Effective Post-Training Quantization Of Neural Networks For Inference on Low Power Neural Accelerator

Alexander Demidovskij
Computer Science Department
Higher School of Economics
Intel Corporation
Bolshaya Pecherskaya street 25/12
Nizhny Novgorod, 603005, Russia
ademidovskij@hse.ru

Eugene Smirnov
Intel Corporation
Turgenev street 30
Nizhny Novgorod, 603024, Russia
eugene.smirnov@intel.com

***Abstract*—Neural network deployment to the target environment is considered a challenging task especially because of heavy burden of hardware requirements that DNN models lay on computation capabilities and power consumption. In case of low power edge devices, such as GNA - neural co-processor, quantization becomes the only way to make the deployment possible. This paper draws attention to the post-training quantization for low-power devices and proves that this approach is practically effective. We propose a novel quantization algorithm capable of reducing DNNs precision to 16-bit or 8-bit integer with negligible drop in accuracy (less than 0.1 percent). The elaborated approach is demonstrated on a set of speech recognition networks trained in Kaldi framework with OpenVINO framework as an inference backend that supports quantization and GNA as a target. Quantization influence on original topologies was rigorously measured and analyzed.**

***Index Terms*—low-bit inference, quantization techniques, artificial neural networks**

I. Introduction

Low power consumption creates hardware constraints on the software that can be executed on edge devices, such as surveillance cameras, embedded electronics etc., especially when task is to run inference of a neural network that requires mullions of floating point operations. In this paper we are going to demonstrate that the only way to infer a model on low-power GNA device is to quantize it by reducing the precision of its weights and activations. GNA (Gaussian Mixture Model and neural network acceleration) is a block that was specifically designed to speedup the neural network inference on systems-on-chip (SoCs) ([1]). It is a co-processor on multiple hardware: Amazon Alexa* Premium Far-Field Developer Kit, Intel® Pentium® Silver processor J5005, Intel® Celeron® processor J4005, Intel® Core™ i3-8121U processor. GNA was chosen as a primary target due to low power consumption and high performance. As a device it is capable to operate while other elements of SoC are in a sleep mode which is extremely important for multiple voice assistants, such as Amazon Alexa, Google Assistant, Apple Siri etc. One of the biggest challenges for deployment of neural networks to a GNA device is absence of support of floating point computations. Taking into consideration that modern neural networks are often already trained in 32-bit floating point precision, the only actual way is to quantize the model.

The structure of this work is as follows: Section 2 gives overview of the background study on existing quantization methods, Section 3 contains deep analysis of GNA hardware limitations. In Section 4 an effective quantization algorithm is proposed with further evaluation of the method in Section 5. Conclusions and directions of further research are formulated in the final part of the paper.

The reported study was funded by RFBR, project number 19-37-90058.

II. Related work

Neural Network quantization is often used for reduction of memory consumption and energy needed to compute the network. It happens due to reduction of bits required to store a single weight - from 32-bit floating point number to, for example, 8-bit integer. The biggest challenge is to keep accuracy level at the appropriate level and minimize its drop after network quantization.

There are multiple types of quantization ([2], [3]): binarization (reducing the weight representation to either -1 or 1) ([4]), fixed point quantization ([5]), adaptive quantization ([6], [7]) etc. At the same time, the target precision can be very different: from 1 bit to 8 bits per a single weight ([8]). Another criteria for comparison of quantization approaches is their relation to the training process: a topology can be trained already in a target precision ([9]) or in 32-bit floating point precision and then quantized to a target one ([10]). In addition to that, quantization is closely coupled with inference on different execution targets ([2]), for example FPGA ([8]), GPU ([11]) etc. Finally, there are scholars who analyze what brings the accuracy drop the most and it is stated that the

978-1-7281-6926-2/20/$31.00 ©2020 IEEE

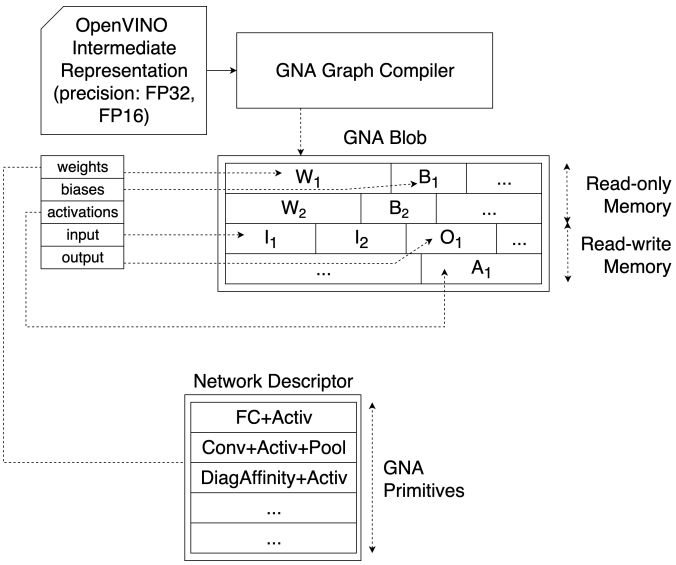
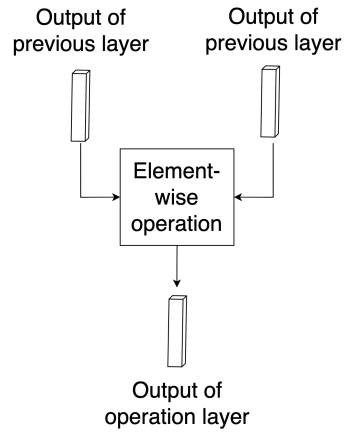
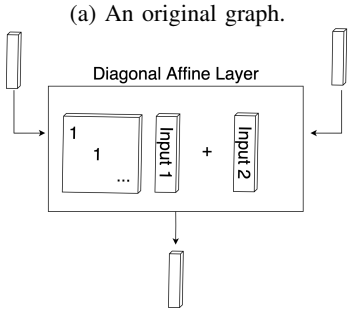


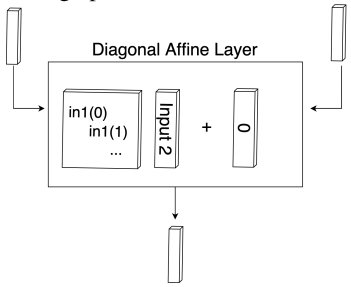
Fig. 1: A generic scheme of network initialization in GNA



(a) An original graph.



(b) Transformation of graph with Sum as an element-wise operation.



(c) Transformation of graph with Multiply as an element-wise operation.

Fig. 2: A generic scheme of expressing not supported element-wise operations with Diagonal Affinity Layer.

biggest part of the accuracy drop happens due to weights quantization ([12]).

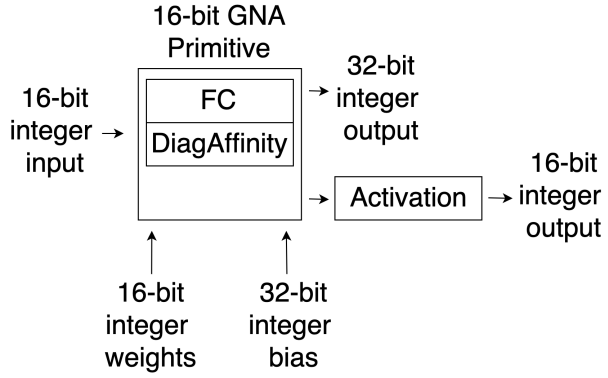
However, to the best of our knowledge there are no attempts of quantizing the model for inference on the GNA with respect to its specificity and hardware limitations. Therefore, the aim of this paper is to focus on the post-training quantization of neural networks for speech recognition trained in 32-bit floating point precision and further deployment of such models to target devices.

III. GNA ARCHITECTURAL LIMITATIONS

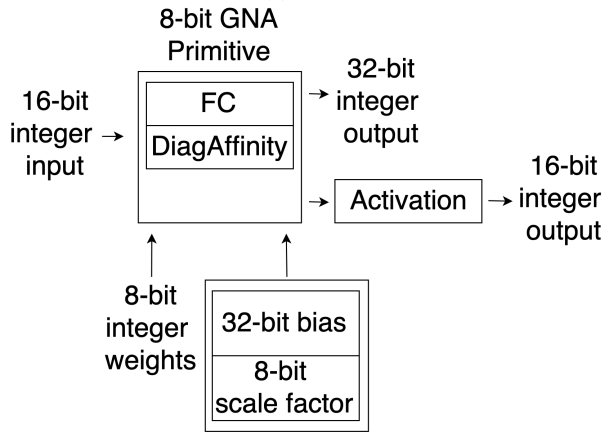
Due to multiple hardware limitations there are only 6 software primitives that could be executed on a GNA 1.0 device, they are usually referred to as 'fused' or GNA primitives: {Fully Connected Layer, Activation Layer}, {Convolution Layer, Activation Layer, Pooling Layer}, {Diagonal Affinity Layer, Activation}, {Copy Layer}, {Interleave Layer}, {Deinterleave Layer}, {Recurrent Layer, Activation Layer}. Last 4 'fused' primitives are not used in actual inference implementation. The very first challenge of any neural network execution on such a device is to express any operation of the neural network as a combination of available GNA primitives. The combination of them is called a GNA Blob (Fig. 1). Transformation of the input OpenVINO IR to a GNA Blob is achieved with GNA Graph Compiler. Due to the limited number of supported primitives, there is a Diagonal Affinity Layer that plays a service role: all summations and multiplications are expressed as matrix operations (Fig. 2).

Each primitive is described with the header containing the parameters information. The weights. biases, inputs and outputs are remembered as pointers to a GNA memory. Network inputs are inputs of the first layer, which outputs are inputs for the second layer and so on. Also, there is a very important aspect that memory on GNA devices is of two types: read-only and read-write. During the initialization phase weights, biases and activation functions approximations should be properly distributed in memory. For example, weights of fully connected and convolution layers, need to be places in read only memory region, while weights of diagonal layers or biases usually calculated and need to be placed in read-write memory region.

The second important limitation, as it was already stated, is that GNA does not support floating point computations. Activation which is a piece-wise linear function can not be executed separately due to absence of appropriate GNA primitive. There are only approximations of activation functions that are stored in format of PWL structure in the Read-Only memory. The precision interface of GNA Primitives is shown for 16-bit (Fig. 3a) and 8-bit (Fig. 3b). Extremely hard this limitation is for the case of inference in 8-bit integer precision



(a) 16-bit



(b) 8-bit

Fig. 3: Precision interface of GNA Primitive

as only Fully Connected layer is supported.

The third limitation is that any GNA 1.0 primitive can process vectors and matrices and is unable to process tensors of higher ranks.

The fourth limitation is mandatory alignment of all memory addresses of 'fused' primitive, like input pointer, output pointer, weights pointer, etc to 64 bits. This requires additional overhead for supporting concatenation of blobs which size is not a multiple of 64.

IV. Proposed quantization algorithm

A. Deployment scheme

Considering all aforementioned hardware limitations, we propose the overall deployment scheme to GNA to be the following:

1) convert a model from Kaldi or TensorFlow to OpenVINO[1] DLDT[2] IR format
2) transform OpenVINO DLDT IR to in-memory Network object
3) quantize the Network:
   a) calculate inputs scale factors from input samples
   b) propagate scale factors through the network
   c) apply scale factors to the network
4) transform quantized Network to GNA Blob
5) initialize network in runtime
6) infer a network on GNA

Due to the primary focus of the paper on the quantization stage of deployment, other steps such as conversion of the pre-trained model to the OpenVINO IR format are taken out of scope of current work and only quantization algorithm is explained in great details in next section.

B. Quantization steps

a) Obtaining input scale factor: Input scale factor is calculated from input source. For speech processing topologies, those inputs are feature vectors files that contain phonemes, words, sentences pronounced by a speaker. Preparation of the input data is out of scope of this paper, input is considered to be a vector in 32- or 16-bit precision. Scale factor is obtained from information about the input precision and input data distribution (1).

$$sf_{input} = \frac{\max(precision)}{2(max - min)}, \tag{1}$$

where $\max(precision)$ is either 2^{32} for 32-bit vectors and 2^{16} for 16-bit vectors, max and min are minimum and maximum numeric values of given input vectors.

Algorithm 1: Scale factor propagation procedure

Result: A list of nodes with computed scale factors
Network ← OpenVINO IR loaded in memory
inputScaleFactors ← calculated input scale factors
sortedNodes ← sort Network nodes topologically
targetPrecision ← 16-bit or 8-bit integer precision
$ForwardPropSF(sortedLayers)$

Function `ForwardPropSF(`$curIndex$`)`**:**
  inputs = Network.inputs
  curNode = Network.nodes[curIndex]
  inputIndex = $findIndex(inputs, curNode)$
  isInput = inputIndex != -1
  inSF = -1
  if *isInput* **then**
    inSF = $inputScaleFactors[inputIndex]$
  else
    inSF = curNode.parents[0].outSF
    conflictingParents = find parents with output scale factor different to the first parent
    for *parent in conflictingParents* **do**
      $BackPropSF(parent, inSF, curIndex)$
    end
  end
  $ApplyInputSF(curIndex, inSF)$
  $ForwardPropSF(curIndex + 1)$

