An Overview of Arithmetic Adaptations for Inference of Convolutional Neural Networks on Re-configurable Hardware

Ilkay Wunderlich
Institute of Computer Engineering
Technische Universität Dresden
Dresden, Germany
email: ilkay.wunderlich@tu-dresden.de

Benjamin Koch
AVI Systems
Freital, Germany
email: benjamin.koch@avi-systems.eu

Sven Schönfeld
AVI Systems
Freital, Germany
email: sven.schoenfeld@avi-systems.eu

***Abstract*—Convolutional Neural Networks (CNNs) have gained high popularity as a tool for computer vision tasks and for that reason are used in various applications. There are many different concepts, like single shot detectors, that have been published for detecting objects in images or video streams. However, CNNs suffer from disadvantages regarding the deployment on embedded platforms such as re-configurable hardware like Field Programmable Gate Arrays (FPGAs). Due to the high computational intensity, memory requirements and arithmetic conditions, a variety of strategies for running CNNs on FPGAs have been developed. The following methods showcase our best practice approaches for a TinyYOLOv3 detector network on a XILINX Artix-7 FPGA using techniques like fusion of batch normalization, filter pruning and post training network quantization.**

***Keywords*—convolutional neural network; image processing; re-configurable hardware; batchnorm fusing; pruning; quantization;**

I. Introduction

This section introduces the historic background of Single Shot Detectors (SSDs) and the challenges of implementing CNNs on reconfigurable hardware. Afterwards, the general "life" of neural networks is explained and expanded with the adaptation stage.

A. Single shot detector network

After the success of Convolutional Neural Networks (CNNs) for image classification, object detection stepped into the focus of research. A first brute force approach used sliding windows throughout the image with a classification network. This strategy limits itself to the granularity of the window size and window strides.

With Region-based Convolutional Neural Networks (RC-NNs) a more sophisticated tool for object detection was presented. The RCNN itself contains two CNNs, which are solving the tasks of detecting objects of interest and classifying the found objects [1].

The first SSD was published by Joseph Redmon et al. with the iconic name You Only Look Once (YOLO) [2] [3]. The first version was supplemented by two updates: YOLO9000, which is also known as YOLOv2 [4] and the most recently YOLOv3 [5]. Additionally to the YOLO versions smaller versions, named TinyYOLOvX, are provided. The main focus for the following elaborations is put on the TinyYOLOv3 architecture, which optimises the trade off between detection performance and computational effort.

B. Challenges on re-configurable hardware

Implementing CNN on re-configurable hardware introduces several constraints, that affect the architecture of the used networks as well as the underlying arithmetic operations, memory access and scheduling of operations.

Using state of the art Field Programmable Gate Arrays (FPGAs), e.g., XILINX Artix-7, only fixed-point arithmetic can be implemented in an efficient way, so quantization of the network model will be necessary. Because of that, choosing quantization factors and a proper format for intermediate results to keep the deviation regarding the fixed-point model as low as possible is a key consideration to adapt a model for hardware inference.

The most challenging constraint is the limited number of logic elements to implement the building blocks of the CNN. Dissecting the parts of the neural network in candidates for hardware and software implementation is a key consideration to be made in the system architecture. Since FPGAs typically have little on-chip memory an efficient way to access memory has to be part of the architecture design as well. Also, the typical lower clock frequency in re-configurable hardware adds another trade-off.

This leads to the conclusion that a general purpose accelerator is hard to design. Every use case and neural network should be analysed according to needed performance and target platform.

C. Adaptation Stage

The "life" of any neural network can be categorized into two stages: training and inference stage. At the training stage the neural network gets trained for its later task using labelled data, which gets divided into training and test sets. A variety of optimization techniques exists for the training stage. For example, one common target of optimizing the training stage aims on reducing the amount of needed iterations to reach the global minimum of the loss function. Several machine learning libraries implement optimizers for speeding up training, like Root Mean Square Propagation (RMSprop) [6] or Adaptive Moment Estimation (ADAM) [7].

The inference stage is the application of the neural network, after it is properly trained. Adaptations to the network might be needed, depending on the hardware platform on which the inference stage takes place. The entirety of the adaptation work flow is summarized as the adaptation stage.

In section II and III, two optional but very useful adaptation steps are introduced. In Section IV, the mandatory adaptation of switching the arithmetic backbone of the network from floating point to fix point operations is presented. The benefits of these techniques are summarized in section V.

II. Batchnorm Fusing

In this section, the concept of batch normalization and the general layout of a convolution layer are briefly explained. Thereafter, formulae and results of eliminating the batch normalization layer are given.

