NNCodec: An Open Source Software Implementation of the Neural Network Coding ISO/IEC Standard

Daniel Becking [1] **Paul Haase** [1] **Karsten Müller** [1] **Heiner Kirchhoffer** [1] **Wojciech Samek** [1 2] **Detlev Marpe** [1]

Abstract

This paper presents *NNCodec*, the first open source and standard-compliant implementation of the Neural Network Coding (NNC) standard (ISO/IEC 15938-17), and describes its software architecture and main coding tools. For this, the underlying distributions and information content of neural network weight parameters is analyzed and examined towards higher compression gains. At the core of the coding engine is a context-adaptive arithmetic coder that adapts its binary probability models on-the-fly to weight statistics. We show that NNCodec achieves higher compression than Huffman code, that is commonly used for neural network compression, but also that the average codeword length of NNCodec is often below the Shannon entropy bound.

By introducing specifically trained local scaling parameters, NNCodec can compensate for quantization errors in the latent weight space to a certain degree, which we show experimentally for ResNets, EfficientNet, and a Vision Transformer network topology. The software and demo are available at https://github.com/fraunhoferhhi/nncodec.

1. Introduction

The ubiquitous application of AI methods in many fields of signal processing led to an increasing demand of efficient distribution, training, inference and storage of the underlying neural networks (NN). For this, also efficient compression methods are sought, which provide a minimal coding rate, at which NN performance metrics, e.g., classification accuracy, are not degrading. Similar to a number of multimedia compression methods, a signal theoretical and statistical analysis of the respective source data provides insights for designing an optimized NN codec. In information theory, lossless compression describes coding algorithms that allow the exact, i.e., uniquely decodable, reconstruction of the original source data from the compressed data representation. Assuming data is generated by a probabilistic source, information theory provides concepts to describe the minimum information contained in this data. In his landmark paper (Shannon, 1948), Claude Shannon states that the minimum information required to fully represent a data element w that has probability $P(w)$ is of $I = -\log_2 P(w)$ bit, where I is referred to as *information content*. Data elements which occur frequently, i.e., have a high probability, have a low information content, and vice versa for infrequently occuring elements. Based on the information content, the entropy is defined as $H_P(\mathcal{W}) = -\sum_{w \in \mathcal{W}} P(w) \log_2 P(w)$ and denotes the minimum average number of bits required to represent any element $w \in \mathcal{W} \subset \mathbb{R}^n$. In other words, entropy is a lower bound of the average bit-length required to compress data in a lossless manner.

Entropy coding can efficiently compress the original data, if the data source contains dependencies or statistical properties to be exploited. It compresses input elements w into output codewords of a length approximately proportional to $-\log_2 P(w)$ bits. Thus, more frequently appearing elements are represented by fewer bits. This variable-length coding scheme can be used to further compress an already quantized NN. *Huffman code* (Huffman, 1952) is such a variable-length entropy coding strategy. Experiments in (Han et al., 2016) show that Huffman coding can save 20% to 50% of NN storage. However, in practice Huffman code can require large codeword tables, be computationally complex and produce a bitstream with more redundancies than principally needed (Wiegand et al., 2011). As an improved entropy coding strategy, *arithmetic coding* can be applied. It does not require the storage of a codeword table since the arithmetic code for a sequence of input elements w is iteratively constructed. The superiority of adaptive arithmetic coding schemes for classical source signals like image or video has been shown (Sullivan et al., 2012). More recently, also its efficient applicability to NN source data has been shown (Wiedemann et al., 2020). Accordingly, the context-adaptive binary arithmetic coder of *DeepCABAC* became the coding core of the recently published NNC stan-

[1]Fraunhofer Heinrich Hertz Institute (HHI), 10587 Berlin, Germany [2]Deptartment of Electrical Engineering and Computer Science, Technical University of Berlin, 10623 Berlin, Germany. Correspondence to: <nncodec@hhi.fraunhofer.de>, or <firstname.lastname@hhi.fraunhofer.de>.

Proceedings of the 40^{th} International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023.

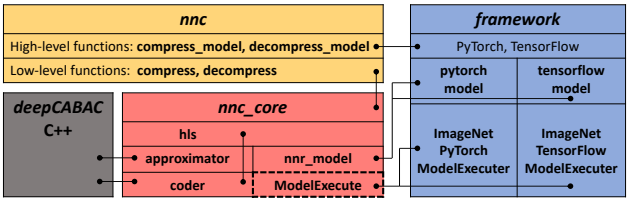

Figure 1. NNCodec software architecture and packages.

dard (ISO, 2022), for which this paper presents an open source software implementation. Main contributions are:

- High-level NNCodec architecture description.
- Fully standard-compliant implementation of the ISO/IEC Neural Network Coding (NNC) standard.
- Feature description of specific NNC tools, which exploit signal statistics for high compression gains.
- Signal statistical analysis of coding tool effects.
- Extensive coding results with full coding tool combination and comparison to other entropy coding methods.

2. NNCodec: Overview of the NNC Standard Software Implementation and Architecture

NNCodec is the first publicly available implementation of the NNC standard. It provides a clearly arranged user interface (cp. *nnc* in Fig. 1) and thus aids the machine learning (ML) community to apply highly efficient compression to NNs in various ML scenarios. Its built-in support for common frameworks, such as TensorFlow© and PyTorch© enables broad applicability to a wide range of NNs in various applications. Additionally, NNCodec offers support for data driven compression methods (see Sec. 3), e.g., local scaling adaptation (LSA) (Kirchhoffer et al., 2022; Haase et al., 2021), for classification models based on the ImageNet dataset. However, its modular architecture design allows for extensibility to arbitrary models and datasets.
The NNCodec software structure, depicted in Fig. 1, is a python package that comprises three modules, *nnc_core*, *framework* and *nnc*, and a fast C++ based DeepCABAC coding engine extension directly linked to it. Here, the *nnc_core* module provides the core coding and compression features for signaling the high level syntax (hls) for parameter approximation (quantization) and for entropy coding as well as data structures (nnr_model) and interface specifications (ModelExecute) for model processing and data driven methods. It implements the compression methods and processes in a generic form in order to be agnostic from any external framework. The PyTorch, TensorFlow and ImageNet support is handled by the *framework* module, which defines framework specific data structures derived from the generic ones (nnc_model and ModelExecute). This ensures correct handling in the *nnc_core* module and enables framework specific functionalities at a higher level at the same time. The main coding tools are described in the next subsection.

3. NNC Technologies

The NNC coding pipeline consist of three stages, namely preprocessing, quantization and entropy coding. First, the two core coding stages quantization and entropy coding are decribed in subsections 3.1 and 3.2, respectively. Classical NN data reduction methods, like sparsification, pruning or low-rank decomposition can be applied prior to the core coding pipeline. Of specific interest for modifying signal statistics are BatchNorm folding and local scaling adaptation as part of the preprocessing stage, which are described in subsections 3.3 and 3.4, respectively.

