



International
Standard

ISO/IEC 15938-17

**Information technology —
Multimedia content description
interface —**

**Part 17:
Compression of neural networks for
multimedia content description and
analysis**

*Technologies de l'information — Interface de description du
contenu multimédia —*

*Partie 17: Compression des réseaux neuronaux pour la
description et l'analyse du contenu multimédia*

**Second edition
2024-01**

Reference number
ISO/IEC 15938-17:2024(en)

Full document details: ISO IEC 15938-17-2024

 **Documents**
© ISO/IEC 2024 Document Sharing Hub



COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2024

All rights reserved. Unless otherwise specified, or required in the context of its implementation, no part of this publication may be reproduced or utilized otherwise in any form or by any means, electronic or mechanical, including photocopying, or posting on the internet or an intranet, without prior written permission. Permission can be requested from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland

© ISO/IEC 2024 – All rights reserved

Contents

Page

Foreword	v
Introduction	vi
1 Scope	1
2 Normative references	1
3 Terms and definitions	1
4 Abbreviated terms, conventions and symbols	3
4.1 General	3
4.2 Abbreviated terms	3
4.3 List of symbols	3
4.4 Number formats and computation conventions	6
4.5 Arithmetic operators	6
4.6 Logical operators	7
4.7 Relational operators	7
4.8 Bit-wise operators	7
4.9 Assignment operators	8
4.10 Range notation	8
4.11 Mathematical functions	8
4.12 Array functions	9
4.13 Order of operation precedence	11
4.14 Variables, syntax elements and tables	11
5 Overview	13
5.1 General	13
5.2 Compression tools	13
5.3 Creating encoding pipelines	14
6 Syntax and semantics	15
6.1 Specification of syntax and semantics	15
6.1.1 Method of specifying syntax in tabular form	15
6.1.2 Bit ordering	16
6.1.3 Specification of syntax functions and data types	16
6.1.4 Semantics	17
6.2 General bitstream syntax elements	18
6.2.1 NNR unit	18
6.2.2 Aggregate NNR unit	18
6.2.3 Composition of NNR bitstream	19
6.3 NNR bitstream syntax	20
6.3.1 NNR unit syntax	20
6.3.2 NNR unit size syntax	20
6.3.3 NNR unit header syntax	20
6.3.4 NNR unit payload syntax	25
6.3.5 Byte alignment syntax	31
6.4 Semantics	31
6.4.1 General	31
6.4.2 NNR unit size semantics	31
6.4.3 NNR unit header semantics	31
6.4.4 NNR unit payload semantics	39
7 Decoding process	45
7.1 General	45
7.2 NNR decompressed data formats	46
7.3 Decoding methods	47
7.3.1 General	47
7.3.2 Decoding method for NNR compressed payloads of type NNR_PT_INT	47
7.3.3 Decoding method for NNR compressed payloads of type NNR_PT_FLOAT	48

7.3.4	Decoding method for NNR compressed payloads of type NNR_PT_RAW_FLOAT	48
7.3.5	Decoding method for NNR compressed payloads of type NNR_PT_BLOCK	49
7.3.6	Decoding process for an integer weight tensor	50
8	Parameter reduction	51
8.1	General	51
8.2	Methods	51
8.2.1	Batchnorm folding	51
8.3	Syntax and semantics	52
8.3.1	Sparsification using compressibility loss	52
8.3.2	Sparsification using micro-structured pruning	52
8.3.3	Combined pruning and sparsification	52
8.3.4	Unstructured statistics-adaptive sparsification	53
8.3.5	Structured sparsification (global and local approach)	53
8.3.6	Weight unification	53
8.3.7	Low rank/low displacement rank for convolutional and fully connected layers	54
8.3.8	Batchnorm folding	54
8.3.9	Local scaling adaptation (LSA)	54
9	Parameter quantization	55
9.1	General	55
9.2	Methods	55
9.2.1	Uniform quantization method	55
9.2.2	Codebook-based method	55
9.2.3	Dependent scalar quantization method	55
9.2.4	Predictive residual encoding (PRE)	55
9.3	Syntax and semantics	55
9.3.1	Uniform quantization method	55
9.3.2	Codebook-based method	56
9.3.3	Dependent scalar quantization method	56
10	Entropy coding	56
10.1	Methods	56
10.1.1	DeepCABAC	56
10.2	Syntax and semantics	58
10.2.1	DeepCABAC syntax	58
10.3	Entropy decoding process	64
10.3.1	General	64
10.3.2	Initialization process	64
10.3.3	Binarization process	65
10.3.4	Decoding process flow	66
	Annex A (normative) Implementation for NNEF	73
	Annex B (informative) Implementation for ONNX®	75
	Annex C (informative) Implementation for PyTorch®	77
	Annex D (informative) Implementation for TensorFlow®	79
	Annex E (informative) Recommendation for carriage of NNR bitstreams in other containers	81
	Annex F (informative) Recommendation for naming method regarding performance metric type	83
	Annex G (informative) Encoding side information for selected compression tools	84
	Bibliography	95

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives or www.iec.ch/members_experts/refdocs).

ISO and IEC draw attention to the possibility that the implementation of this document may involve the use of (a) patent(s). ISO and IEC take no position concerning the evidence, validity or applicability of any claimed patent rights in respect thereof. As of the date of publication of this document, ISO and IEC had received notice of (a) patent(s) which may be required to implement this document. However, implementers are cautioned that this may not represent the latest information, which may be obtained from the patent database available at www.iso.org/patents and <https://patents.iec.ch>. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see www.iso.org/iso/foreword.html. In the IEC, see www.iec.ch/understanding-standards.

This document was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 29, *Coding of audio, picture, multimedia and hypermedia information*.

This second edition cancels and replaces the first edition (ISO/IEC 15938-17:2022), which has been technically revised.

The main changes are as follows:

- Support for incremental compression of updates of neural networks respective to a base model,
- Additional sparsification tools,
- Additional entropy coding tools, leveraging dependencies in incremental updates,
- Additional quantization tools, including representation as residuals of updates, and
- Additional high-level syntax, covering the new coding tools as well as more metadata (e.g. performance metrics).

A list of all parts in the ISO/IEC 15938 series can be found on the ISO and IEC websites.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html and www.iec.ch/national-committees.

Introduction

Artificial neural networks have been adopted for a broad range of tasks in multimedia analysis and processing, media coding, data analytics and many other fields. Their recent success is based on the feasibility of processing much larger and complex neural networks (deep neural networks, DNNs) than in the past, and the availability of large-scale training data sets. As a consequence, trained neural networks contain a large number of parameters and weights, resulting in a quite large size (e.g. several hundred MBs). Many applications require the deployment of a particular trained network instance, potentially to a larger number of devices, which may have limitations in terms of processing power and memory (e.g. mobile devices or smart cameras), and also in terms of communication bandwidth. Any use case, in which a trained neural network (or its updates) needs to be deployed to a number of devices thus benefits from a standard for the compressed representation of neural networks.

Considering the fact that compression of neural networks is likely to have a hardware dependent and hardware independent component, this document is designed as a toolbox of compression technologies. Some of these technologies require specific representations in an exchange format (i.e. sparse representations, adaptive quantization), and thus a normative specification for representing outputs of these technologies is defined. Others do not at all materialize in a serialized representation (e.g. pruning), however, also for the latter ones required metadata is specified. This document is independent of a particular neural network exchange format, and interoperability with common formats is described in the annexes.

This document thus defines a high-level syntax that specifies required metadata elements and related semantics. In cases where the structure of binary data is to be specified (e.g. decomposed matrices) this document also specifies the actual bitstream syntax of the respective block. Annexes to the document specify the requirements and constraints of compressed neural network representations; as defined in this document; and how they are applied.

- [Annex A](#) specifies the implementation of this document with the Neural Network Exchange Format (NNEF¹⁾), defining the use of NNEF to represent network topologies in a compressed neural network bitstream.
- [Annex B](#) provides recommendations for the implementation of this document with the Open Neural Network Exchange Format (ONNX²⁾), defining the use of ONNX to represent network topologies in a compressed neural network bitstream.
- [Annex C](#) provides recommendations for the implementation of this document with the PyTorch³⁾ format, defining the reference to PyTorch elements in the network topology description of a compressed neural network bitstream.
- [Annex D](#) provides recommendations for the implementation of this document with the Tensorflow⁴⁾ format, defining the reference to Tensorflow elements in the network topology description of a compressed neural network bitstream.
- [Annex E](#) provides recommendations for the carriage of tensors compressed according to this document in third party container formats.
- [Annex F](#) provides recommendations for the naming of common performance metrics to specify the metric that was used for validation.

1) NNEF is the trademark of a product owned by The Khronos® Group. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO/IEC of the product named.

2) ONNX is the trademark of a product owned by LF PROJECTS, LLC. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO/IEC of the product named.

3) PyTorch is the trademark of a product supplied by Facebook, Inc. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO/IEC of the product named.

4) TensorFlow is the trademark of a product supplied by Google LLC. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO/IEC of the product named.

- [Annex G](#) provides recommendations for implementing the encoding side of some of the compression tools.

The compression tools described in this document have been selected and evaluated for neural networks used in applications for multimedia description, analysis and processing. However, they may be useful for the compression of neural networks used in other applications and applied to other types of data.

Information technology — Multimedia content description interface —

Part 17:

Compression of neural networks for multimedia content description and analysis

1 Scope

This document specifies Neural Network Coding (NNC) as a compressed representation of the parameters/weights of a trained neural network and a decoding process for the compressed representation, complementing the description of the network topology in existing (exchange) formats for neural networks. It establishes a toolbox of compression methods, specifying (where applicable) the resulting elements of the compressed bitstream. Most of these tools can be applied to the compression of entire neural networks, and some of them can also be applied to the compression of differential updates of neural networks with respect to a base network. Such differential updates are for example useful when models are redistributed after fine-tuning or transfer learning, or when providing versions of a neural network with different compression ratios.

This document does not specify a complete protocol for the transmission of neural networks, but focuses on compression of network parameters. