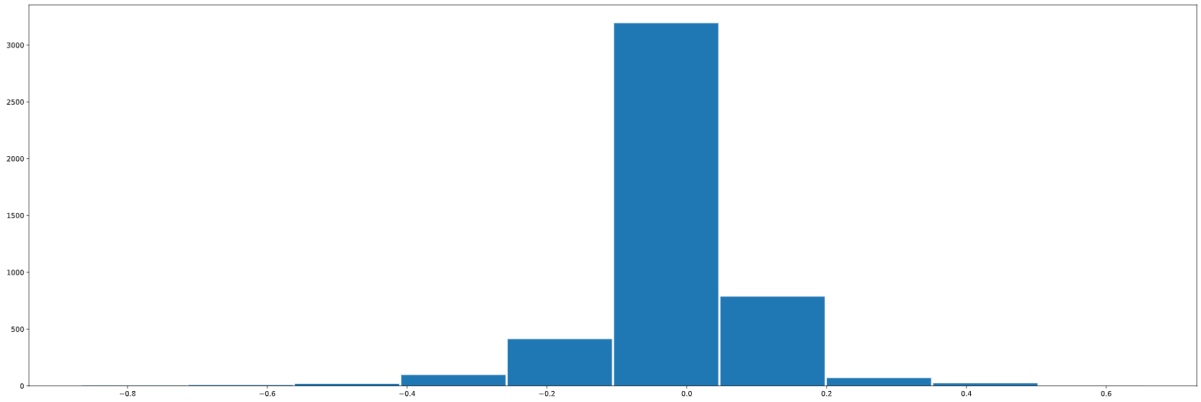


Figure 3.5: Histogram of Tiny YOLOv3 Conv2 Min/Max quantization

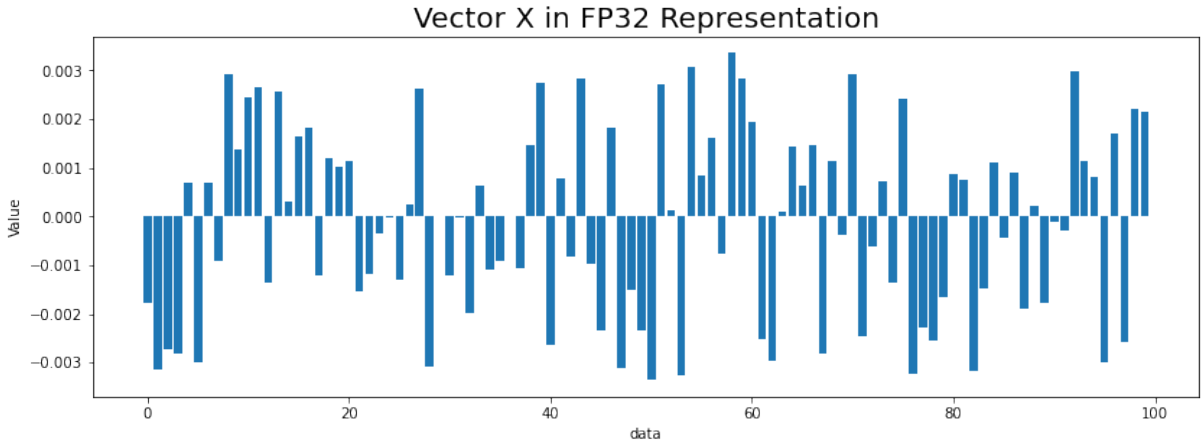


Figure 3.6: Error of Tiny YOLOv3 Conv2 Min/Max quantization

3.5 Remove Outliers

To minimize the quantization error for values nearby zero outliers can be cut off. The disadvantage of removing outliers is the high quantization error for the values that were truncated. The quantization error for removed outliers is much higher than the error for values nearby zero. This leads to a lower result when it comes to accuracy as seen in chapter 4. The more outliers are removed, the higher is the total quantization error. Outliers are defined by the following pseudocode where m describes which value still counts as an outlier. This means the smaller the m factor the more outliers are cut off as seen in figure 3.9 and figure 3.10.

```
function remove_outlier(data, m):
    mean = mean(data)
    std = std(data)

    for value in data
```

```
if abs(value - mean) > m * std
    value = m std

return data
```

As seen in figure 3.7 and figure 3.8 with a smaller m factor more outlier values are truncated.

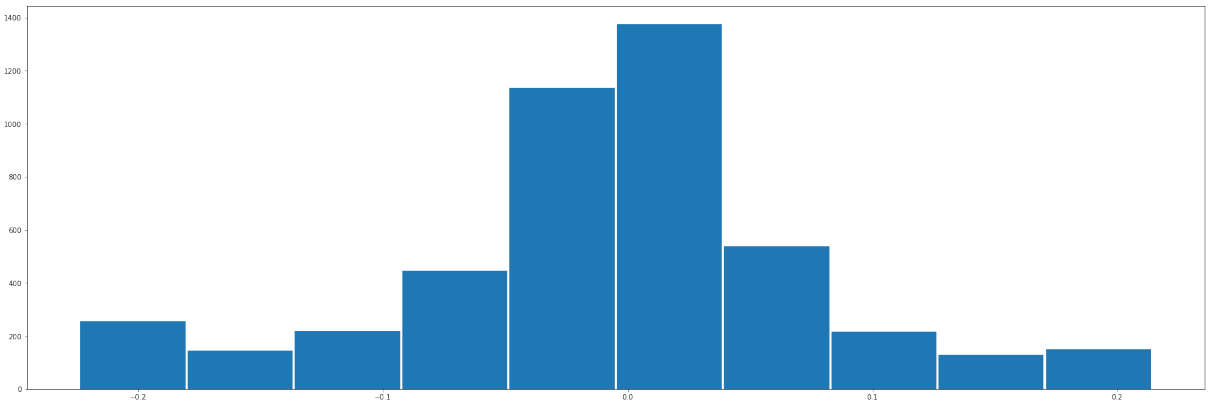


Figure 3.7: Histogram with removed outliers of Tiny YOLOv3 Conv2 with m=2

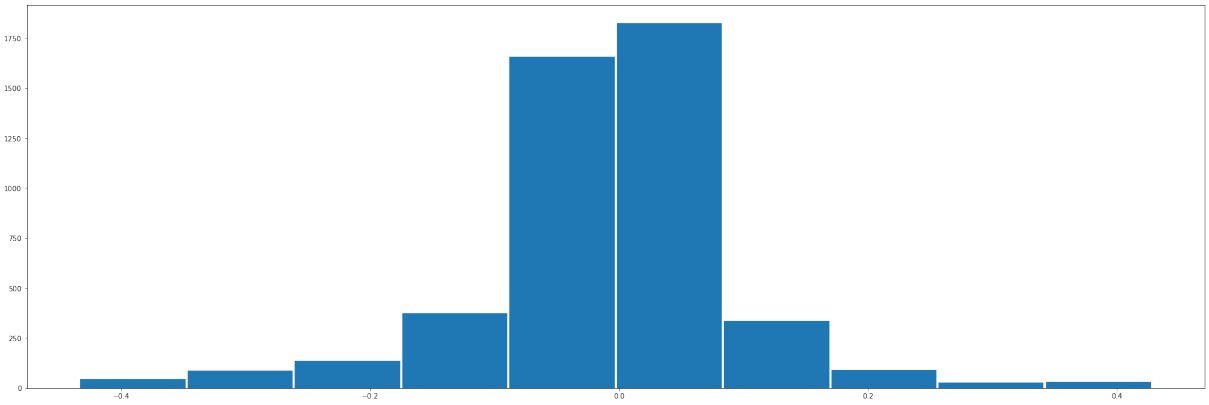


Figure 3.8: Histogram with removed outliers of Tiny YOLOv3 Conv2 with m=4

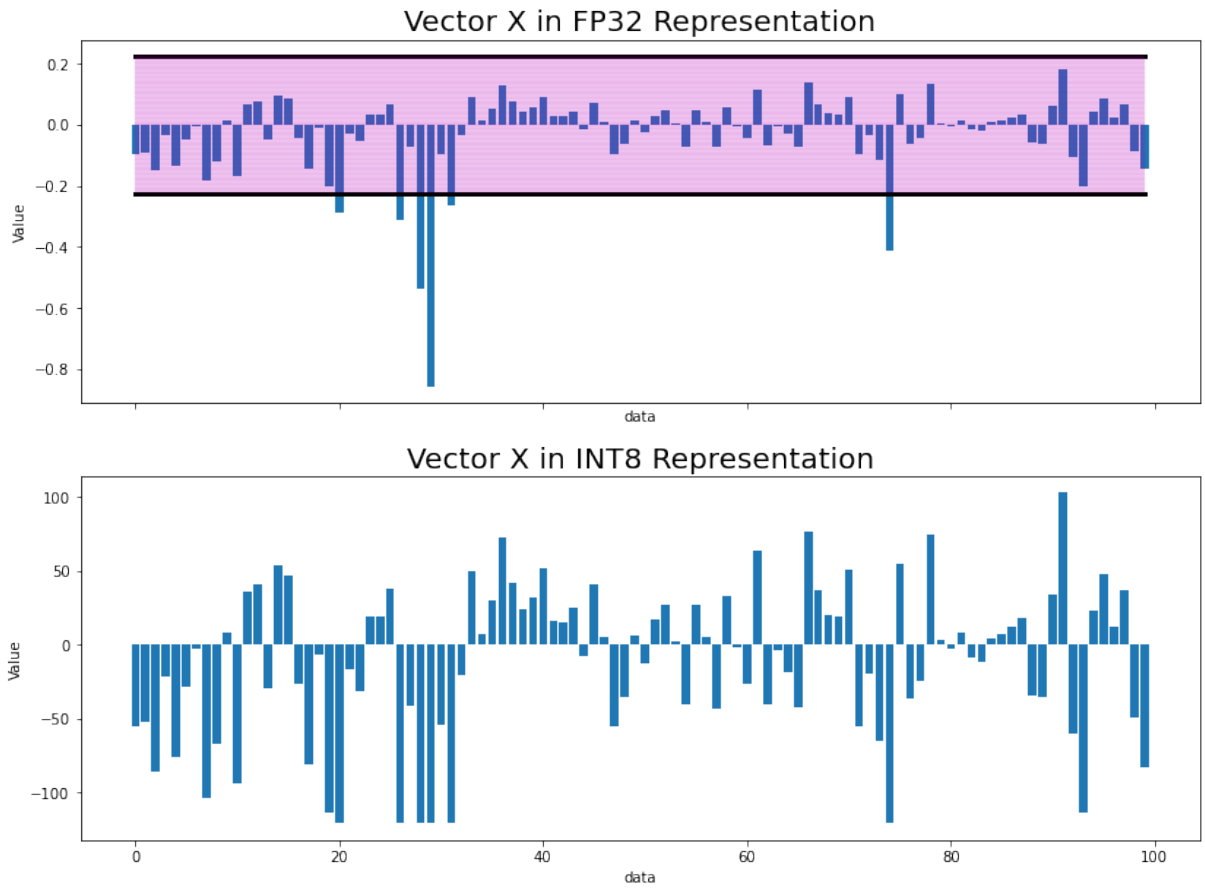


Figure 3.9: Remove outlier quantization of Tiny YOLOv3 Conv2 with m=2

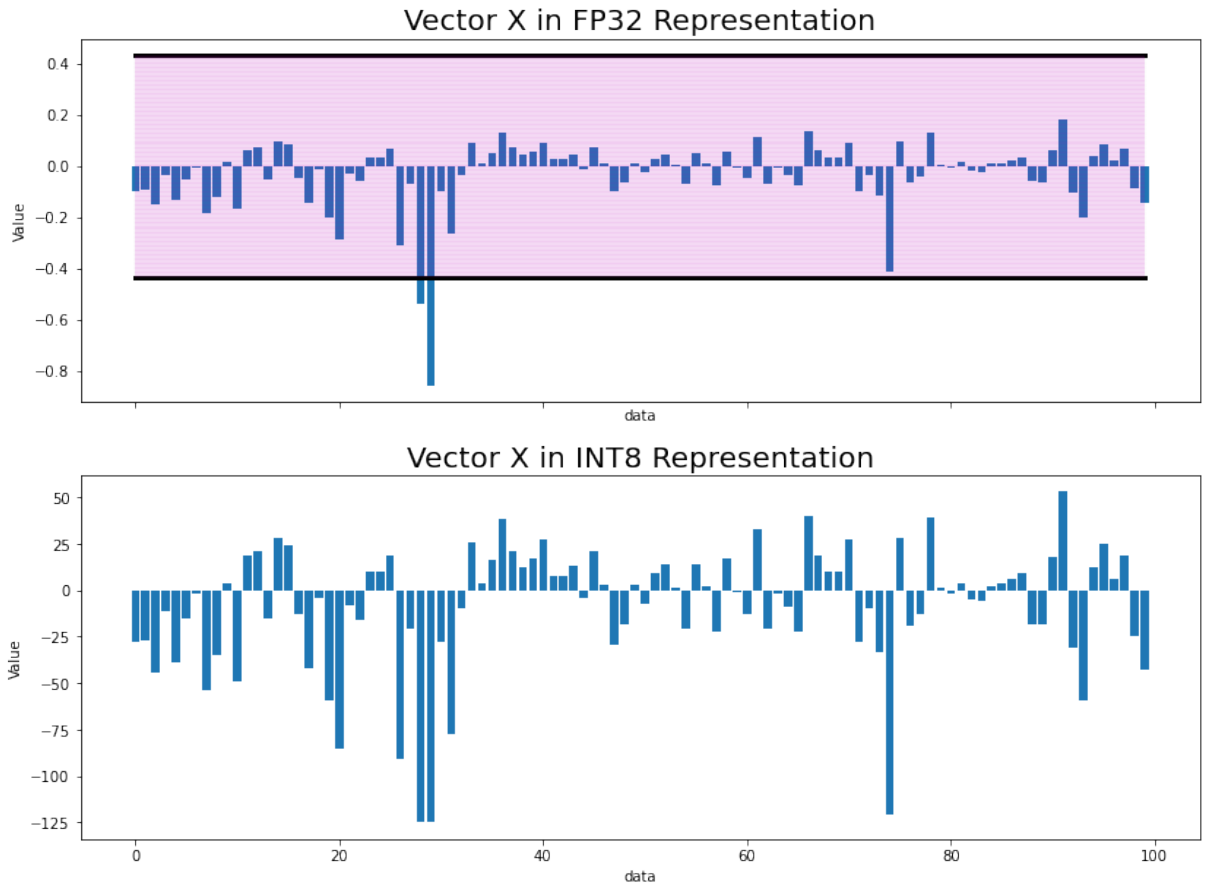


Figure 3.10: Remove outlier quantization of Tiny YOLOv3 Conv2 with m=4

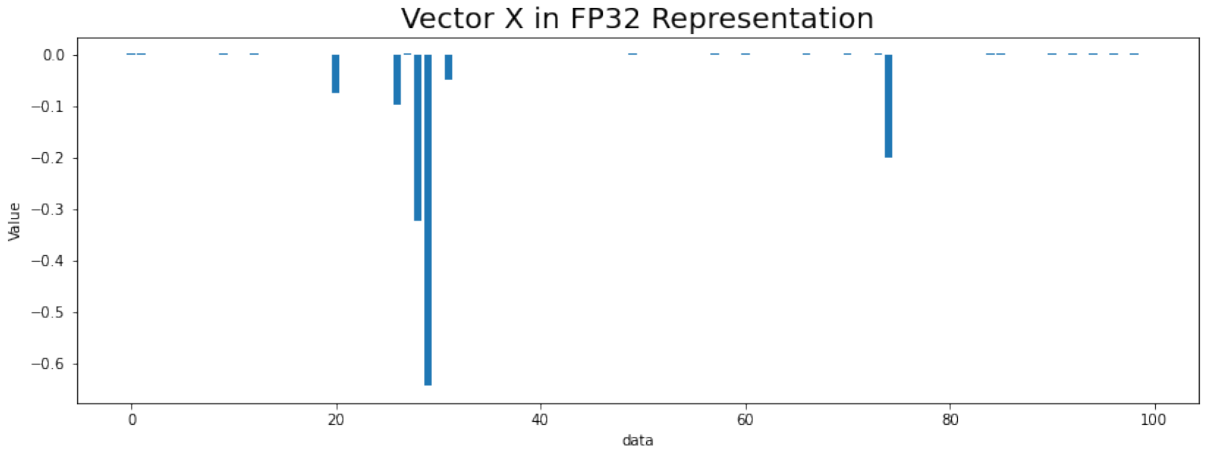


Figure 3.11: Remove outlier error of Tiny YOLOv3 Conv2 with m=2

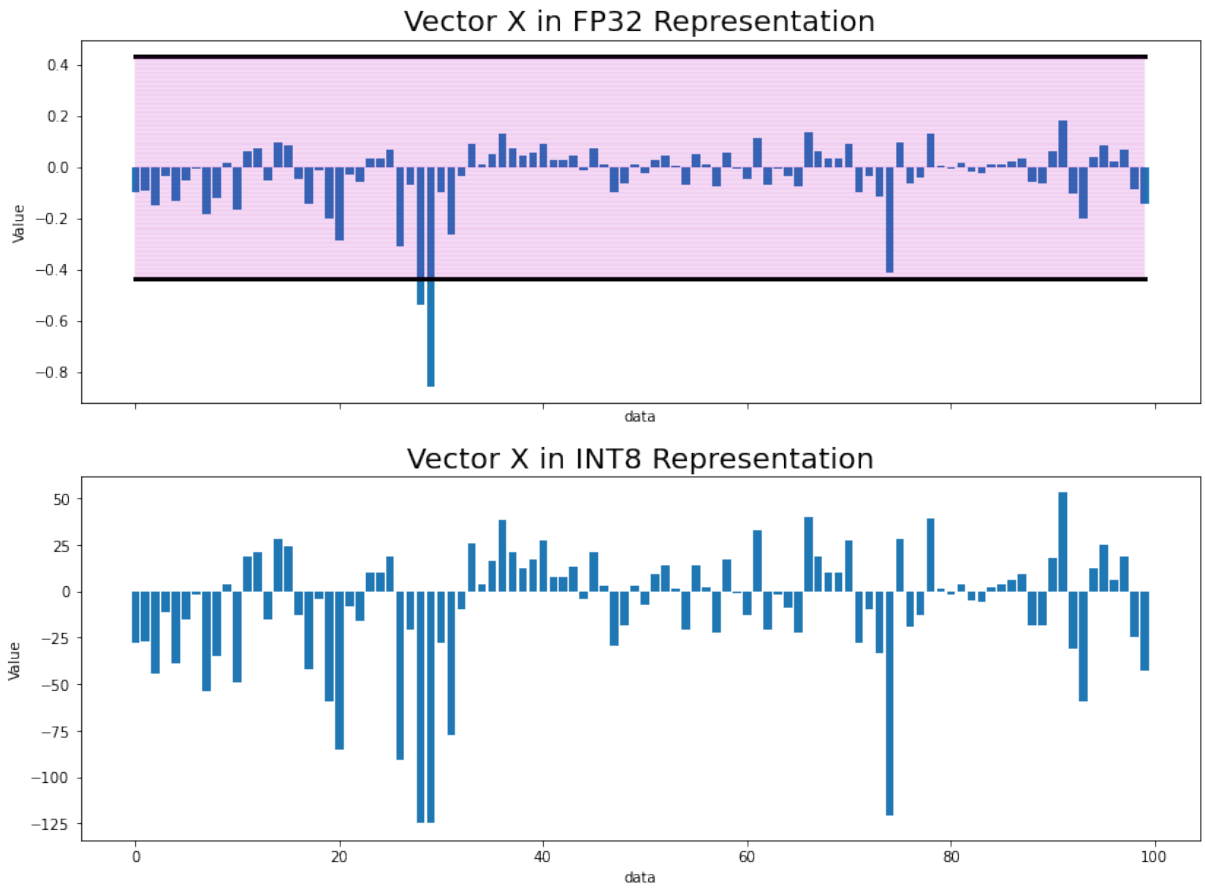


Figure 3.12: Remove outlier error of Tiny YOLOv3 Conv2 with m=4

3.6 Power of 2 quantization

As already discussed in section 3.2.3 the power of 2 adjustments is an ideal method for FPGA's. In order to quantize the network for a power of 2, the integer range is extended so far that the adjustment value $\frac{q_y}{q_x q_w}$ results exactly to a power of 2. If you look at figure 3.13 you can see the extended range which increases the quantization error a little bit but in return brings an enormous advantage for an execution on FPGA's because there are fewer arithmetic operations.

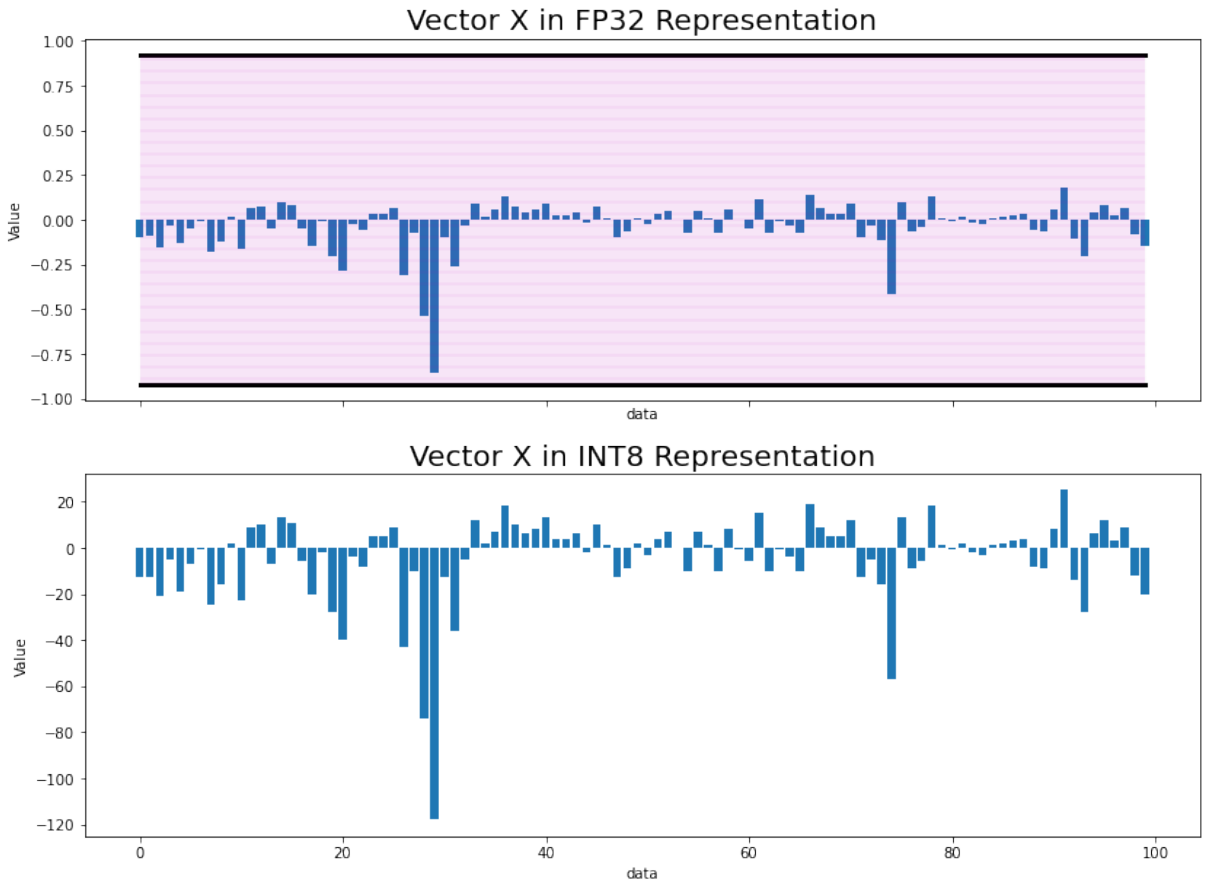


Figure 3.13: Power of 2 adjustment quantization of Tiny YOLOv3 Conv2

3.7 Bias Correction

As seen in figure 3.6 the quantization error can be biased. To reduce this bias error a technique called Bias Correction is used. Such an algorithm is presented in [2] where a Level 1 method is used to correct the biased quantization error of the output/activations of a layer.

$$
\begin{aligned}
y_q &= x_q \quad w_q + \\
y_{dequantized} &= w_q / scale \\
y &= x \quad w \\
\mathbb{E}[x] &= y_{dequantized} \quad y
\end{aligned}
\tag{3.14}
$$

3.8 SimpleNet

For test purpose a very simple network definition is created and trained. SimpleNet only consists out of 2 convolutional, 2 BN and 2 fully connected layer as seen in table 3.1. The network is trained with the CIFAR10 dataset and reaches a accuracy of 72% with floating point inference.

Table 3.1: SimpleNet Layers

Layer	Type	Input			Output		
0	Convolutional	3	32	32	20	28	28
1	Batch Normalization	20	28	28	20	28	28
2	Maxpool	20	28	28	20	28	28
3	Convolutional	20	28	28	50	14	14
4	Batch Normalization	50	14	14	50	10	10
5	Maxpool	50	14	10	50	5	5
6	Reshape	50	5	5	1	1	1250
7	Fully Connected	1	1	1250	1	1	500
8	Fully Connected	1	1	500	1	1	10

Layer schematic of SimpleNet.

3.9 Tiny YOLOv3

The You Only Look Once (YOLO) [15] DNN is since its release in 2016 the state-of-the-art object detection neural network. YOLOv3 [16] is the third evolution of the YOLO networks. This work uses a smaller much faster version of the YOLOv3 network termed as Tiny YOLOv3. As a base for all experiments on Tiny YOLOv3 this git repository is used [1]. The network structure of the Tiny YOLOv3 that is used can be seen in Table 3.2.

This repository uses a BN layer after each red marked layer in the network definition. To quantize this network all the BN has to be fused into the convolutional layer as described in section 3.3.

[1] PyTorch-YOLOv3 by eriklindernoren `https://github.com/eriklindernoren/PyTorch-YOLOv3`

Table 3.2: Tiny YOLOv3 Layers

Layer	Type	Filters	Input			Output		
0	Convolutional	16	416	416	3	416	416	16
1	Maxpool		416	416	16	208	208	16
2	Convolutional	32	208	208	16	208	208	32
3	Maxpool		208	208	32	104	104	32
4	Convolutional	64	104	104	32	104	104	64
5	Maxpool		104	104	64	52	52	64
6	Convolutional	128	52	52	64	52	52	128
7	Maxpool		52	52	128	26	26	128
8	Convolutional	256	26	26	128	26	26	256
9	Maxpool		26	26	256	13	13	256
10	Convolutional	512	13	13	256	13	13	512
11	Maxpool		13	13	512	13	13	512
12	Convolutional	1024	13	13	512	13	13	1024
13	Convolutional	256	13	13	1024	13	13	256
14	Convolutional	512	13	13	256	13	13	512
15	Convolutional	255	13	13	512	13	13	255
16	YOLO		13	13	255	13	13	feature map
17	Route layer 13							
18	Convolutional	128	13	13	256	13	13	128
19	Up-sampling		13	13	128	26	26	128
20	Route layer 19 and 8							
21	Convolutional	256	13	13	384	13	13	256
22	Convolutional	255	13	13	256	13	13	256
23	YOLO		13	13	256	26	26	feature map

Layer schematic of Tiny YOLOv3.
Red marked layers have a Batch norm Layer included.

Chapter 4

Experiments

This chapter reports experiments on post-training quantization, comparing different quantization techniques on SimpleNet and Tiny YOLOv3.

4.1 SimpleNet quantization

Since SimpleNet is a very simple image detection network it is very interference resistant. The quantization error hardly does not take effect until a very small bit width of 3bit as seen in table 4.1. Furthermore the remove of outlier also is rather counterproductive, the error for big values which are cut of is slightly more than the error for values nearby zero. This leads to a lesser overall accuracy.
When comparing the three different alignment types, the quantization error also only comes into effect at a very small bit width of 4 bits, as can be seen in table 4.1, 4.2 and 4.3.
For training and evaluation the CIFAR10 dataset is used.

Table 4.1: Float adjustment comparison

Datatype	Bit Width	AP(%) Min/Max	AP(%) Remove Outlier m=4
integer	2bit	10	10
integer	3bit	24	24
integer	4bit	70	60
integer	8bit	72	68
integer	16bit	72	68
float	32bit	72	72

Comparison between different quantization methods for SimpleNet with float adjustment.

Table 4.2: $x \quad 2^{e}adjustment$ comparison

Datatype	Bit Width	AP(%) Min/Max	AP(%) Remove Outlier m=4
integer	2bit	10	10
integer	3bit	23	14
integer	4bit	62	58
integer	8bit	72	68
integer	16bit	72	68
float	32bit	72	72

Comparison between different quantization methods for SimpleNet with $x \quad 2^{n}$ adjustment.

Table 4.3: Power2 adjustment quantization

Datatype	Bit Width	AP
integer	2bit	10
integer	3bit	18
integer	4bit	60
integer	8bit	72
integer	16bit	72
float	32bit	72

Comparison between different quantization methods for SimpleNet with FPGA optimized adjustment.

4.2 VitisAI

The results for VitisAI are inferenced on a Accelerator Card named Alveo U250 evaluated with an random imageset from COCO2014. As we can see in table 4.4 the floating point accuracy is slightly higher than in the Pytorch Tiny YOLOv3 network used in this thesis. For a quantization to 8bit integer with 416 416 the Mean Average Precision (mAP) drops from 0.362 to 0.296 as seen in table 4.4, compared to the network used in this thesis from 0.291 to 0.280 as seen in table 4.6.

Table 4.4: Tiny YOLOv3 VitisAI results

Input Resolution	mAP float32	mAP int8	latency (ms)	fps
224 224	0.244	0.217	1.30	769.23
416 416	0.326	0.296	3.17	315.46
608 608	0.315	0.301	6.05	165.29

Result of Tiny YOLOv3 with VitisAI quantization. [1]

[1] VitisAI YOLO https://github.com/Xilinx/Vitis-AI/tree/master/examples/DPUCADX8G/yolo

4.3 Pytorch Tiny YOLOv3 quantization

Comparing Tiny YOLOv3 to SimpleNet the effects of quantization are much higher. As seen in table 4.5 the results with a bit width of 4bit are zero mAP. Also like SimpleNet the removal of outlier will have a negative effect on the precision as seen in table 4.5. The three adjustment types have also a very similar impact as in SimpleNet as seen in Table 4.5, 4.6 and 4.7. A more informative representation is seen in figure 4.2b where the different adjustment types are plotted. Here we can see a small loss of accuracy for $x \quad 2^n$ adjustment, a small loss at 8bit too for pow2 adjustment and a slightly more loss at 4bit. For a better understanding, figure 4.1 and figure 4.2 show the same data as in table 4.5, table 4.6 and table 4.7 as a line graph. As we can see here the accuracy decreases from float to $m \quad 2^n$ adjustment to FPGA optimized adjustment, moreover the removal of outlier always leads in an decrease of accuracy.
For evaluation the full testset from the COCO2017 dataset with image size $416 \quad 416$ pixel is used.

Table 4.5: Float adjustment comparison

Datatype	Bit Width	mAP Min/Max	mAP Remove Outlier m=4	theoretical size
integer	4bit	0	0	1/2xMB
integer	8bit	0.281	0.175	xMB
integer	16bit	0.287	0.183	2xMB
float	32bit	0.291	0.291	4xMB

Comparison between different quantization methods for Tiny YOLOv3 with float adjustment.

Table 4.6: $x \quad 2^{e}adjustment$ comparison

Datatype	Bit Width	mAP Min/Max	mAP Remove Outlier m=4	theoretical size
integer	4bit	0	0	1/2xMB
integer	8bit	0.280	0.172	xMB
integer	16bit	0.286	0.183	2xMB
float	32bit	0.291	0.291	4xMB

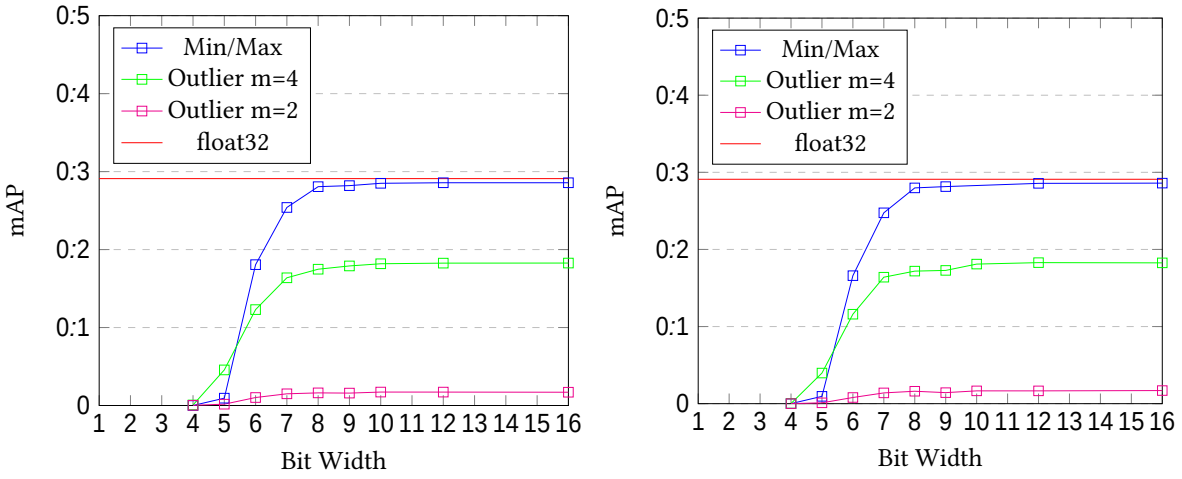
Comparison between different quantization methods for Tiny YOLOv3 with $x \quad 2^n$ adjustment.

Table 4.7: Power2 adjustment quantization

Datatype	Bit Width	mAP	theoretical size
integer	4bit	0	1/2xMB
integer	8bit	0.272	xMB
integer	16bit	0.286	2xMB
float	32bit	0.291	4xMB

Comparison between different quantization methods for Tiny YOLOv3 with FPGA optimized adjustment.

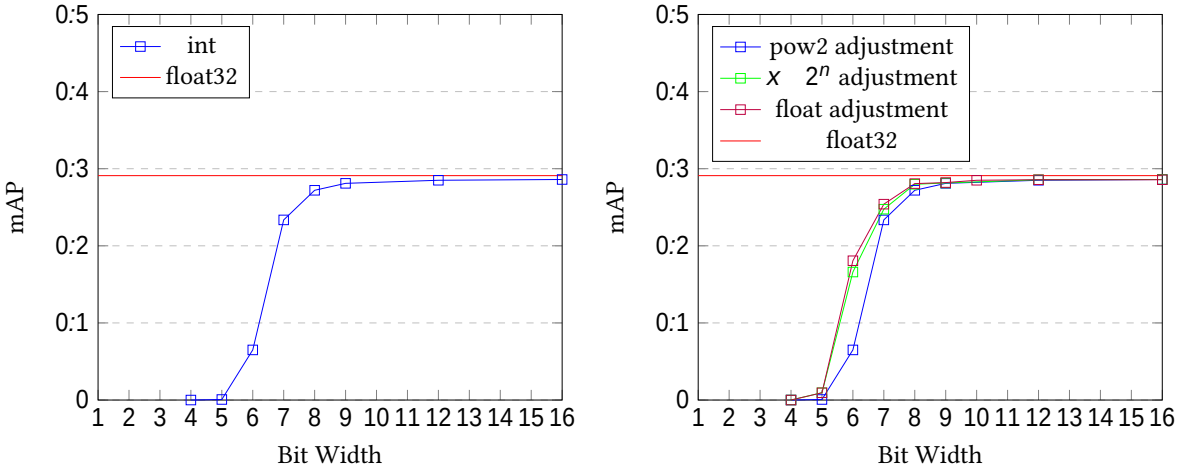
Figure 4.1: Comparison Adjustment



(a) Float Adjustment

(b) $x \quad 2^n$ adjustment

Figure 4.2: Comparison Adjustment



(a) PowerTwo scaling with FPGA adjustment factor

(b) Comparison between the three adjustment types

Chapter 5

Conclusion

Post-Training quantization is a good technique to optimize a neural network for hardware applications without a long retraining time. However, these quantization methods are very suitable for 8bit, but when it comes to a lower precision like 4bit or less more optimization techniques are required to achieve suitable accuracy. Comparing image detection networks like SimpleNet in Section 4.1 to real time object detection networks like Tiny YOLOv3 in Section 4.3 we can see that such simple image detection networks are more interference resistant than more complex object detection networks. Moreover a removal of outlier has in most cases a negative impact on accuracy to due the reason that the big outlier values have more impact on the accuracy than the values nearby zero which doesn't compensate the smaller quantization errors for values nearby zero achieved with the removal of outliers.

5.1 Future Outlook

To push quantization even further, better optimization methods are needed. Promising methods are per channel quantization and per channel bias correction, as some papers [2] [7] have achieved remarkable results for these algorithms. One case that still needs to be investigated is the impact of these per channel quantization on a FPGA optimized adjustment for embedded devices.

Bibliography

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., vol. 25. Curran Associates, Inc., 2012. [Online]. Available: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[2] M. Nagel, M. van Baalen, T. Blankevoort, and M. Welling, "Data-Free Quantization Through Weight Equalization and Bias Correction," *arXiv:1906.04721 [cs, stat]*, Nov. 2019, arXiv: 1906.04721. [Online]. Available: http://arxiv.org/abs/1906.04721

[3] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference," *arXiv:1712.05877 [cs, stat]*, Dec. 2017, arXiv: 1712.05877. [Online]. Available: http://arxiv.org/abs/1712.05877

[4] A. Fan, P. Stock, B. Graham, E. Grave, R. Gribonval, H. Jegou, and A. Joulin, "Training with Quantization Noise for Extreme Model Compression," *arXiv:2004.07320 [cs, stat]*, Feb. 2021, arXiv: 2004.07320. [Online]. Available: http://arxiv.org/abs/2004.07320

[5] T. Kim, Y. Yoo, and J. Yang, "FrostNet: Towards Quantization-Aware Network Architecture Search," *arXiv:2006.09679 [cs, stat]*, Nov. 2020, arXiv: 2006.09679. [Online]. Available: http://arxiv.org/abs/2006.09679

[6] S. R. Jain, A. Gural, M. Wu, and C. H. Dick, "Trained Quantization Thresholds for Accurate and Efficient Fixed-Point Inference of Deep Neural Networks," *arXiv:1903.08066 [cs]*, Feb. 2020, arXiv: 1903.08066. [Online]. Available: http://arxiv.org/abs/1903.08066

[7] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," *arXiv:1806.08342 [cs, stat]*, Jun. 2018, arXiv: 1806.08342. [Online]. Available: http://arxiv.org/abs/1806.08342

[8] V. Sze, Y.-H. Chen, T.-J. Yang, and J. Emer, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," *arXiv:1703.09039 [cs]*, Aug. 2017, arXiv: 1703.09039. [Online]. Available: http://arxiv.org/abs/1703.09039

[9] Y. Guo, "A Survey on Methods and Theories of Quantized Neural Networks," *arXiv:1808.04752 [cs, stat]*, Dec. 2018, arXiv: 1808.04752. [Online]. Available: http://arxiv.org/abs/1808.04752

[10] I. Hubara, Y. Nahshan, Y. Hanani, R. Banner, and D. Soudry, "Improving Post Training Neural Quantization: Layer-wise Calibration and Integer Programming," *arXiv:2006.10518 [cs, stat]*, Dec. 2020, arXiv: 2006.10518. [Online]. Available: http://arxiv.org/abs/2006.10518

[11] R. Banner, Y. Nahshan, E. Hoffer, and D. Soudry, "Post-training 4-bit quantization of convolution networks for rapid-deployment," *arXiv:1810.05723 [cs]*, May 2019, arXiv: 1810.05723. [Online]. Available: http://arxiv.org/abs/1810.05723

[12] S. Santurkar, D. Tsipras, A. Ilyas, and A. Madry, "How Does Batch Normalization Help Optimization?" *arXiv:1805.11604 [cs, stat]*, Apr. 2019, arXiv: 1805.11604. [Online]. Available: http://arxiv.org/abs/1805.11604

[13] D. Lin, P. Sun, G. Xie, S. Zhou, and Z. Zhang, "Optimal Quantization for Batch Normalization in Neural Network Deployments and Beyond," *arXiv:2008.13128 [cs, stat]*, Aug. 2020, arXiv: 2008.13128. [Online]. Available: http://arxiv.org/abs/2008.13128

[14] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," *arXiv:1502.03167 [cs]*, Mar. 2015, arXiv: 1502.03167. [Online]. Available: http://arxiv.org/abs/1502.03167

[15] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," *arXiv:1506.02640 [cs]*, May 2016, arXiv: 1506.02640. [Online]. Available: http://arxiv.org/abs/1506.02640

[16] J. Redmon and A. Farhadi, "YOLOv3: An Incremental Improvement," *arXiv:1804.02767 [cs]*, Apr. 2018, arXiv: 1804.02767. [Online]. Available: http://arxiv.org/abs/1804.02767

Appendix A

Appendix

A.1 Python code of Batch-Norm fusing

The following python function takes two inputs, the target convolutional layer and the BN layer which is going to be fused into the convolutional layer. The function is written in the open source machine learning framework Pytorch[1]. *Conv* has to be a *nn.Conv2d* and *bn* a *nn.BatchNorm2d* layer. The output is again a *nn.Conv2d* layer. Note here that the definition must be rewritten to prevent duplicate execution of the fused BN layer.

```
def fuse(conv, bn):
    w = conv.weight
    mean = bn.running_mean
    var_sqrt = torch.sqrt(bn.running_var + bn.eps)
    beta = bn.weight
    gamma = bn.bias
    if conv.bias is not None:
        b = conv.bias
    else:
        b = mean.new_zeros(mean.shape)
    w = w * (beta / var_sqrt).reshape([conv.out_channels, 1, 1, 1])
    b = (b - mean)/var_sqrt * beta + gamma


    fused_conv = nn.Conv2d(conv.in_channels,
```

[1]Pytorch https://pytorch.org/

```
                        conv.out_channels,
                        conv.kernel_size,
                        conv.stride,
                        conv.padding,
                        bias=True)
    fused_conv.weight = nn.Parameter(w)
    fused_conv.bias = nn.Parameter(b)
    return fused_conv
```

A.2 Python code of util

```
import numpy as np
import torch
import torch.nn as nn
import math
def findShiftScale(val,num_bits):
    # val = x * 2^e
    # e must be a negative integer
    # x must be a positive integer

    #Algorithm works bad if val == optimal solution;
    #Alter decimals for different bit width at pow2 quant
    if np.around(val.item(),9) ==
        math.pow(2, int(round((math.log(val)/math.log(2))))):
        return int(round(math.log(val)/math.log(2))), 1

    e = np.ceil(np.log2(val))
    x = 1

    e_lifo = []
    x_lifo = []

    approx = x * 2**e
    delta = val-approx
```

```
    oldloss = np.square(val-approx)

    while True:
        approx = x * 2**e
        delta = val-approx
        loss = np.square(val-approx)

        if loss < oldloss and delta > 0:
            e_lifo.append(e)
            x_lifo.append(x)

        oldloss = loss

        if delta < 0: # Make approximation smaller
            e -= 1
            x *= 2
            x -= 1

        else:
            x += 1

        if x > 2**num_bits or e < -40:
            if len(e_lifo) != 0:
                return e_lifo[-1],x_lifo[-1]
            else:
                return math.ceil(math.log(val)/math.log(2)), 1
    return 0,0


#Reject outliers
def reject_outliers(data, m=2):
    mean = torch.mean(data)
    std = torch.std(data)
    for idx, value in np.ndenumerate(data.cpu()):
```

```
        if (abs(value - mean)) > m * std:
            if value - mean > 0:
                data[idx] = m * std
            else:
                data[idx] = -m * std
    return data.cuda()

#Power Two
def power_two(data,S,num_bits):
    import math

    max_w = torch.max(torch.abs(data))
    qmax = ((2**num_bits)/2)-1
    qmin = -(qmax+1)
    S["s_w"] = qmax/max_w

    x = S["s_y"]/(S["s_w"]*S["s_x"])
    next = math.pow(2, math.ceil(math.log(x)/math.log(2)))
    S["s_w"] = S["s_y"] / (S["s_x"] * next)
    max_w = qmax * S["s_w"]

    return S["s_w"]
```

A.3 Python code of Tiny-YoloV3

The Python code is based on the git Repo [2] commit 47b7c912877ca69db35b8af3a38d6522681b3bb3

```
from __future__ import division

from models import *
from utils import *
from utils.utils import *
from utils.datasets import *
from utils.parse_config import *
```

[2] PyTorch-YOLOv3 by eriklindernoren https://github.com/eriklindernoren/PyTorch-YOLOv3

```
import os
import sys
import time
import datetime
import argparse
import tqdm

import torch
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision import transforms
from torch.autograd import Variable
import torch.optim as optim

# Stats Functions
class Quant_Method:
    Min_Max = 1
    Bias_Correction = 2
    Power_Two = 3
    Mean = 4
    Two_Third = 5
    Remove_Outlier = 6

# Get Min and max of x tensor, and stores it
def updateStats(x, stats, key, inout):
    max_val = torch.max(torch.abs(x))

    if key not in stats:
        stats[key] = {"input": {"max": 0, "total": 0},"output":
            {"max": 0, "total": 0}}
        stats[key][inout]['max'] = max_val
        stats[key][inout]['total'] += 1
    else:
```

```
        if stats[key][inout]['max'] < max_val:
            stats[key][inout]['max'] = max_val
        stats[key][inout]['total'] += 1


    return stats

# Reworked Forward Pass to access activation
#Stats through updateStats function
def gatherActivationStats(model, x, stats, targets=None):

  img_dim = x.shape[2]
  loss = 0
  layer_outputs, yolo_outputs = [], []

  for i, (module_def, module) in enumerate(zip(model.module_defs,
    model.module_list)):



    if module_def["type"] in ["convolutional", "upsample", "maxpool"]:
        stats = updateStats(x.clone(), stats,
            str(module_def["type"]) + "_" + str(i),"input")
        if module_def["type"] in ["convolutional"]:
            if int(module_def["batch_normalize"]) > 0:
                x = module[0](x)
                x = module[2](x)
            else:
                x = module(x)
        else:
            x = module(x)
        stats = updateStats(x.clone(), stats, str(module_def["type"]) +
            "_" + str(i),"output")
```

```
    elif module_def["type"] == "route":
        x = torch.cat([layer_outputs[int(layer_i)] for
            layer_i in module_def["layers"].split(",")], 1)
    elif module_def["type"] == "shortcut":
        layer_i = int(module_def["from"])
        x = layer_outputs[-1] + layer_outputs[layer_i]
    elif module_def["type"] == "yolo":
        x, layer_loss = module[0](x, targets, img_dim)
        loss += layer_loss
        yolo_outputs.append(x)
    layer_outputs.append(x)

  return stats

# Entry function to get stats of all functions.
def gatherStats(model, test_loader, num_bits):

    model.eval()
    stats = {}
    i = 0

    Tensor = torch.cuda.FloatTensor if torch.cuda.is_available()
        else torch.FloatTensor
    for batch_i, (_, imgs, targets) in
        enumerate(tqdm.tqdm(dataloader, desc="Gathering stats")):
      # Extract labels

        imgs = Variable(imgs.type(Tensor), requires_grad=False)

        with torch.no_grad():
            i += 1
            if i < 25:
                stats = gatherActivationStats(model, imgs, stats)
```

```
    final_stats = {}

    qmax = ((2**num_bits)/2)-1
    qmin = -(qmax+1)

    for key, value in stats.items():
        final_stats[key] = {"s_x" : qmax/(value["input"]["max"]),
            "s_y" : qmax/(value["output"]["max"])}

    return final_stats

from quant_util import *
def quantize_layer(layer, num_bits, S, name, quant_method,m):

    if quant_method == Quant_Method.Min_Max:
        max_w = torch.max(torch.abs(layer.weight.data))
    if quant_method == Quant_Method.Bias_Correction:
        print("Not implemented yet")
    if quant_method == Quant_Method.Power_Two:
        scale = power_two(layer.weight.data.cpu(),S[name],num_bits)
    if quant_method == Quant_Method.Mean:
        layer.weight.data = mean_scale(layer.weight.data)
        max_w = torch.max(torch.abs(layer.weight.data))
    if quant_method == Quant_Method.Two_Third:
        layer.weight.data = squeeze_net(layer.weight.data)
        max_w = torch.max(torch.abs(layer.weight.data))
    if quant_method == Quant_Method.Remove_Outlier:
        layer.weight.data = reject_outliers(layer.weight.data,m)
        max_w = torch.max(torch.abs(layer.weight.data))


    qmax = ((2**num_bits)/2)-1
    qmin = -(qmax+1)
```

```
    if layer.bias is not None:
        max_b = torch.max(torch.abs(layer.bias.data))
        S[name]["s_b"] = qmax/max_b



    if quant_method != Quant_Method.Power_Two:
        S[name]["s_w"] = qmax/max_w
    else:
        S[name]["s_w"] = scale


    layer.weight.data = (torch.mul(layer.weight.data,S[name]["s_w"]))

    layer.weight.data.clamp_(qmin, qmax).round_()

    if layer.bias is not None:
        layer.bias.data = (torch.mul(layer.bias.data,
            (S[name]["s_w"]*S[name]["s_x"])))


    if layer.bias is not None:
        if(torch.max(torch.abs(layer.bias.data)) > qmax):
            k = torch.max(torch.abs(layer.bias.data))
            #print("bias" + str(k))
        layer.bias.data.round_()#.clamp_(qmin, qmax).round_()

    return layer

# Quantization Functions
from quant_util import *
def quantize_tensor(x, num_bits, scale):

    qmax = (2.**num_bits - 1.)/2.
```

```
    qmax = math.floor(qmax)
    qmin = -(qmax+1)

    q_x = x
    q_x.mul_(scale)
    q_x.clamp_(qmin, qmax).round_()

    return q_x

def dequantize_tensor(q_x, scale):
    return (q_x.float() / scale)

def calc_adjust_layer(x, layer, S_layer,num_bits, shift_scale):

    qmax = (2.**num_bits - 1.)/2.
    qmax = math.floor(qmax)
    qmin = -(qmax+1)
    a = layer(x)
    #a = F.relu(a)

    #scale output from layer
    adjustment = (S_layer["s_y"]/(S_layer["s_w"]*S_layer["s_x"]))
    if shift_scale:
        shift,scale = findShiftScale(adjustment.cpu().numpy(),num_bits)
        a = scale * a # Integer will be multiplied;
        #Shift out the unnecessary bits
        a  = ((2.**shift) * a)
    else:
        a = a * adjustment

    a.clamp_(qmin, qmax).round_()
    return a

# Forwarding
```

```
def quantForward(model, x, stats, num_bits, shift_scale):
    img_dim = x.shape[2]
    loss = 0
    layer_outputs, yolo_outputs = [], []

    x = quantize_tensor(x, num_bits,S["convolutional_0"]["s_x"])

    for i, (module_def, module) in
        enumerate(zip(model.module_defs, model.module_list)):
        if module_def["type"] in ["convolutional", "upsample", "maxpool"]:
            if module_def["type"] in ["convolutional"]:
                if int(module_def["batch_normalize"]) > 0:
                    x = calc_adjust_layer(x, module[0],
                    stats[str("convolutional") + "_" + str(i)],
                    num_bits,shift_scale)
                    x = module[2](x)
                else:
                    if i == 15 or i == 22:
                        x = module[0](x)
                        x = dequantize_tensor(x, (S[str("convolutional") +
                            "_" + str(i)]["s_w"]*S[str("convolutional") +
                            "_" + str(i)]["s_x"]))

                    else:
                        x = calc_adjust_layer(x, module[0],
                            stats[str("convolutional") +
                            "_" + str(i)],num_bits,shift_scale)

            else:
                x = module(x)
        elif module_def["type"] == "route":

            x = torch.cat([layer_outputs[int(layer_i)] for
                layer_i in module_def["layers"].split(",")], 1)
```

```
        elif module_def["type"] == "shortcut":
            layer_i = int(module_def["from"])
            x = layer_outputs[-1] + layer_outputs[layer_i]
        elif module_def["type"] == "yolo":
            x, layer_loss = module[0](x, None, img_dim)
            loss += layer_loss
            yolo_outputs.append(x)
        layer_outputs.append(x)
    yolo_outputs = to_cpu(torch.cat(yolo_outputs, 1))
    return yolo_outputs# if targets is None else (loss, yolo_outputs)


def forward(model, x):
    targets = None
    img_dim = x.shape[2]
    loss = 0
    layer_outputs, yolo_outputs = [], []
    for i, (module_def, module) in
        enumerate(zip(model.module_defs, model.module_list)):
        if module_def["type"] in
            ["convolutional", "upsample", "maxpool"]:
            if module_def["type"] in ["convolutional"] and
                int(module_def["batch_normalize"]) > 0:
                x = module[0](x)
                x = module[2](x)
            else:
                x = module(x)
        elif module_def["type"] == "route":
            x = torch.cat([layer_outputs[int(layer_i)] for layer_i in
                module_def["layers"].split(",")], 1)
        elif module_def["type"] == "shortcut":
            layer_i = int(module_def["from"])
            x = layer_outputs[-1] + layer_outputs[layer_i]
```

```
        elif module_def["type"] == "yolo":
            x, layer_loss = module[0](x, targets, img_dim)
            loss += layer_loss
            yolo_outputs.append(x)
        layer_outputs.append(x)
    yolo_outputs = to_cpu(torch.cat(yolo_outputs, 1))
    return yolo_outputs if targets is None else (loss, yolo_outputs)


def testQuant(model, dataloader, num_bits, quant=False, stats=None,
    shift_scale=False):
    model.eval()

    data_config = parse_data_config("config/coco.data")
    valid_path = data_config["valid"]

    path=valid_path
    iou_thres=0.5
    conf_thres=0.001
    nms_thres=0.5
    img_size=416

    Tensor = torch.cuda.FloatTensor if torch.cuda.is_available()
        else torch.FloatTensor

    labels = []
    sample_metrics = []  # List of tuples (TP, confs, pred)
    for batch_i, (_, imgs, targets) in
        enumerate(tqdm.tqdm(dataloader, desc="Detecting objects")):

        # Extract labels
        labels += targets[:, 1].tolist()
        # Rescale target
        targets[:, 2:] = xywh2xyxy(targets[:, 2:])
```

```
            targets[:, 2:] *= img_size

            imgs = Variable(imgs.type(Tensor), requires_grad=False)

            with torch.no_grad():
                if quant:
                    outputs = quantForward(model, imgs, stats,
                        num_bits = num_bits, shift_scale=shift_scale)
                else:
                    outputs = forward(model,imgs)
                outputs = non_max_suppression(outputs,
                    conf_thres=conf_thres, nms_thres=nms_thres)

            sample_metrics += get_batch_statistics(outputs,
                targets, iou_threshold=iou_thres)

        # Concatenate sample statistics
        true_positives, pred_scores, pred_labels =
            [np.concatenate(x, 0) for x in list(zip(*sample_metrics))]
        precision, recall, AP, f1, ap_class =
            ap_per_class(true_positives, pred_scores, pred_labels, labels)

        #print("Average Precisions:")
        #for i, c in enumerate(ap_class):
        #    print(f"+ Class '{c}' ({class_names[c]}) - AP: {AP[i]}")

        print(f"mAP: {AP.mean()}")

    if __name__ == "__main__":

        print(torch.cuda.is_available())

        device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
    # Initiate model

    data_config = parse_data_config("config/coco.data")
    valid_path = data_config["valid"]
    class_names = load_classes(data_config["names"])

    model = Darknet("config/yolov3-tiny.cfg").to(device)
    model.load_darknet_weights("weights/yolov3-tiny.weights")

    import copy
    a_model = copy.deepcopy(model)

    for i, (module_def, module) in
        enumerate(zip(a_model.module_defs, a_model.module_list)):
        if module_def["type"] in ["convolutional"]:
            if(int(module_def["batch_normalize"]) > 0):
                a_model.module_list[i][0] = fuse(module[0], module[1])
                #print(module)

    batch_size = 8

    data_config = parse_data_config("config/coco.data")
    path = data_config["valid"]
    class_names = load_classes(data_config["names"])



    num_bits = 8
    dataset = ListDataset(valid_path,
        img_size=416, augment=False, multiscale=False)
    dataloader = torch.utils.data.DataLoader(
        dataset, batch_size=batch_size, shuffle=False,
            num_workers=8, collate_fn=dataset.collate_fn
    )
```

```
#testing without quantization
#testQuant(a_model, dataloader, num_bits = 8, quant=False)

import copy
import numpy as np

for num_bits in [4,5,6,7,8,9,10,12,16]:
    print(num_bits)

    q_model = copy.deepcopy(a_model)
    S = gatherStats(q_model, dataloader, num_bits)

    for i, (module_def, module) in
        enumerate(zip(q_model.module_defs, q_model.module_list)):
        if module_def["type"] in ["convolutional",
            "upsample", "maxpool"]:
            if module_def["type"] in ["convolutional"]:
                q_model.module_list[i][0] =
                    quantize_layer(module[0],num_bits,
                    S, module_def["type"]
                    + "_" + str(i), Quant_Method.Remove_Outlier,m=2)
    testQuant(q_model, dataloader,num_bits = num_bits,
        quant=True,stats=S, shift_scale=True)
```