[1]https://docs.openvinotoolkit.org/
[2]https://github.com/opencv/dldt

b) Scale factor propagation: During this step input scale factors are passed through the network layers until outputs are reached. The overall goal is to calculate output scale factor for each layer and its weights if there are any. The main procedure is described in Algorithm 1. In the simplest case of a DNN without branching and single input, this algorithm finishes with one traversal through the tree. There are several important aspects: forward and backward propagation of input scale factors, resolving the conflicting input scale factors in case of branching in the topology, differences in calculation of weights scale factor between 16-bit and 8-bit quantization.

Calculation of the output scale factor in case of forward propagation of scales factors only depends on presence of weights in a layer (2) which is reflected in Algorithm 2.

$$sf_i^{output} = \begin{cases} sf_i^{input} \times sf_i^{weights} & \text{if node } i \text{ has weights} \\ sf_i^{input} & \text{otherwise} \end{cases} \tag{2}$$

Algorithm 2: Procedure for calculating scale factors for a single node

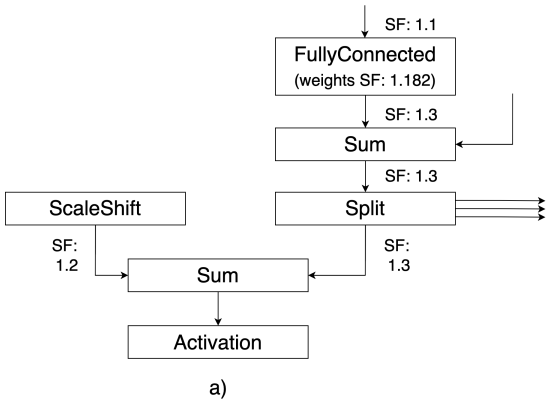
```
Function ApplyInputSF(curIndex, inSF):
    currentNode = Network.nodes[curIndex];
    if currentNode has weights then
        currentNode.weightScaleFactor = according to (2);
        if precision is 8-bit integer then
            currentNode.perChannelScaleFactors = according to (2) in channel-wise manner (max is taken from each row of a weights matrix);
        end
        currentNode.outScaleFactor = inSF * currentNode.weightScaleFactor;
        return;
    end
    currentNode.outScaleFactor = inSF;
return
```

In case of backward propagation of scale factors through a layer with weights, the propagation stops and only the weight scale factor is recalculated (3) which is reflected in Algorithm 4.

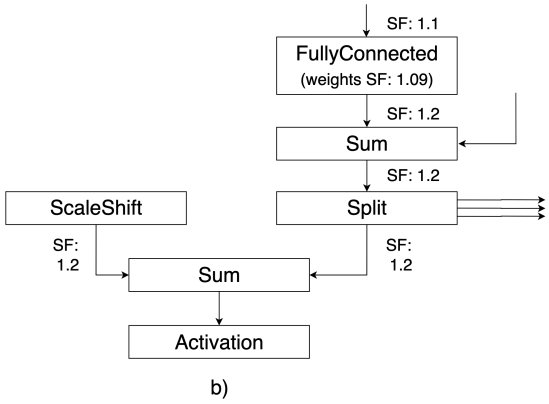
$$sf_i^{weights} = \frac{sf_i^{output}}{sf_i^{input}} \tag{3}$$

The second important aspect is calculation of weights scale factors for 16-bit and 8-bit quantization. In both cases, the overall weight scale factor is elaborated from the weights tensor analysis (4).

$$sf_i^{weights} = \frac{\max(precision)}{2(max)}, \tag{4}$$



(a) Conflict demonstration. Sum layer has different input scale factors.



(b) Backward propagation mechanism stopped at first layer with weights (FullyConnected layer) resulting in changed weights scale factor.

Fig. 4: Heuristic for scale factor conflict resolution.

where $\max(precision)$ is either 2^{32} for 32-bit vectors and 2^{16} for 16-bit vectors, max is maximum numeric value of a given tensor.

However, for 8-bit quantization, the per channel scale factors are calculated for a weights tensor as well. This is performed mostly for saving memory and such scale factors are stored in biases (Fig. 3b). Per channel quantization is considered to produce significantly better accuracy than a per layer one for 4-bit quantization ([12]). In this paper, the same idea is used for 8-bit quantization of weights.

The third aspect of quantization is handling a topology with branches. The main challenge is a potential conflict of output scale factors of two layers that are inputs for another node in a network. The hard requirement of the proposed quantization algorithm is that input scale factors of a layer can not be different. Therefore, there is a mechanism of backward propagation of scale factors through the parents of the node that faced such a conflict. A schematic example is given at Fig. 4 and a full procedure described in Algorithm 3 and 4.

c) Network quantization: After input scale factors are calculated for each layer, it is necessary to apply those scale factors to weights. Due to the fact that scale factors can change multiple times during the previous step, when they are first

Algorithm 3: Procedure for updating the scale factor of node by requested value from children

Function `BackPropSF`(*node, outScaleFactor, curIndex*)**:**
- $ApplyoutSF(node, outScaleFactor)$;
- **if** *node in Network.inputs* **then**
  - inputScaleFactors[i] = $outScaleFactor$;
  - **return**;
- **end**
- childrenToChange = all children except that one that requested a change;
- **for** *child in childrenToChange* **do**
  - childIndex = $findIndex(Network.nodes, child)$;
  - $ForwardPropSF(childIndex)$;
- **end**
- **for** *parent in node.parents* **do**
  - $BackPropSF(parent, node.inSF, curIndex)$;
- **end**

return

Algorithm 4: Procedure for calculating scale factors by requested output scale factor

Function `ApplyoutSF`(*node, outScaleFactor*)**:**
- node.outScaleFactor = outScaleFactor;
- **if** *node has weights* **then**
  - node.weightScaleFactor = according to (3);
  - **if** $targetPrecision$ *is 8-bit integer* **then**
    - node.perChannelScaleFactors = according to (3) in a channel-wise manner;
  - **end**
  - **return**;
- **end**
- node.inSF = outScaleFactor;

return

accumulated, application of scale factors is made once with a forward propagation of scale factors through a network.

V. Evaluation

The proposed algorithm verification is performed on three well known Kaldi[3] topologies: wsj_dnn5b_smbr[4], rm_cnn4a_smbr[5], rm_lstm4f[6]. Each of them represents a particular type of models: Deep Neural Networks (DNN) without convolutional layers, Convolutional Neural Networks (CNN) and Long-Short Term Memory Networks (LSTM).

[3]https://github.com/kaldi-asr/kaldi

[4]https://download.01.org/openvinotoolkit/2018_R3/models_contrib/GNA/wsj_dnn5b_smbr/

[5]https://download.01.org/openvinotoolkit/2018_R3/models_contrib/GNA/rm_cnn4a_smbr/

[6]https://download.01.org/openvinotoolkit/2018_R3/models_contrib/GNA/rm_lstm4f/

TABLE I: Theoretical complexity of pre-trained speech recognition models

Name	Parameters, 10^6	FLOP, 10^6
rm_lstm4f	3.692	1.829
rm_cnn4a_smbr	56.513	26.851
wsj_dnn5b_smbr	57.811	28.903

Name	16-bit IOP	Mixed-bit IOP	
		16-bit	8-bit
rm_lstm4f	3.689	0,018	3.682
rm_cnn4a_smbr	54.913	1.607	53.35
wsj_dnn5b_smbr	57.802	0.013	57.806

TABLE II: Accuracy measurement of the original network and quantized versions (Word Error Rate, WER, %)

Name	OpenVINO 16-bit integer	OpenVINO 8-bit integer	Kaldi 32-bit float
rm_lstm4f	2.22%	2.23%	2.23%
rm_cnn4a_smbr	1.93%	1.95%	1.81%
wsj_dnn5b_smbr	6.44%	6.5%	6.41%

Analysis of theoretical complexity of networks is presented in Table I. It is important to highlight that there is a dramatic increase of operations required to compute the network in 16 bits although they are no longer floating point ones. At the same time, further reduction of precision to 8-bit clearly demonstrates that majority of operations are executed in the target precision, while there is a small amount of operations still requiring computations in 16 bit, to support both element-wise and not 64-bit aligned splitting operations. The effect of weights quantization is shown on Fig. 5 and 6. Due to a mandatory data alignment requirements of GNA hardware, weights are padded with zeros in case of an original model does not satisfy this requirement and histograms for quantized weigths reflect it.