3.1. Quantization

Analogously to other coding standards, NNC has a parameter quantization stage, which provides a) further compression and b) integer quantization indices that can be losslessly entropy coded. For this, NNC specifies methods for scalar quantization with a uniform reconstruction quantizer (URQ) and for vector quantization using dependent quantization (DQ) (Haase et al., 2020b) also known as Trellis-coded quantization (TCQ), which usually achieves a higher compression efficiency at the same model performance level. The scalar quantization method uses a single URQ with uniformly spaced reconstruction levels. In contrast, DQ employs two scalar quantizers with distinct sets of reconstruction levels and a procedure for switching between them. For both methods, the reconstruction levels can be determined by an integer quantization index and, for DQ, additionally, by the applied quantizer. As a further method, NNC specifies transmission of an integer codebook, which can be derived from the output of arbitrary quantization methods, e.g., K-means clustering. In all cases, a quantization step size Δ is derived from an integer quantization parameter qp (cp. Eq. (2) in A.1), which provides a mechanism for controlling the rate-performance trade-off (Kirchhoffer et al., 2022). NNC allows to specify an individual qp value for each model parameter (tensor). Implicitly, optimization of the these qp values can significantly improve the compression efficiency. For this, NNCodec provides a data-free qp optimization technique (enabled by the flag '$--opt\_qp$'), that is based on the tensor statistics, such as standard deviation or the number of weights (Haase et al., 2021).

3.2. Entropy Coding

For entropy coding, NNC employs an adapted version of the context-based adaptive binary arithmetic coding (CABAC) scheme (Marpe et al., 2003). It consists of three stages: Binarization, context modeling, and binary arithmetic coding. The binarization stage maps each symbol to be encoded (e.g., a quantized weight) to a sequence of binary symbols

(bins). The context modeling stage associates each bin with a so-called context model, which associates a probability estimate with the bin. The binary arithmetic coding stage encodes (or decodes) the binary symbol by using this probability estimate, provided by the context model. Each context model implements a backward-adaptive state-based probability estimator as in (Haase et al., 2020a) which maintains an internal state that represents the probability estimate. After encoding (or decoding) of a bin, the state is updated. If a bin of value 1 was processed, the probability estimate for bin $= 1$ increases and if a 0 was processed, the probability for bin $= 1$ decreases. The degree of increasing or decreasing the probability estimate can be controled by so-called adaptation rate parameters.

NNC supports a forward-signaling of the adaptation rate settings for each context model, i.e., an encoder can decide to optimize the adaptation rate (and the initial probabilities) of each context model and transmit these optimized parameters in the bitstream. For the binarization of a (quantized integer) weight, a truncated unary code is combined with a sign flag and an exponential Golomb Code. This ensures that weights with a smaller magnitude are represented by fewer bins. Furthermore, a sophisticated context modeling ensures that DeepCABAC is able to adapt to a wide range of differently shaped weight distributions. A more detailed description can be found in (Kirchhoffer et al., 2022).

3.3. BatchNorm Folding

Batch Normalization (BN) (Ioffe & Szegedy, 2015) is a technique to normalize the input activations of an NN layer per data batch for more stable training. Especially in modern architectures, BatchNorm parameters are ubiquitous. To recap, a BatchNorm module consists of a set of four vector parameters: running mean E, running variance Var, and two learnable scale- and shift- vectors γ and β. The output y of an input x of a BatchNorm layer is defined as

$y = \frac{x - E(x)}{\sqrt{\text{Var}(x) + \epsilon}} \cdot \gamma + \beta$ with $\epsilon \leq 0.001$ for stability.

BatchNorm folding (BNF) (Jacob et al., 2018) is a technique which "folds" the multiplicative BN parts α into the preceding NN layer's weights $\mathcal{W}$, i.e., $\mathcal{W}_{\text{fold}} = \alpha\mathcal{W}$, and replaces the layer's bias term b with δ, where

$$\alpha = \gamma / \sqrt{\text{Var}(x) + \epsilon} \quad \text{and} \quad \delta = \beta + \alpha(b - E(x)). \quad (1)$$

At the decoder, the δ parameter can be loaded into the weight module's bias buffer b (which allows for removing the associated BN module from the computational graph for model speed-ups), or be loaded into the BN module's β-buffer with the remaining BN parameters set to default, i.e., $E = 0$, Var$= 1$, and $\gamma = 1$.

3.4. Local Scaling Adaptation

Local scaling adaptation (LSA) equips NN layers with additional trainable scaling factors s at each output element.

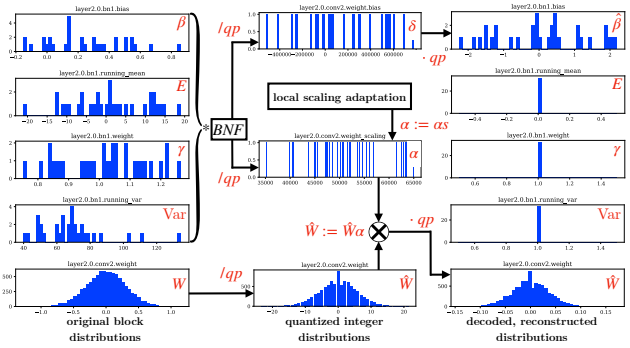

Figure 2. NN parameter distributions throughout coding.

Specifically, one scaling factor is assigned per tensor row, which in turn can represent a convolutional filter or a single output neuron. After quantization, all model parameters except the scaling factors are fixed. Then only the scaling factors are trained, which allows the NNCodec encoder to compensate for the potentially resulting quantization error to some extent and consequently support higher compression through coarser quantization. Subsequently, the scaling factors can be merged with α by element-wise multiplication, i.e., $\alpha := \alpha s$, so that LSA does not introduce additional parameters to be encoded when used in conjunction with BNF. Finally, the decoder of NNCodec multiplies the entries of the α vector with the associated rows of $\mathcal{W}$.

4. Neural Network Parameter Statistics

Analyzing an NN's weight statistics often reveals asymmetric, monotonically decreasing distributions with a mean value near zero (Gauss- or Laplace-like). In appendix A.3, we demonstrate this sort of distribution for all weight parameters within a ResNet-56 (original Fig. 8, quantized Fig. 9 and reconstructed Fig. 10). To take advantage of the large quantity of zero values, DeepCABAC determines in a first binarization step whether the weight element is a "significant" non-zero element or whether it is quantized to 0. The context model for this *SigFlag* is initially set to a probability of 50%, but automatically adapts to the statistics.

In Fig. 2, we demonstrate the distributions of a coded data unit (a block consisting of a convolution layer and its associated BatchNorm module). The original BatchNorm parameters on the left hand side are folded into their respective multiplicative (α) and additive (δ) compounds according to Eq. (1). Then α, δ and the weights $\mathcal{W}$ are quantized uniformly (cp. Fig. 2 center). Optionally, LSA is applied which trains scaling factors s that are multiplied into α and be encoded together with β and $\mathcal{W}$ in one coding unit. At the decoder, the parameters are reconstructed by multiplying the integer representations with their associated qp value, by merging α with $\mathcal{W}$ and by loading δ into the BatchNorm module's β parameter; the remaining BN parameters are set to default (cp. Fig. 2 right).

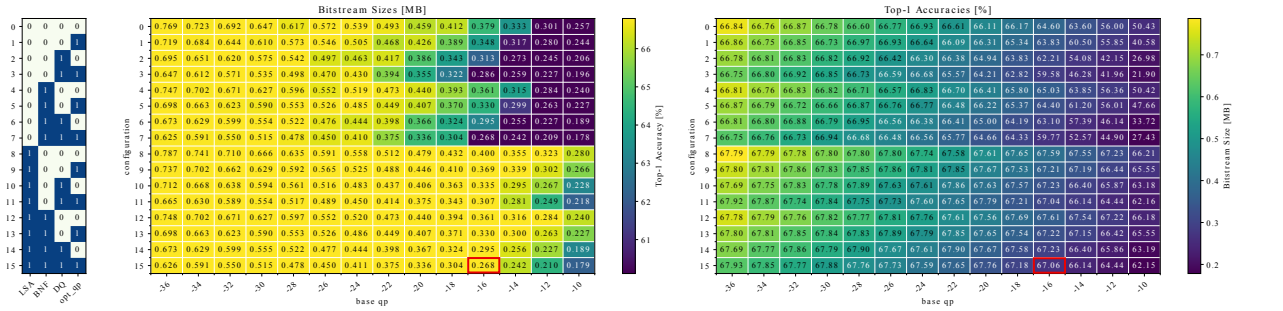

Figure 3. Coding results for ResNet-56 on CIFAR-100 using different combinations of coding tools.

5. Coding Results

Implementation aspects and details on the experimental setup, data and models can be found in appendix A.1.

5.1. Analysis of Coding Tool Configurations

Fig. 3 depicts the coding results for ResNet-56, pre-trained on CIFAR-100 to 66.79% top-1 accuracy. The left table in Fig. 3 lists the 16 combinations of enabled tools. Then, the left matrix shows — for each of this combinations — the resulting bitstream sizes (as text) and the achieved accuracies after reconstruction of the decoded NN (color-coded). On the x-axes, the base qp spans a range of different quantization step sizes Δ from fine ($qp = -36$) to coarse ($qp = -10$). The actual qp per parameter can deviate from the base qp, e.g., for non-weight parameters (cp. appendix A.3), or if *opt_qp* is enabled. The matrix on the right shows inverted results, i.e., accuracies in text and bitstream sizes color-coded. The left matrix's color codes are scaled such that all coded NNs without accuracy degradation are highlighted in yellow. This color-based, row-wise boundary allows for easier indentification of an appropriate rate-distortion-result across all combinations, e.g., the NN coded with tool combination 15 and $qp = -16$ (highlighted by a red patch in Fig. 3). In appendix A.2, we present further coding details and results (including tasks on ImageNet).

5.2. Comparative Results

The Shannon entropy marks a lower bound for the average number of bits required to represent one symbol (here, the integer representation of a weight $\hat{w}$). The entropy of a quantized NN parameter $\hat{\mathcal{W}}$ can be defined as $H_P(\hat{\mathcal{W}}) = -\sum_{\hat{w}\in\mathcal{C}} P(\hat{w}) \log_2 P(\hat{w})$ where $\mathcal{C}$ is a set of quantization levels and the probability value $P(\hat{w})$ of $\hat{w}$ is the number of its occurrences in $\hat{\mathcal{W}}$, divided by $\#\mathcal{W}$ (the count of all its elements).

We encoded the data payload of each unit separately, applying bzip2, Huffman code and NNCodec, and compared the resulting bitstream sizes to the Shannon bound $H_P(\hat{\mathcal{W}})$, as shown in Fig. 4. Huffman code is an optimal entropy code, however with long sequences of symbols, e.g., for NN layers with large $\#\mathcal{W}$, the codebook (*Huffman tree*) becomes very large which is relevant for ResNet's deeper located layers

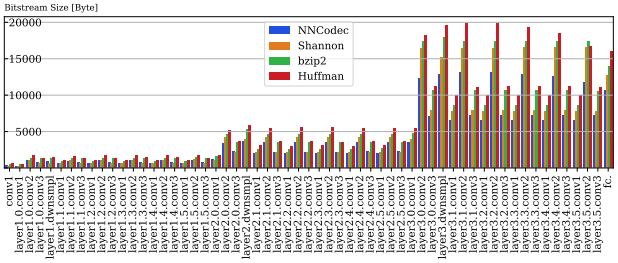

Figure 4. Comparison of resulting bitstream sizes when coding the quantized ($qp = -14$) ResNet-56 tensor-wise using NNCodec vs. Huffman vs. bzip2, compared to the Shannon entropy bound.

(cp. Fig. 4). bzip2 also deploys Huffman code, however, it uses additional run-length (delta) encodings and transforms on top. bzip2 is often close to the Shannon entropy bound but always above. Fig. 4 confirms that NNCodec bitstream lenghts are mostly below the Shannon bound, which is achieved via adjusting various probability estimators of the context models *during* codestreaming. Only bitstreams containing a very small number of weight elements can be above the bound, because they do not provide sequences long enough to sufficiently adapt the context models to the distribution of the coded bins.

6. Conclusion

We presented *NNCodec*, the first open source software implementation of the recently published ISO/IEC NNC standard. The paper gives an overview of the software architecture and tools.We analyzed NN data distributions throughout the coding process and showed that NNCodec's average codeword length is often below the Shannon entropy bound and thus outperforms other common NN coding strategies used for NN compression. We explored NNCodec's hyperparameter space and the resulting compression ratios for varying NN architectures. An NNCodec demo will be prepared for the ICML Neural Compression Workshop.

With this work we want to encourage the Machine Learning community to use this ISO/IEC standard-compliant compression software in their AI pipelines. The 2nd edition of the NNC standard is in progress and additionally targets coding of differential NN updates in distributed scenarios.

References

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. Imagenet: A large-scale hierarchical image database. In *Proc. of the IEEE/CVF Conf. on Computer Vision and Pattern Recognition*, pp. 248–255, 2009.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

Haase, P., Matlage, S., Kirchhoffer, H., Bartnik, C., Schwarz, H., Marpe, D., and Wiegand, T. State-based multi-parameter probability estimation for context-based adaptive binary arithmetic coding. In *2020 Data Compression Conf. (DCC)*, pp. 163–172, 2020a.

Haase, P., Schwarz, H., Kirchhoffer, H., Wiedemann, S., Marinc, T., Marban, A., Müller, K., Samek, W., Marpe, D., and Wiegand, T. Dependent scalar quantization for neural network compression. In *2020 IEEE Int. Conf. on Image Processing (ICIP)*, pp. 36–40, 2020b.

Haase, P., Becking, D., Kirchhoffer, H., Müller, K., Schwarz, H., Samek, W., Marpe, D., and Wiegand, T. Encoder optimizations for the nnr standard on neural network compression. In *2021 IEEE Int. Conf. on Image Processing (ICIP)*, pp. 3522–3526, 2021.

Han, S., Mao, H., and Dally, W. J. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In Bengio, Y. and LeCun, Y. (eds.), *4th Int. Conf. on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conf. Track Proc.*, 2016.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proc. of the IEEE/CVF Conf. on Computer Vision and Pattern Recognition*, pp. 770–778, 2016.

Huffman, D. A. A method for the construction of minimum-redundancy codes. *Proc. of the IRE*, 40(9):1098–1101, 1952.

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Int. Conf. on Machine Learning*, pp. 448–456. pmlr, 2015.

ISO. Information technology — Multimedia content description interface — Part 17: Compression of neural networks for multimedia content description and analysis. Standard ISO/IEC 15938-17:2022, Int. Organization for Standardization, Geneva, CH, 2022. URL https://www.iso.org/standard/78480.html.

Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H., and Kalenichenko, D. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proc. of the IEEE/CVF Conf. on Computer Vision and Pattern Recognition*, June 2018.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. In *3rd Int. Conf. on Learning Representations*, 2015.

Kirchhoffer, H., Haase, P., Samek, W., Müller, K., Rezazadegan-Tavakoli, H., Cricri, F., Aksu, E. B., Hannuksela, M. M., Jiang, W., Wang, W., Liu, S., Jain, S., Hamidi-Rad, S., Racapé, F., and Bailer, W. Overview of the neural network compression and representation (nnr) standard. *IEEE Trans. on Circuits and Systems for Video Technology*, 32(5):3203–3216, 2022.

Krizhevsky, A. Learning Multiple Layers of Features from Tiny Images. April 2009.

Marpe, D., Schwarz, H., and Wiegand, T. Context-based adaptive binary arithmetic coding in the h.264/avc video compression standard. *IEEE Trans. on Circuits and Systems for Video Technology*, 13(7):620–636, 2003.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Inf. Processing Syst.*, 32, 2019.

Shannon, C. E. A Mathematical Theory of Communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.

Sullivan, G. J., Ohm, J.-R., Han, W.-J., and Wiegand, T. Overview of the high efficiency video coding (hevc) standard. *IEEE Trans. on Circuits and Systems for Video Technology*, 22(12):1649–1668, 2012.

Tan, M. and Le, Q. Efficientnet: Rethinking model scaling for convolutional neural networks. In *Int. Conf. on Machine Learning*, pp. 6105–6114. PMLR, 2019.

Wiedemann, S., Kirchhoffer, H., Matlage, S., Haase, P., Marban, A., Marinč, T., Neumann, D., Nguyen, T., Schwarz, H., Wiegand, T., Marpe, D., and Samek, W. Deepcabac: A universal compression algorithm for deep neural networks. *IEEE Journal of Selected Topics in Signal Processing*, 14(4):700–714, 2020.

Wiegand, T., Schwarz, H., et al. Source coding: Part i of fundamentals of source and video coding. *Foundations and Trends® in Signal Processing*, 4(1–2):1–222, 2011.

A. Appendix

A.1. Implementation Aspects and Experimental Setup

A.1.1. Datasets and Models

We perform experiments using CIFAR-100 (Krizhevsky, 2009) and ImageNet-1k (Deng et al., 2009) datasets. Standard data pre-processing is used, i.e., normalization and cropping. For CIFAR-100, we additionally use random horizontal flipping for the training data splits. Furthermore, we use 50.000 and 10.000 of the ImageNet and CIFAR training data samples as validation dataset to optimize local scaling parameters (during LSA). The batch size was set to 64 in all experiments.

For the ImageNet task, we used models from the torchvision model zoo [1]. For the CIFAR-100 task, we adapted a ResNet-56 (He et al., 2016) PyTorch implementation by Yerlan Idelbayev [2] and trained it via Adam (Kingma & Ba, 2015) optimization for 150 epochs.

A.1.2. Environment

We deployed the PyTorch (Paszke et al., 2019) deep learning framework, version 1.13.1 and torchvision version 0.14.1, accordingly. All experiments were conducted on a homogeneous GPU cluster equipped with NVIDIA Ampere A100 GPUs (40 GB RAM) using CUDA Version 11.7.

A.1.3. Hyperparameter Settings and Other Implementation Aspects

For data-driven compression with LSA, the scaling parameters s are optimized using Adam (Kingma & Ba, 2015) optimization with an initial learning rate of $1e-3$. For ImageNet, 10 epochs of LSA were applied, for CIFAR-100 30 epochs.

Coding: For the *Huffman* code implementation, we used the *dippykit* digital image processing library[3], licensed under the GNU General Public License v3.0. Since the tool does not compute the size of the Huffman tree, we used an approximation by accounting each symbol (i.e., integer weight) with 32 bits and accounting its binary code representation with the number of bits as long as the codeword. In reality, storing such a code table on common hardware requires considerably more memory. *bzip2*[4] is an open source, patent free data compressor. We used the tool out of the box, as is.

For converting the integer qp value into a floating point step size Δ, the formula

$$\Delta = 4 + (qp \bmod 4) \cdot 2^{\lfloor \frac{qp}{4} \rfloor - 2} \tag{2}$$

can be used. Note that it only applies if NNCodec's hyperparameter *qp_density*$= 2$, which is the default value that does not need to be changed for the vast majority of applications.

A.2. Additional Results

Table 1. Comparison of resulting bitstream sizes using different codes for full NN coding.

model	data	orig. acc. [%]	orig. size	config	base qp	acc.↓ [%]	NNCodec	Shannon	bzip2	Huffman
ResNet-56	CIFAR-100	66.79	2.51 MB	14	-14	-0.39	**255.71 kB**	298.91 kB	389.63 kB	354.68 kB
ResNet-50	ImageNet	80.34	102.55 MB	15	-30	-1.01	**7.62 MB**	10.11 MB	10.77 MB	10.55 MB
EfficientNet-B0	ImageNet	77.67	21.45 MB	15	-30	-0.89	**2.41 MB**	2.87 MB	3.43 MB	3.26 MB
ViT-B/16	ImageNet	81.07	346.27 MB	3	-32	-1.12	**32.87 MB**	41.63 MB	47.22 MB	42.60 MB

We can find in all experiments a coding tool combination which is lossless in terms of not degrading the original accuracy at all. Sometimes the reconstructed model even achieves a little higher accuracy, which can be due to regularizing effects of compression but also due to the local scaling adaptation. However, for higher compression gains it might be plausible to sacrifice some model performance. The ImageNet results are depicted in Fig. 5 for ResNet50 (He et al., 2016), Fig. 6 for EfficientNet-B0 (Tan & Le, 2019) and Fig. 7 for ViT-B/16 (Dosovitskiy et al., 2020).

[1] https://pytorch.org/vision/stable/models.html
[2] https://github.com/akamaster/pytorch_resnet_cifar10
[3] https://github.com/dippykit/dippykit
[4] https://sourceware.org/bzip2

Table 1 compares the different coding methods in terms of resulting bitstream sizes. We chose the coding tool configuration and *qp* value of the exemplary NNs in Table 1 such that the compression is high and the performance degradation of the model is in the range of 1% at most.

As a Vision Transformer, we used the ViT-B/16 model, which makes use of Layer Normalization instead of Batch Normalization. Consequently, our BNF tool has no effect. For the EfficientNet (Tan & Le, 2019) experiments, only fully-connected layers are equipped with trainable scaling parameters, since PyTorch has a custom class object for their EfficientNet convolution layers ("Conv2dNormActivation"[5]) and our LSA implementation only seeks for torch.nn.Conv2d and torch.nn.Linear modules to be replaced with our so-called *ScaledConv2d* and *ScaledLinear* objects.

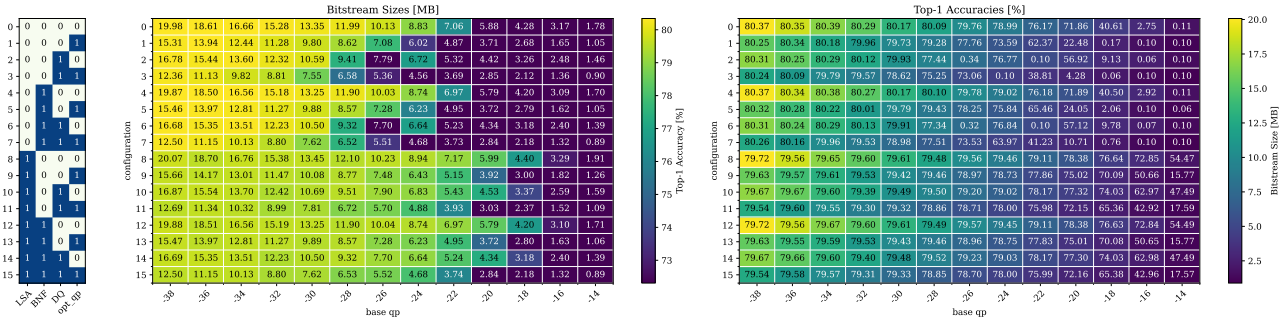

Figure 5. Coding results for ResNet-50 on ImageNet

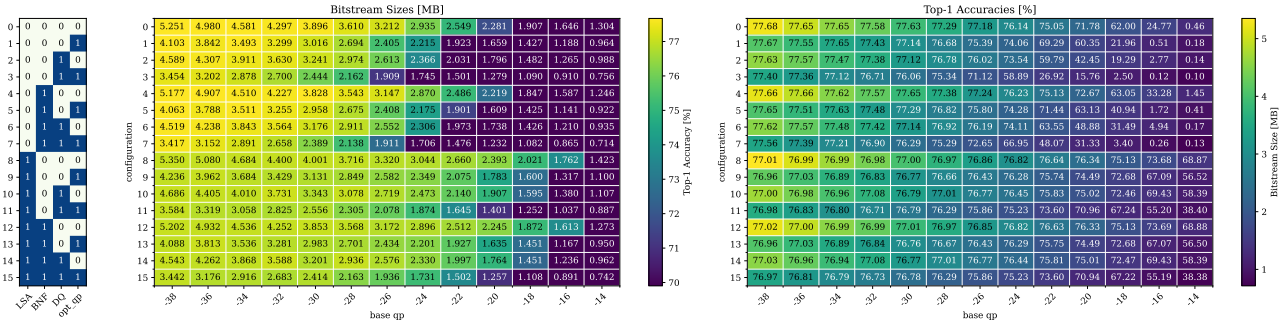

Figure 6. Coding results for EfficientNet-B0 on ImageNet

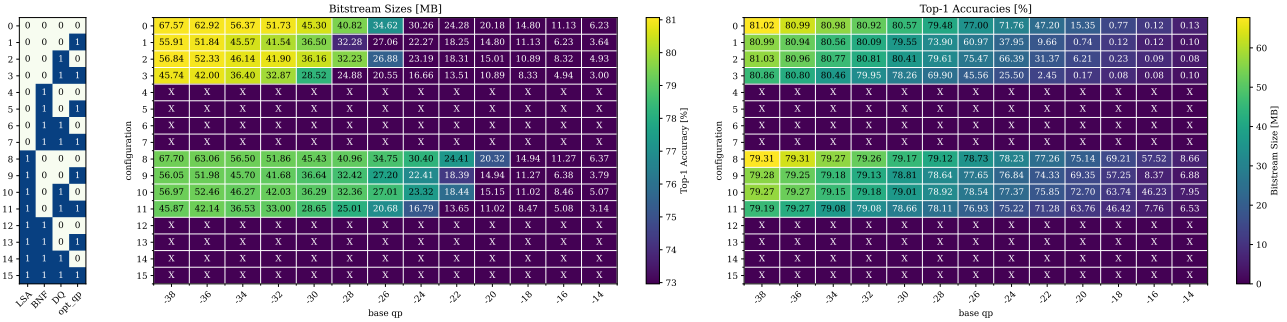

Figure 7. Coding results for ViT-B/16 on ImageNet

[5] https://pytorch.org/vision/main/generated/torchvision.ops.Conv2dNormActivation.html

A.3. Weight Distributions of ResNet-56

This section shows histogram plots of all ResNet-56 parameters. Fig. 8, Fig. 9 and Fig. 10 depict the distributions of the original ($\mathcal{W}$), quantized ($\hat{\mathcal{W}}$) and reconstructed ($\hat{\mathcal{W}}\alpha \cdot qp$) weight parameters (i.e., convolutional and fully-connected layers). Fig. 11, Fig. 12 and Fig. 13 show the distributions of the original ($beta$, E, γ, Var), merged & quantized (δ, α) and reconstructed ($\hat{\beta}$) non-weight parameters, respectively. "Non-weight parameters" here includes all BatchNorm parameters (trainable parameters and statistics buffers), scaling parameters, bias parameters and merged (e.g., BN-fold) versions of the previously mentioned.

Since a very fine quantization step size is used for the non-weight parameters (defaults to $qp = -75$ because they are much more sensitive to compression), the elements are often assigned to a quantization level alone, meaning that the probability $P(\hat{w})$ is identical for all elements in that case. However, the non-weight parameters in sum often account for less than 1% of the total network parameters and therefore do not contribute much to the overall bitstream size, but can have a great impact on model performance.

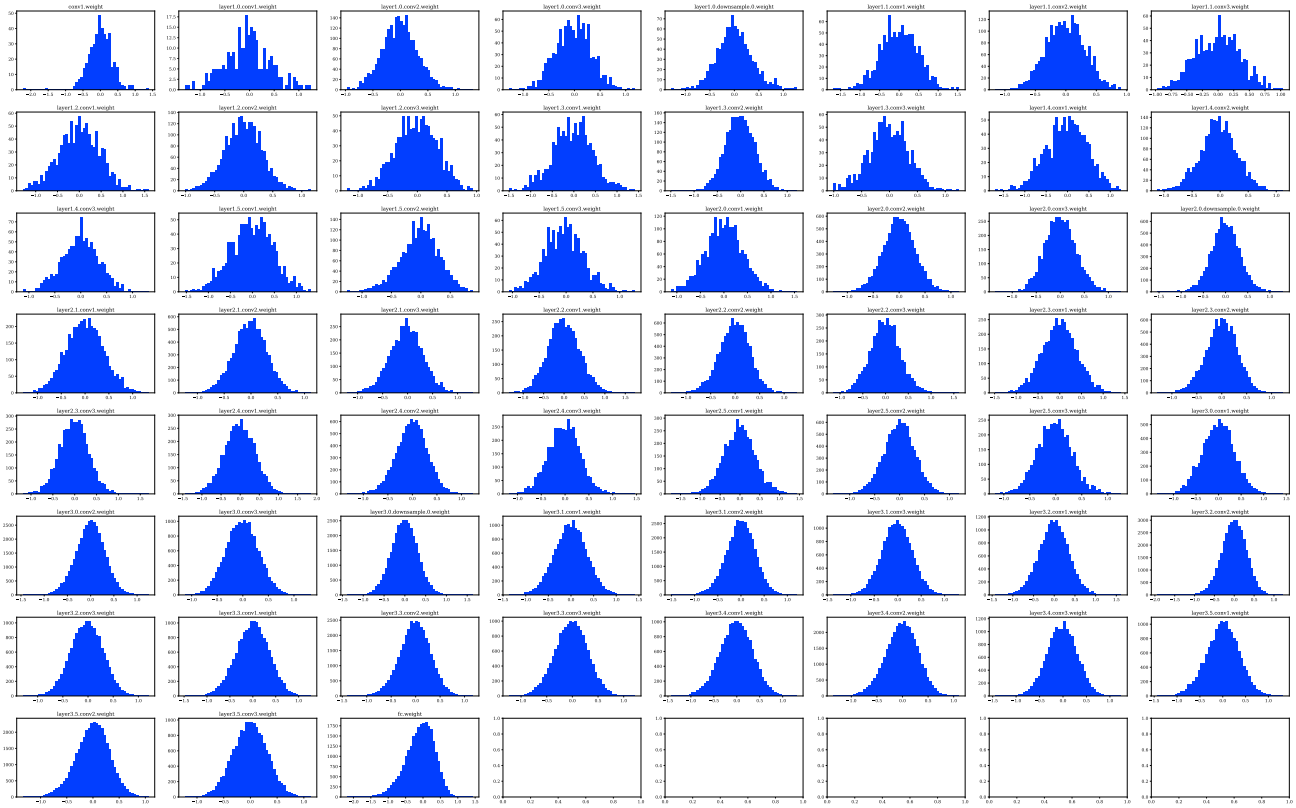

Figure 8. Original weight tensor distributions of ResNet-56.

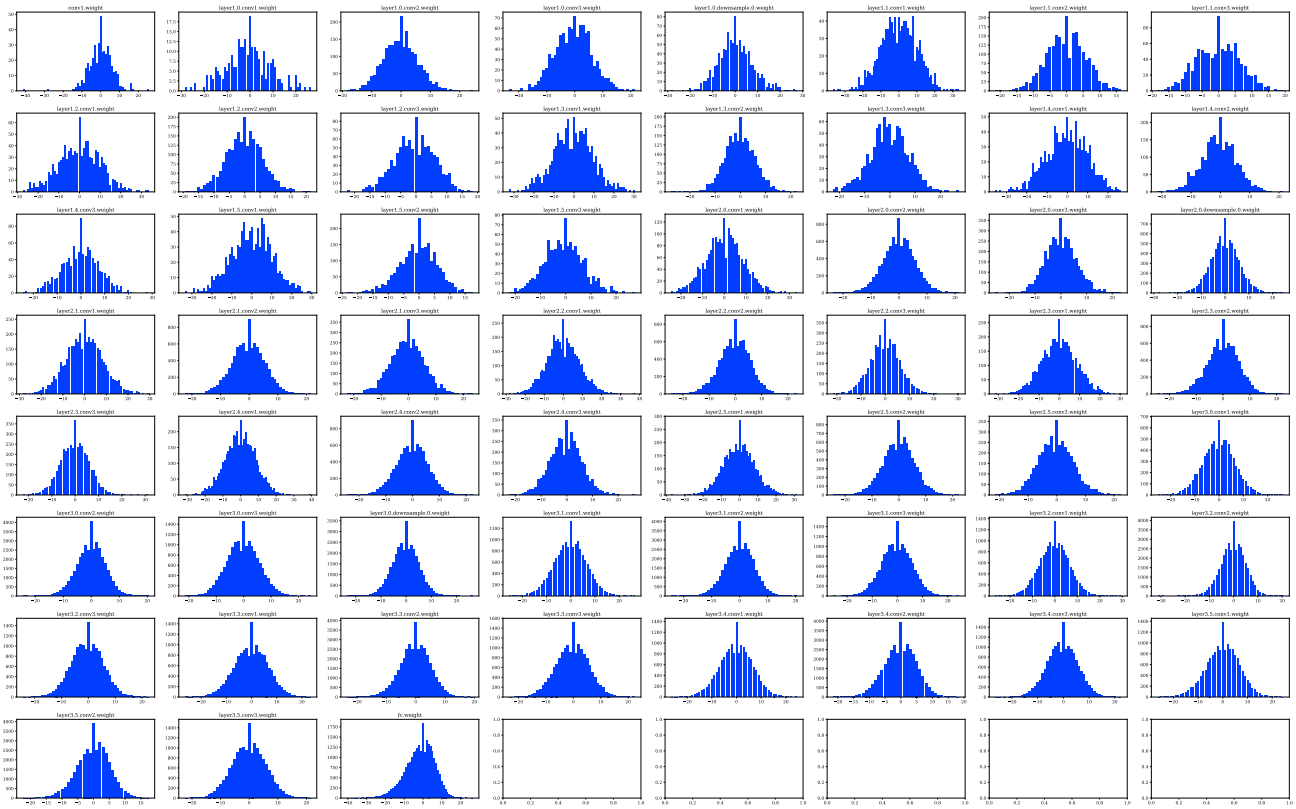

Figure 9. Quantized weight tensor distributions of ResNet-56 using dependent quantization (DQ) with $qp = -18$ and *opt_qp*.

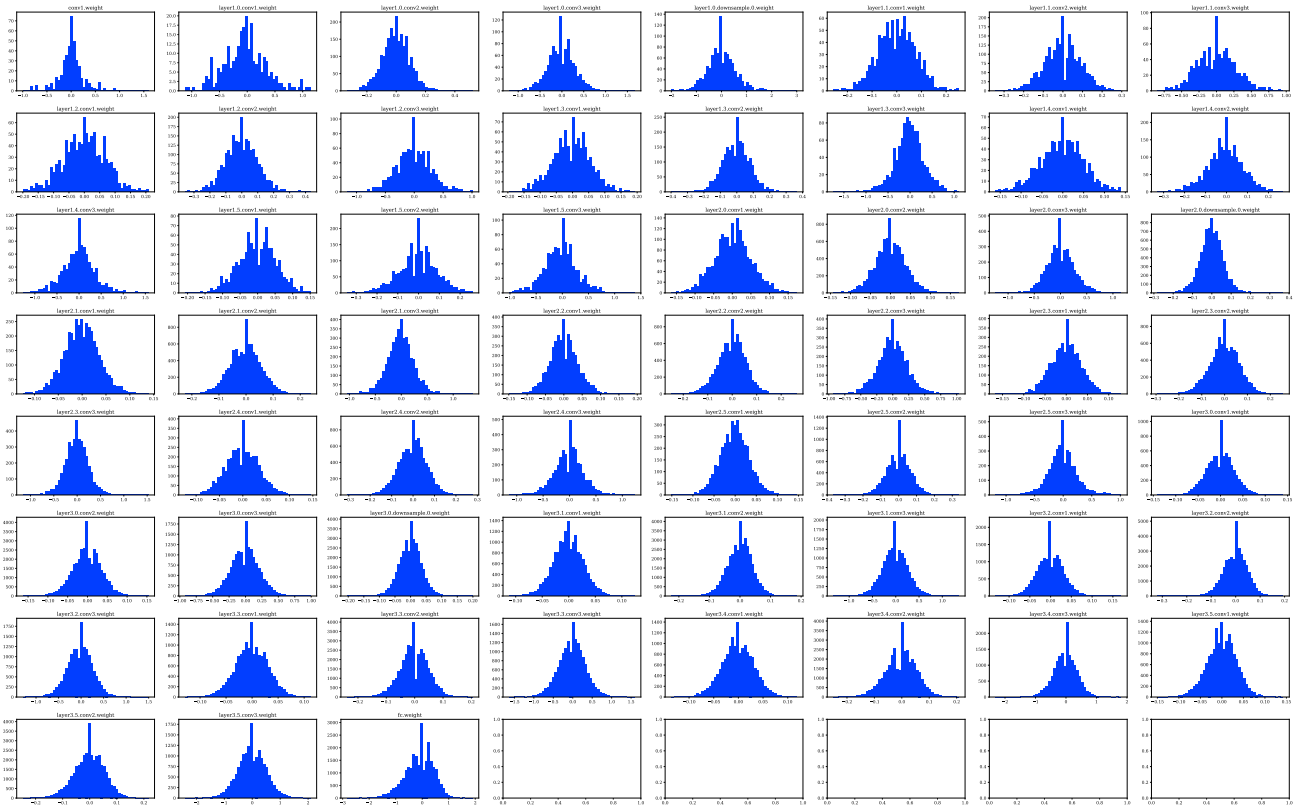

Figure 10. Decoded and reconstructed weight tensor distributions of ResNet-56.

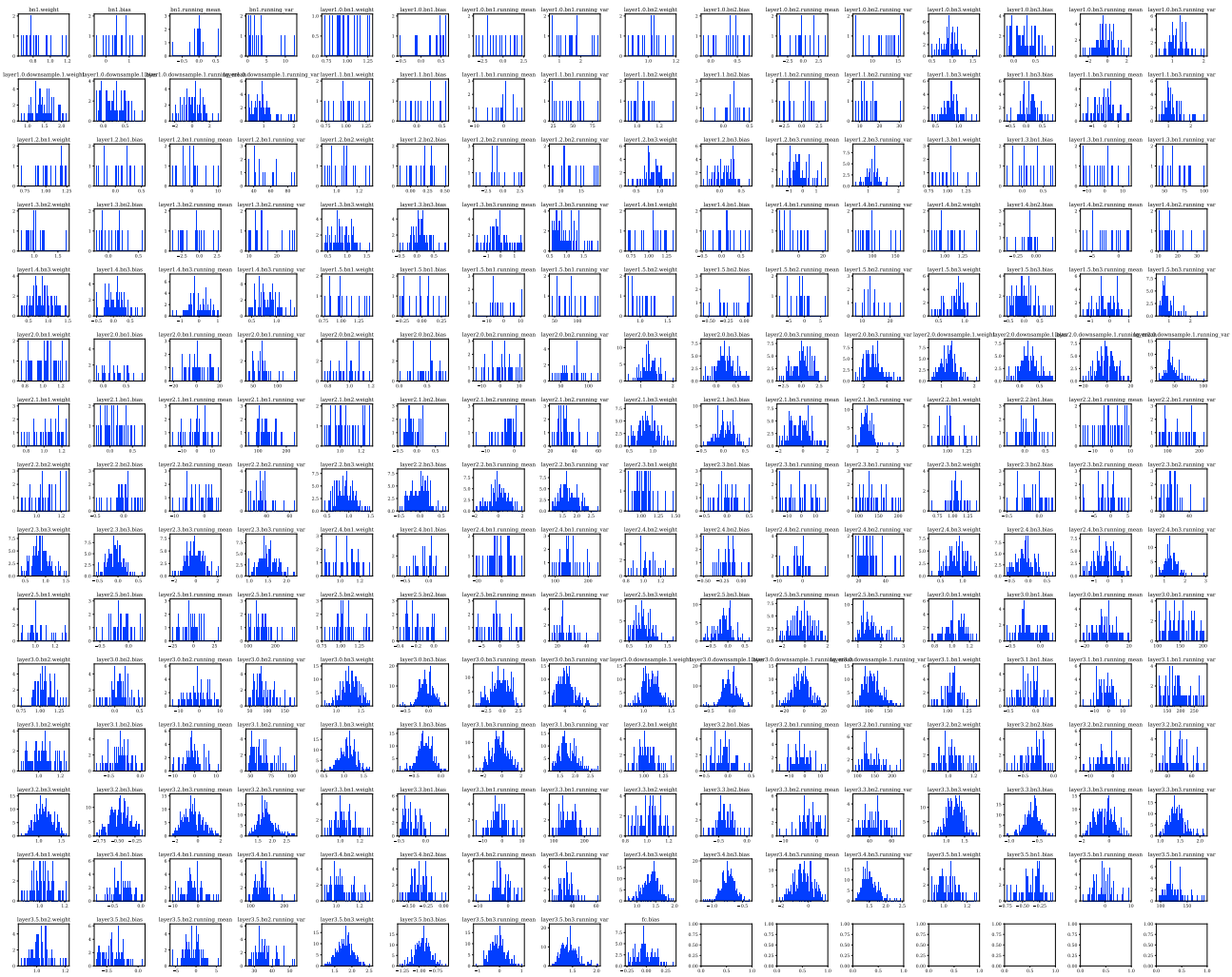

Figure 11. Original weight vector and buffer (running statistics) distributions of ResNet-56.

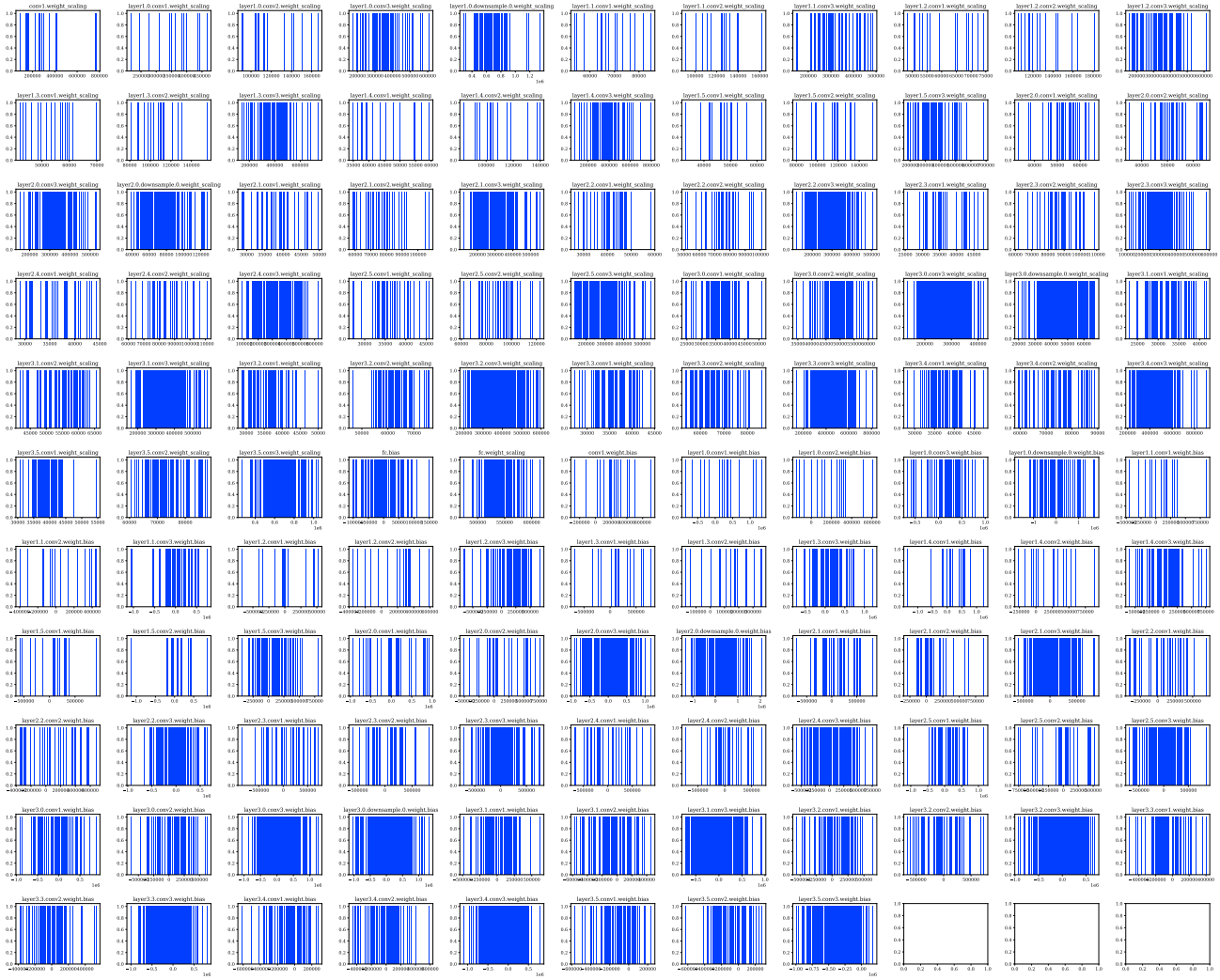

Figure 12. Quantized weight vector and buffer (running statistics) distributions of ResNet-56 using dependent quantization (DQ) with $qp = -18$ and *opt_qp*.

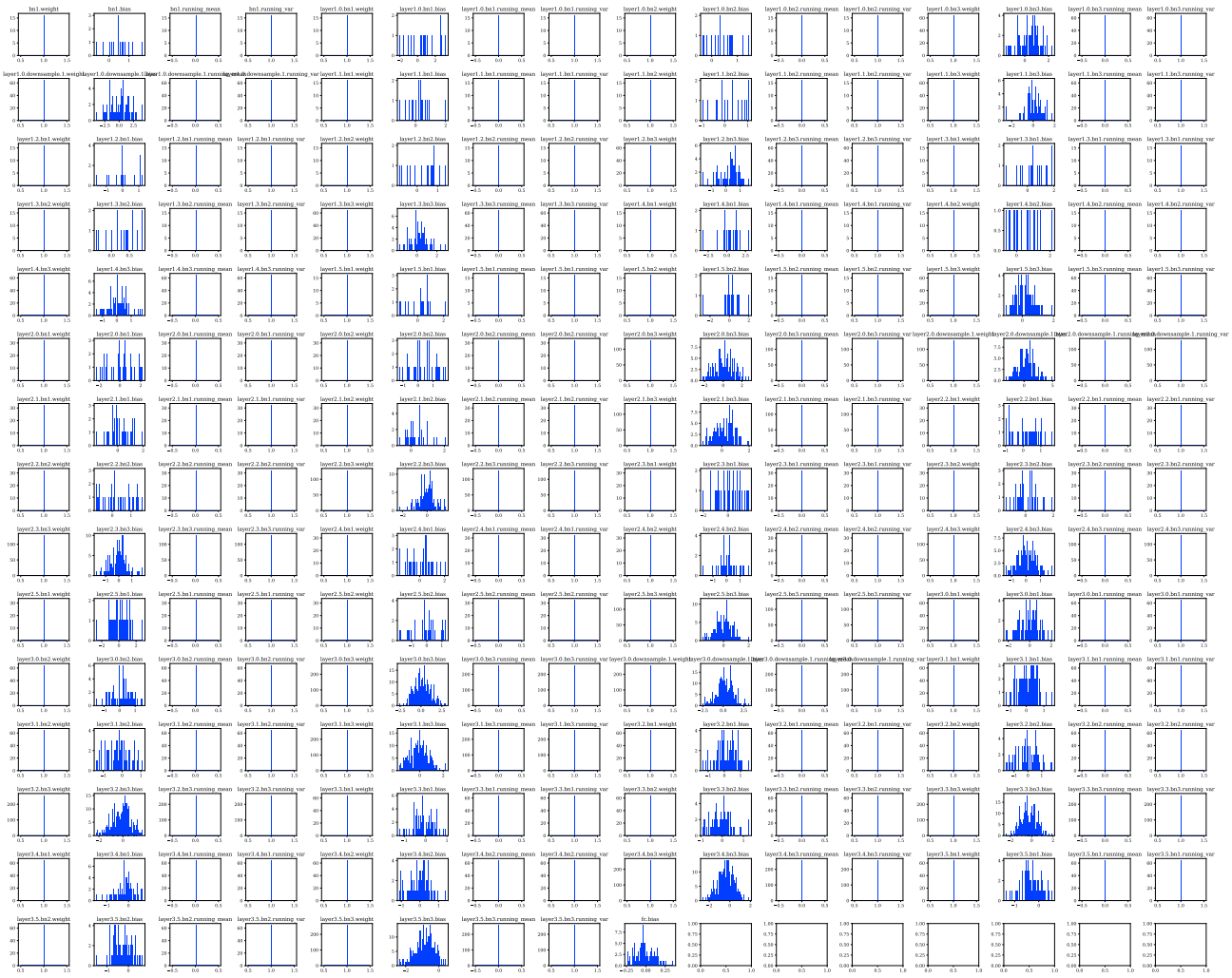

Figure 13. Decoded and reconstructed weight vector and buffer (running statistics) distributions of ResNet-56.