A. Background of Batch Normalization

Batch normalization [8], which is often times abbreviated with batchnorm, is a sub layer used to reduce internal covariate shifts. These shifts are defined as changes in the distribution of the network's activation, which are caused by changes in the parameters of the network during training stage. Diminishing the covariate shift enables higher learning rates, reduces the risk of getting stuck in poor local minima and prevents vanishing or exploding gradients. Another advantageous side effect of batch normalization is the increased generalization ability of the network [8]. Many modern networks are using batchnorm sub layers, e.g., YOLOv3 [5], MobileNet [9] and ResNet [10].

The batch normalization sub layer is located between the convolutional layer and the activation sub layer which is illustrated for layer i in Figure 1.

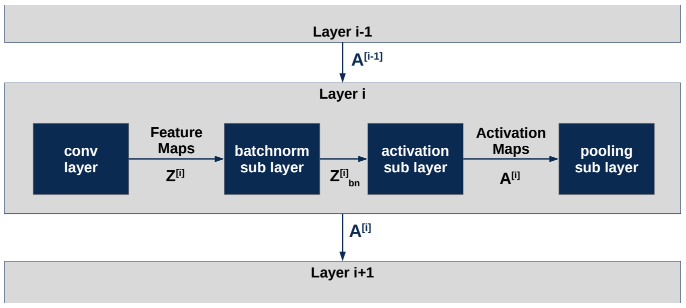

Fig. 1. Example of a general convolutional layer (abbreviated with conv) with its subsequent batchnorm, activation and pooling sub layer and their output descriptions.

This sub layer contains a set of up to four parameters:

1) **Mini-Batch Mean**:
The mini-batch mean $\mu^{[i]}$ is measured regarding the mean of the feature maps of the current mini-batch $\mathcal{B}^{[i]} = \left\{ z_1^{[i]}, ..., z_m^{[i]} \right\}$ at each batchnorm sub layer i in the network. This metric gets updated for each mini-batch and epoch during the training stage:
$$\mu^{[i]} \leftarrow \frac{1}{m} \cdot \sum_{j=0}^{m} \mathcal{B}^{[i]} \tag{1}$$
2) **Mini-Batch Variance**:
Similar to mini-batch mean with respect to the variance of the current mini-batch:
$$\sigma^{2[i]} \leftarrow \frac{1}{m} \cdot \sum_{j=0}^{m} (\mathcal{B}^{[i]} - \mu^{[i]})^2 \tag{2}$$
3) **Scale**:
Trained scaling term $\gamma^{[i]}$, which is an optional and trainable component of the batchnorm sub layer.
4) **Shift**:
Trained shifting term $\beta^{[i]}$, which is an optional and trainable component of the batchnorm sub layer as well. Commonly used instead of bias parameters b in the convolutional layer.

The batchnorm parameters are one dimensional vectors, which are applied to each feature map from the previous convolutional layer separately. The amount of elements is determined by the amount of filters used in the conv layer. The forward propagation step for the batchnorm sub layer is given in (3):
$$Z_{bn}^{[i]} = BN(Z^{[i]}) = \gamma^{[i]} \cdot \underbrace{\frac{Z^{[i]} - \mu^{[i]}}{\sqrt{\sigma^{2[i]} + \epsilon}}}_{normalization} + \beta^{[i]} \tag{3}$$
where ϵ is a small scalar value added to the variance to provide numerical stability (e.g., keras default: $\epsilon = 0.001$ [11]).

B. Fusion of Batchnorm Parameters into Convolutional Parameters

In order to get rid of the computational effort of the batch normalization sub layer, it is recommended to fuse the batchnorm parameters into the convolutional parameters before entering the inference stage. To derive the formulae for batchnorm fusing, the kernel convolution, which is performed filter-wise in the convolutional layer is introduced for layer i:
$$Z^{[i]} = Conv\left(A^{[i-1]}; W^{[i]}, b^{[i]}\right) = A^{[i-1]} * W^{[i]} + b^{[i]} \tag{4}$$
where $A^{[i-1]}$ denotes the three dimensional activation map from the previous layer of matrix shape (w, h, c) - (width,height,color channels). $W^{[i]}$ is the filter matrix of shape (f_w, f_h, c) and $b^{[i]}$ represents the optional bias vector is shape $(c, 1)$. For easier reading, the layer indices $()^{[i]}$ are skipped in the following arguments with hinting $A^{[i-1]}$ as A_{prev}. It can be shown, that the convolution operation $Conv(A; W, b)$ holds the following property:
$$k \cdot Conv\left(A_{prev}; W, b\right) + h = Conv\left(A_{prev}; k \cdot W, k \cdot b + h\right) \tag{5}$$

for $k, h = const.$ and $k, h \in \mathbb{R}^c$. With (3),(4) and (5) the formulae for batchnorm fusing are derived as follows:

Replacing $Z^{[i]}$ from (3) with (4):

$$Z_{bn} = \gamma \cdot \frac{Conv\left(A_{prev}; W, b\right) - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta \tag{6}$$

Reordering to get the equation above to a similar form as shown in (5):

$$Z_{bn} = \underbrace{\frac{\gamma}{\sqrt{\sigma^2 + \epsilon}}}_{k} \cdot Conv\left(A_{prev}; W, b\right) + \underbrace{\beta - \frac{\gamma \cdot \mu}{\sqrt{\sigma^2 + \epsilon}}}_{h} \tag{7}$$

which can be written as:

$$Z_{bn} = Conv\left(A_{prev}; \frac{\gamma}{\sqrt{\sigma^2+\epsilon}} \cdot W, \frac{\gamma}{\sqrt{\sigma^2+\epsilon}} \cdot b + \beta - \frac{\gamma \cdot \mu}{\sqrt{\sigma^2+\epsilon}}\right) = \tag{8}$$

$$Conv\left(A_{prev}; \frac{\gamma}{\sqrt{\sigma^2 + \epsilon}} \cdot W, \frac{\gamma}{\sqrt{\sigma^2 + \epsilon}} \cdot (b - \mu) + \beta\right)$$

With (8) the formulae for the fused parameters W_{bn}, b_{bn} are derived as:

$$W_{bn} = \frac{\gamma}{\sqrt{\sigma^2 + \epsilon}} \cdot W \tag{9}$$

$$b_{bn} = \frac{\gamma}{\sqrt{\sigma^2 + \epsilon}} \cdot (b - \mu) + \beta \tag{10}$$

$$Z_{bn} = Conv\left(A_{prev}; W_{bn}, b_{bn}\right) \tag{11}$$

For convolution layers trained without biases (10) is reduced to:

$$b_{bn} = \beta - \frac{\gamma \cdot \mu}{\sqrt{\sigma^2 + \epsilon}} \tag{12}$$

C. Benefits of Batchnorm fusing

The emphasis for estimating the benefits of batchnorm fusing is shown with respect to the reduction of Floating Point Operations (FLOPS). In Figure 2, the amount of needed FLOPS for the first layers of a TinyYOLOv3 network without batchnorm fusing for an input image size of $416 \times 416 \times 3$ (width, height, color channels) is shown. This network requires approximately 5.5 GFLOPS for processing one image from the input layer to the output layers, excluding the processing steps in the YOLO back-end. After fusing the batchnorm parameters into the convolutional parameters, the FLOPS count is reduced by 23.8 MFLOPS.

This decrease sounds low with a reduction factor of only 0.4%. But more important is the avoidance of batchnorm sub layers as a whole, because each sub layer requires additional logic elements, complexity in the control loop and power. Another benefit is the preparation for pruning, which is performed on the fused weight matrices.

III. Filter Pruning

The general concept of pruning and the proposed routine are presented. The results of pruning are displayed using parameter count and FLOPS as optimization target.

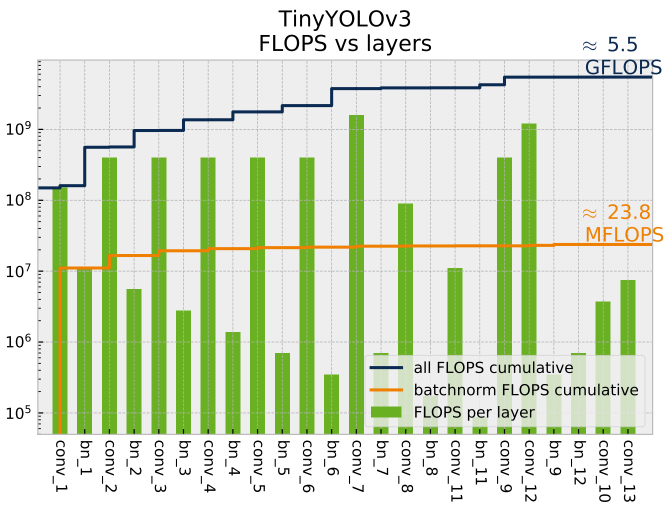

Fig. 2. Required FLOPS (y-axis, logarithmic) for all convolutional layer with their batchnorm sub layer (x-axis) for the TinyYOLOv3 architecture.

A. Pruning Background

In contrast to batch normalization being introduced in 2015, research on pruning of neural networks already started at the end of 1980s [12] and the beginning of 1990s [13]. The goal of pruning is reducing the network size by removing redundant connections while maintaining the performance of the network. M.C. Mozer and P. Smolensky pictorially call this technique as “trimming the fat from a network” [13]. In order to determine redundant connections several metrics are proposed in literature [14]. In the following, the emphasis is put on CNNs and particularly on pruning whole filters after batchnorm fusing.

B. Pruning Metrics

In this section, two metrics for determining filter candidates suitable for pruning are introduced. These metrics help finding filters f_n of the weight matrix W, which have no or low impact for the forward propagation. The total amount of filters stored in the weight matrix W is denoted with nf. A trivial example for a prunable filter is one which coefficients are all zero ($f_n = \mathbf{0}$).

1) The **Frobenius Norm** for filter f_n is defined as the following scalar value:

$$||f_n||_F = \sqrt{\sum_{w,h,c} \left(f_n[w,h,c]\right)^2} \tag{13}$$

This definition gets expanded for the whole weight matrix by stacking the norms of the all filters f_n to one vector:

$$||W||_F = \left[||f_0||_F, ..., ||f_n||_F, ..., ||f_{nf}||_F\right]^T \tag{14}$$

2) The **filter sparsity** is a metric for determining the sparsity of a filter. It is defined as the percentage of values close to zero of a filter f_n:

$$Sp_\epsilon(f_n) = 1 - \frac{\mathcal{C}(|f_n| < \epsilon)}{\mathcal{C}(f_n)} \tag{15}$$

where $\mathcal{C}(f_n)$ denotes the cardinality of the filter f_n (16) and $\mathcal{C}(|f_n| < \epsilon)$ the conditional cardinality of f_n (17).

$$\mathcal{C}(f_n) = \sum_{w,h,c} 1 \tag{16}$$

$$\mathcal{C}(|f_n| < \epsilon) = \sum_{w,h,c} \begin{cases} 1, & |f[w,h,c]| < \epsilon \\ 0, & |f[w,h,c]| \geq \epsilon \end{cases} \tag{17}$$

Similarly to (14) this equation is expanded to:

$$Sp_\epsilon(W) = [Sp_\epsilon(f_0), ..., Sp_\epsilon(f_n), ..., Sp_\epsilon(f_{nf})]^T \tag{18}$$

C. Pruning Routine

The previously mentioned metrics are used in the proposed pruning routine, which is based on the following inputs:

- The maximum deviation of Mean Average Precision (MAP): Δ_{MAP}.
- A representative pruning data set to continuously calculate the mean average precision.
- An optional Starting threshold T_{start}, which is 0 by default.
- A value by which the threshold gets incremented δ_T: E.g., $\delta_T = 0.01$.

The pruning routine determines the chosen metric for every filter in the CNN, as well as the initial MAP of the network on the pruning data set beforehand. Afterwards every filter, which is below the threshold T, is removed and the MAP is calculated again. As long as deviation is lower than the maximum deviation Δ_{MAP}, the threshold is incremented by δ_T. This procedure is repeated until Δ_{MAP} is reached.

After the pruning routine is finished, the possibility of additional training in order to slightly fine tune the remaining filters to reach the initial MAP is possible. It is advisable to perform the fine tuning with a low learning rate and early stopping to avoid over fitting of the network to the pruning data set.

D. Pruning Result

The following example is based on a TinyYOLOv3 network, which is trained on the TU Darmstadt Pedestrian Dataset [15]. The original model, fused model, and the pruned example models as well as additional information are provided in a separate GitHub repository [16].

The maximum deviation of MAP is set to $\Delta_{mAP} = 1\%$ and the threshold increment $\delta_T = 0.02$. In Figure 3, the results of the pruning routine are shown using FROBENIUS Norm $||W||_F$ and filter sparsity $Sp_{\epsilon=0.003}(f_n)$ as the pruning metric. The results cover the total parameter and filter count of the CNN.

It is notable that the parameter count decrease is higher than the decrease in filters. The reason is, that the higher the amount of filters in a layer i the more likely is to encounter prunable filters. The amount of parameters stored in such layers are way larger, because the previous layers $i-1$ typically have higher filter counts as well. This behaviour is visualized for TinyYOLOv3 in Figure 4 by plotting the percentage of stored parameters per layer to the overall TinyYOLOv3 parameter count.

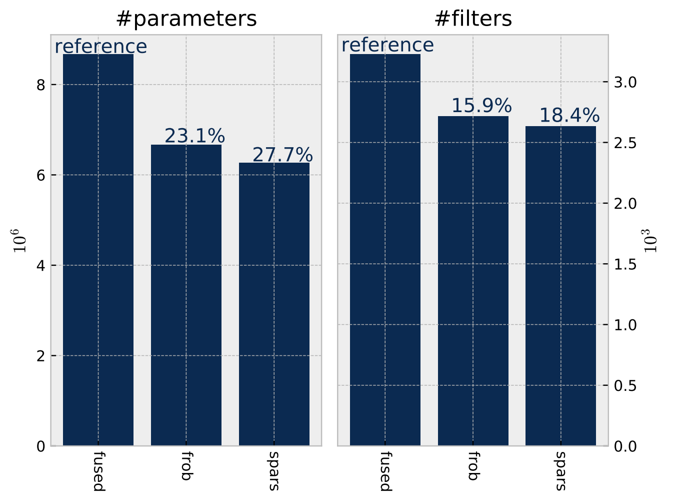

Fig. 3. Pruning result showing for the TinyYOLOv3 network example comparing fused model (fused) and pruned model with FROBENIUS Norm (fro) and Filter Sparsity (spars) as metric. Top of bar: reduction percentage or reference to it. Left: total parameter count. Right: filter count of the network.

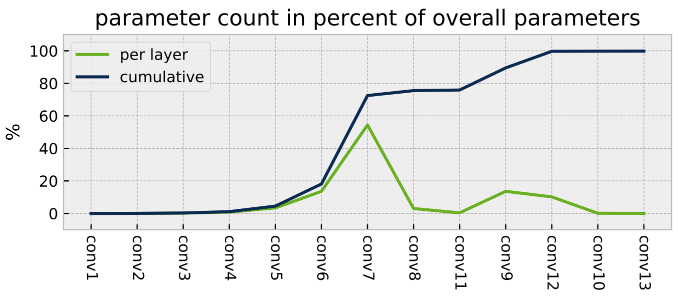

Fig. 4. Percentage of parameters for each layer. Per layer and cumulative.

With the reduction of the parameter count of approximately 23.1% for FROBENIUS and 27.7% for sparsity pruning a higher reduction of FLOPS can be observed compared to batchnorm fusing from Subsection II-C. The comparison of FLOPS reduction percentages between original model (with batchnorm sub layers), fused model and pruned models is visualized in Figure 5. The reduction of FLOPS is 13.3% for FROBENIUS and 15.7% for sparsity pruning compared to the original model with batchnorm sub layers.

Conclusively, it has to be stated that a very high pruning result (e.g., > 80% of parameters pruned) needs more investigation. The CNN might either be oversized or not properly trained. In both cases, the filter weights are still close to the initialization. This leads to low values for the introduced metrics, which result in a huge amount of prunable filters.

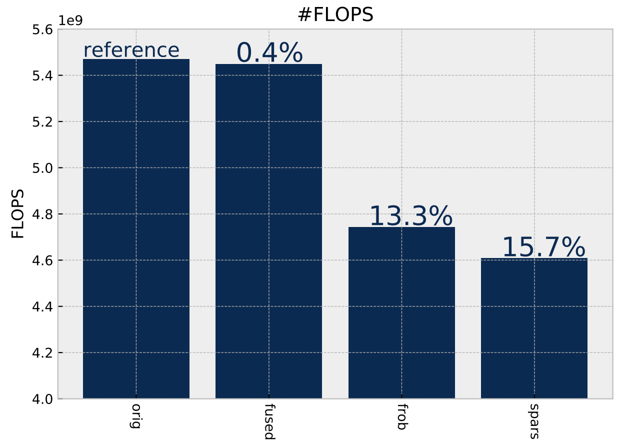

Fig. 5. FLOPS comparison (y-axis) for the TinyYOLOv3 network example. Reduction percentages (top of bar) using the "unfused" original model (orig) as reference.

IV. Quantization

In this section, a method of changing the arithmetic backbone from floating point to integer arithmetic is given and evaluated with investigating on the deviation between floating point CNN and integer CNN. In conclusion, other methods are mentioned and briefly explained.

A. Background

Training for CNNs is traditionally performed using 32 bit floating point arithmetic. The main reason is, that calculating the gradient during CNN training requires high value resolution for which integer or fixed-point systems are insufficient. However, using floating point arithmetic on re-configurable hardware such as FPGAs for CNN inference comes with plenty of disadvantages: higher computation effort, more memory storage and increase of bus widths and counts. These disadvantages can be solved by using a quantized twin of the CNN with a minimal or no drop-off in inference accuracy.

B. Quantization Approach

A straight forward approach based on scaling the floating point values is proposed. This is realized by determining a positive whole-numbered scaling factor S, e.g., $S = 256$. It is advisable to select this factor from the set of powers of two $S = 2^P$ where $P \in \mathbb{N}$, because integer division by a power of two becomes a right shift by the power of two P:

$$\frac{X}{S} = \frac{X}{2^P} = Rshift(X, P) \quad P \in \mathbb{N} \tag{19}$$

This is especially useful since the integer product $Z = MUL_S(X_1, X_2)$ of two mapped operands $X_i = x_i \cdot S$ requires a division by the scaling factor S to obtain $Z = S \cdot x_1 \cdot x_2 + e = S \cdot z + e$ where e denotes the quantization error:

$$\underbrace{Mul_S(X,Y)}_{=:Z} = \frac{X \cdot Y}{S} \overset{S=2^P}{=\!=} Rshift(X \cdot Y, P) = S \cdot \underbrace{x \cdot y}_{=:z} + e \tag{20}$$

Using shift operations instead of divisions spares hardware resources and reduces the combinatorial path, allowing a higher clock frequency of the hardware. With this background, the quantization approach can be introduced:

- **parameter and input quantization:**
  Every parameter value V (floating point, usually 32 bit) of the CNN is quantized by multiplying with the scaling factor S and rounding to integer values. For this, the integer format $int16$ is used.
  $$V_{quant} = int16(round(V * S)) \tag{21}$$
  The input $A^{[0]}$ is quantized similarly for every pixel value $V \in A^{[0]}$.
- **quantized convolution:**
  The forward propagation of the convolution layer, which is split up into the filter convolution $Conv$ and the bias addition Add, is adjusted by appending a right shift operator $Rshift$ after the convolution step as illustrated in Figure 6. The convolution itself is performed using $int32$ range with expansion of the previous activation $A^{[i-1]}$ and the filter parameters $W^{[i]}$ to $int32$:
  $$A^{[i-1]} \leftarrow int32(A^{[i-1]})$$
  $$W^{[i]} \leftarrow int32(W^{[i]})$$
  $$A^{[i]} \leftarrow Conv(A^{[i-1]}, W^{[i]})$$
  After the right shift by P the format is reduced to $int16$ and the bias $b^{[i]}$ is added:
  $$A^{[i]} \leftarrow Rshift(A^{[i]}, P)$$
  $$A^{[i]} \leftarrow int16(A^{[i]})$$
  $$A^{[i]} \leftarrow Add(A^{[i]}, b^{[i]})$$
- **quantized activation function:**
  An approved activation function for CNNs is the Leaky Rectified Linear Unit (ReLU$_\alpha$) as defined in (22) and shown in Figure 7.
  $$a = ReLU_\alpha(z) = \begin{cases} z, & z > 0 \\ \alpha \cdot z, & z \leq 0 \end{cases} \tag{22}$$
  The value of the subscripted α is denoting the negative slope. Common values for α are 0.01, 0.2 and 0.3 (default values of Caffe2, Tensorflow and Keras). For $\alpha = 0$ the Rectified Linear Unit (ReLU) without any "leakage" is obtained.

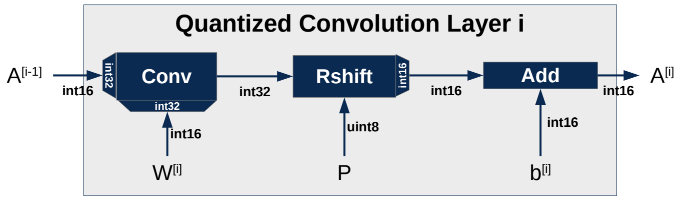

Fig. 6. Quantized Convolution Layer i without pooling and activation sub layer.

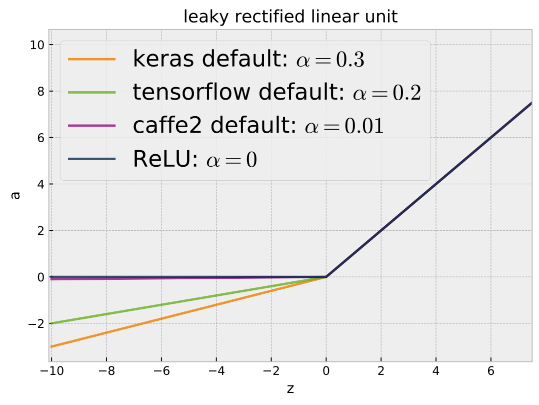

Fig. 7. ReLU$_\alpha$ for various α values.

ReLU$_\alpha$ is a suitable activation function for targeting embedded systems, since its computational effort compared to other activation functions like hyperbolic tangent $tanh$ or sigmoid σ is low. In order to make it even more suitable, the negative slope α is chosen from the set of negative powers of two: $\alpha = 2^{-P_\alpha}$ where $P_\alpha \in \mathbb{N}$: E.g., $\alpha = 2^{-4} = 0.0625$.