The notable result of the proposed quantization algorithm is that the accuracy drop is negligible (maximum registered drop is 0.12%) compared to the baseline results in the framework and 32-bit floating point inference (Table II). Finally, both original and quantized networks were measured in terms of performance and quantization brings considerable speedup which is a quite expected effect due to reduction of operations complexity (Table III). Considering the notation of Table II and III, benchmarking was made with the following setup: CPU inference running on Intel Skylake-i7-6700K 3.8 GHz (not fixed frequency), GPU running on the discrete graphics on the same machine, NCS2 on Intel Movidius Neural Compute Stick 2, GNA running on GNA co-processor on Intel NUC Kit NUC7PJYH. Version of OpenVINO framework is 2019.R1 and Kaldi framework was taken from commit 1f51ef5d.

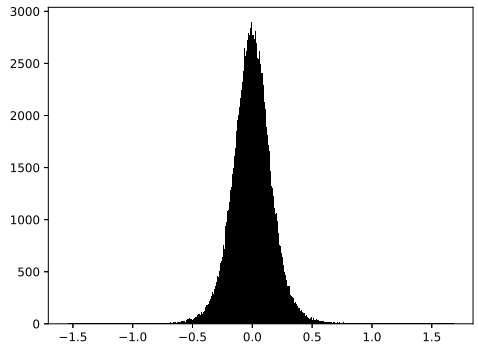
VI. Conclusion

In this paper a simple and effective post-training network quantization algorithm is proposed. It is shown that this ap-

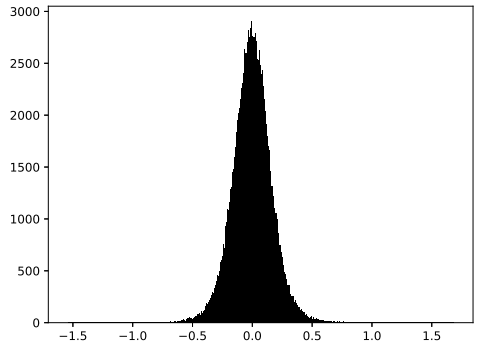
TABLE III: Performance measurement of the original network and quantized versions (in ms - per frame of original Kaldi topology)

Name	OpenVINO 32-bit float		16-bit float		16-bit integer
	CPU	GPU	NCS2	GPU	GNA
rm_lstm4f	0.4	N/A	N/A	N/A	2.49
rm_cnn4a_smbr	5.9	4.83	10.96	2.81	19.0
wsj_dnn5b_smbr	5.6	4.9	8.74	2.68	10.11

Name	OpenVINO 8-bit integer GNA	Kaldi 32-bit float Kaldi
rm_lstm4f	1.78	0.88
rm_cnn4a_smbr	10.37	2.65
wsj_dnn5b_smbr	5.39	2.8

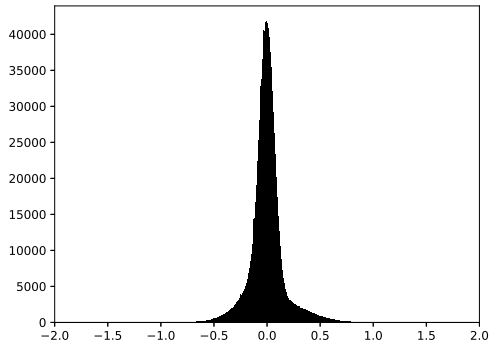


(a) 32-bit floating point precision

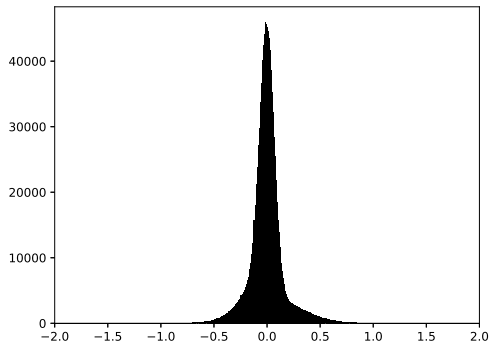


(b) 16-bit integer precision

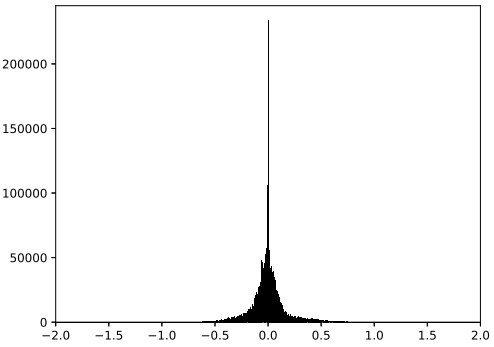
Fig. 5: Histogram of weights for a layer $conv\_7$ from a rm_cnn4a_smbr network



(a) 32-bit floating point precision



(b) 16-bit integer precision



(c) 8-bit integer precision

Fig. 6: Histogram of weights for a layer $affinetransform\_12$ from a wsj_dnn5b_smbr network

proach does not introduce significant accuracy drop (less than 0.12%) when quantizing to 16 bit and 8 bit integer precision. The elaborated deployment scheme allows to infer a neural network on a target device - GNA and successfully satisfies all limitations that this hardware introduces. Proposed quantization solution is generic enough to be applied to any neural network: speech recognition, classification networks, networks that perform intellectual tasks without preliminary training stage [13], [14]. Authors strongly consider that elaboration of solutions for effective Artificial Neural Networks (ANNs) inference on low-power platforms is not just the actual and highly demanded task in the industry that can be immediately applied to existing tasks. Such methods also play an significant role in a broader scientific context. Artificial Neural Networks are the universal means of parallel distributed computation and robust learning. In this field there is a substantial scientific task of constructing neural networks that perform intellectual tasks in massively parallel computation environments like Internet Of Things (IoT). Each component of such systems plays a role of a single neuron or a small sub-network [15]. These computational platforms should be not only distributed and effective, but also robust to unit failures, self-improving in time and should avoid central control. Therefore, current research contributes to this paradigm by providing effective execution rules on the GNA device. Further research is needed in order to build distributed ANN systems consisting of low-power devices, such as GNA, and to analyze performance gains of execution of neural networks in a distributed manner. From the authors point of view the following gaps should be closed in future: considering the case of large and sparse weight matrices during the quantization phase, introduce automatic detection of the initial layer precision for more flexible quantization scheme and extending support for convolutional layers with arbitrary number of channels and kernel size. These additional features will allow to reduce the computational complexity and the accuracy drop after network quantization.

REFERENCES

[1] G. Stemmer, M. Georges, J. Hofer, P. Rozen, J. G. Bauer, J. Nowicki, T. Bocklet, H. R. Colett, O. Falik, M. Deisher *et al.*, "Speech recognition and understanding on hardware-accelerated dsp." in *INTERSPEECH*, 2017, pp. 2036–2037.

[2] R. Ding, Z. Liu, R. Blanton, and D. Marculescu, "Quantized deep neural networks for energy efficient hardware-based inference," in *Proceedings of the 23rd Asia and South Pacific Design Automation Conference*. IEEE Press, 2018, pp. 1–8.

[3] Y. Choi, M. El-Khamy, and J. Lee, "Towards the limit of network quantization," *arXiv preprint arXiv:1612.01543*, 2016.

[4] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Advances in neural information processing systems*, 2015, pp. 3123–3131.

[5] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *International Conference on Machine Learning*, 2015, pp. 1737–1746.

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

[7] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in neural information processing systems*, 2015, pp. 1135–1143.

[8] T. B. Preußer, G. Gambardella, N. Fraser, and M. Blott, "Inference of quantized neural networks on heterogeneous all-programmable devices," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 833–838.

[9] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 6869–6898, 2017.

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

[11] M. Al-Hami, M. Pietron, R. A. Casas, S. L. Hijazi, and P. Kaul, "Towards a stable quantized convolutional neural networks: An embedded perspective." in *ICAART (2)*, 2018, pp. 573–580.

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

[13] G. Pinkas, P. Lima, and S. Cohen, "Representing, binding, retrieving and unifying relational knowledge using pools of neural binders," *Biologically Inspired Cognitive Architectures*, vol. 6, pp. 87–95, 2013.

[14] A. V. Demidovskij, "Towards automatic manipulation of arbitrary structures in connectivist paradigm with tensor product variable binding," in *International Conference on Neuroinformatics*. Springer, 2019, pp. 375–383.

[15] A. Yousefpour, B. Q. Nguyen, S. Devic, G. Wang, A. Kreidieh, H. Lobel, A. M. Bayen, and J. P. Jue, "Failout: Achieving failure-resilient inference in distributed neural networks," *arXiv preprint arXiv:2002.07386*, 2020.