Analogously to the convolution quantization the multiplication by α is rewritten using the right shift operator $Rshift$ and the relationship from (19):

$$\alpha \cdot z = 2^{-P_\alpha} \cdot z = \frac{z}{2^{P_\alpha}} = Rshift(z, P_\alpha) \tag{23}$$

This simplifies (22) for integer arithmetic as follows:

$$a = ReLU_\alpha(z) = \begin{cases} z, & z > 0 \\ Rshift(z, P_\alpha), & z \leq 0 \end{cases} \tag{24}$$

- **pooling:**
  No adaptations are needed for transferring max pooling to integer arithmetic. For average pooling, further right shift simplifications can be made.

C. Quantization Deviation

The deviation between the trained floating point model and its quantized integer twin is estimated by the outputs of each convolution layer and the following sub layers (activation, pooling). The deviation is calculated using the Mean Squared Error (MSE) as metric:

$$MSE(A^{[i]}_{float}, A^{[i]}_{int}) = \frac{1}{N} \sum_{\substack{v \in A^{[i]}_{float} \\ w \in A^{[i]}_{int}}} \left(v - \frac{w}{S}\right)^2 \tag{25}$$

with N denoting the number of elements (cardinality) of the compared arrays:

$$N = \mathcal{C}\left(A^{[i]}_{float}\right) = \mathcal{C}\left(A^{[i]}_{int}\right) \tag{26}$$

with S being the scaling factor used for quantization.

Figure 8 shows the layer-wise calculated MSE between floating point model and integer model using a scaling factor $S = 2^8 = 256$. The used network is the pruned version of the pedestrian detector network introduced in Subsection III-D.

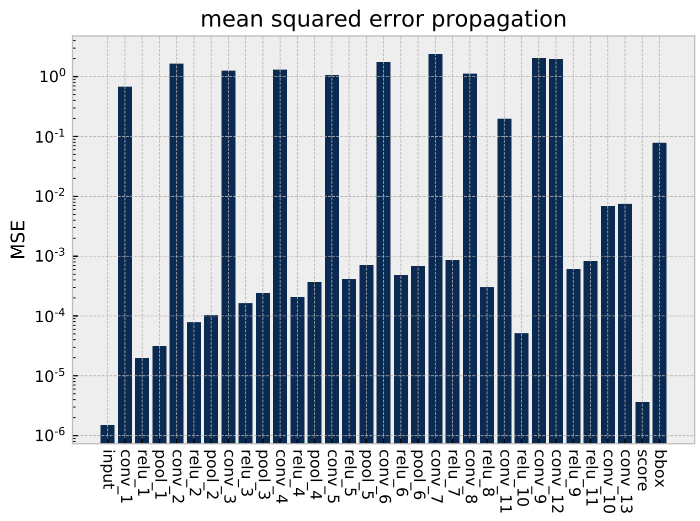

Fig. 8. MSE (y-axis, logarithmic) per layer (x-axis) for TinyYOLOv3 using a scaling factor of $S = 256$ on a randomly chosen image from the test set of TU Darmstadt Pedestrian Dataset [15].

A slow increase of the MSE is observable regarding the outputs of each TinyYOLOv3 layer. This increase plateaus around the fifth layer and stays below a MSE of 0.001. The ReLU$_\alpha$ activation function helps dealing with deviation from the convolution operation by dimming out negative deviation due to the lower negative slope. But most significantly is that the final convolution layers $conv\_10$ and $conv\_13$ display low MSEs. This results in a neglectable difference at the detected classes scores $score$ of $\Delta_{score} = 0.0019$ and a deviation of maximum 2 pixel for the found Bounding Box (bbox).

Similar behaviors with low or neglectable deviations are observed for other TinyYOLOv3 networks, which are trained for different detection tasks, as well as other architectures such as Visual Geometry Group (VGG) classification networks [17].

D. Advanced Approaches

In the following, other promising strategies and more advanced approaches are briefly described:

One approach explicitly tackles the size of the network compressing them by factors up to 30 using weight sharing via k-means clustering method and code books with HUFFMAN encoding [18].

Another goal is to reduce the internal bit widths. In order to achieve that, a distribution based quantization scheme is developed quantizing each filter accordingly to the weight distribution. With that, lower bit widths like 4 bit weights and 8 bit activations are possible [19].

Structural adaptations in architecture designs are proposed for example with MobileNet [9]. MobileNet introduces a new layer type: the depth-wise convolution layer. One of its advantages compared to the "classic" convolution layer is the lower amount of FLOPS required for inference. In MobileNet the ReLU function is extended with a threshold parameter θ

for preventing overflows and allowing lower bit widths. The default value used in MobileNet for θ is 6. The so called ReLU-6 function is directly supported in tensorflow [20].

Another very promising idea is based on replacing multiplications with XNOR operations, which is speeding up inference heavily [21]. The so called XNOR-Net is created for image classification using binary activation and adapted gradient descent methods for training. However, its main disadvantage is the lack of framework support for the commonly used machine learning frameworks.

In order to extend the existing design, appropriate concepts of the presented advanced approaches will be considered.

V. Conclusion

The work flow of the adaptation stage with the following methods is shown:

- **Fusion of batch normalization sub layers** (Section II): Formulae and benefits for eliminating the batchnorm sub layers by fusing the batchnorm parameters into the weight and bias parameters of the preceding convolutional layer are presented.
- **Pruning of convolutional filters** (Section III): A pruning routine with two different pruning metrics is elaborated. The advantages and reduction potentials are exemplary stated.
- **Creation of a quantized twin using integer arithmetic** (Section IV): A straight forward approach with utilisation of shift operation to speed up inference is given. More advanced strategies are mentioned and will be investigated in future works.

Overall, the goal of running a TinyYOLOv3 CNN architecture on an Artix-7 FPGA is accomplished. In order to speed up inference on the FPGA, more optimizations will be developed and supplemented with fitting improvements from other strategies as presented in Subsection IV-D.

Acknowledgment

The investigation of the described adaptations for CNNs on re-configurable hardware are developed at AVI Systems at Freital Germany as well as the chair of VLSI-Design, Diagnostic and Architecture of the Institute of Computer Engineering at Technische Universität Dresden.

References

[1] R. Girshick, J. Donahue, T. Darrell, and J. Malik, “Rich feature hierarchies for accurate object detection and semantic segmentation,”, Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 580–587, 2014.

[2] J. Redmon and A. Angelova, “Real-time grasp detection using convolutional neural networks,” IEEE International Conference on Robotics and Automation (ICRA), pp. 1316–1322, 2015.

[3] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You Only Look Once: Unified, real-time object detection,” IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 779–788, 2016.

[4] J. Redmon and A. Farhadi, “YOLO9000: Better, faster, stronger,” IEEE Conference on Computer Vision and Pattern Recognition (CVPR), doi. 10.1109/cvpr.2017.690, 2017.

[5] J. Redmon and A. Farhadi, “YOLOv3: An incremental improvement,”, arXiv:1804.02767, 2018.

[6] T. Tieleman and G. Hinton, “Divide the gradient by a running average of its recent magnitude,” Tech. Rep., Technical report, p. 31, 2012.

[7] D.P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” arXiv:1412.6980, 2014.

[8] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” Proceedings of the 32nd International Conference on Machine Learning, pp. 448–456, 2015.

[9] A.G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T.Weyand, et al., “MobileNets: Efficient convolutional neural networks for mobile vision applications,” arXiv:1704.04861, 2017.

[10] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” IEEE Conference on Computer Vision and Pattern Recognition (CVPR), doi. 10.1109/cvpr.2016.90, 2016.

[11] F. Chollet and others, “BatchNormalization,” https://keras.io/layers/normalization/, retrieved: 01.2020, 2014.

[12] M. C. Mozer and P. Smolensky, “Skeletonization: A technique for trimming the fat from a network via relevance assessment,” Advances in Neural Information Processing, pp. 107–115, 1989.

[13] E. D. Karnin, “A simple procedure for pruning back-propagation trained neural networks,” IEEE transactions on neural networks, vol. 1, no. 2, pp. 239–242, 1990.

[14] R. Reed, “Pruning algorithms-a survey,” IEEE transactions on neural networks, vol. 4, no. 5, pp. 740–747, 1993.

[15] M. Andriluka, S. Roth, and B. Schiele, “People-tracking-by-detection and people-detection-by-tracking,” IEEE Conference on computer vision and pattern recognition, pp. 1–8, June 2008.

[16] I. Wunderlich, “CNN4FPGA,” GitHub repository, https://github.com/IlkayW/CNN4FPGA.git, 2020.

[17] K. Simonyan and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition,” arXiv:1409.1556, 2014.

[18] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding,” arXiv:1510.00149, 2015.

[19] S. Sasaki, A. Maki, D. Miyashita, and J. Deguchi, “Post training weight compression with distribution-based filter-wise quantization step,” IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS), pp. 1–3 , 2019.

[20] Tensorflow “ReLU-6,”, https://www.tensorflow.org/api_docs/python /tf/nn/relu6?version=stable, retrieved: 01.2020.

[21] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, “XNOR-Net: ImageNet classification using binary convolutional neural networks,” Lecture Notes in Computer Science, pp. 525–-542, doi. 10.1007/978-3-319-46493-0_32, 2016.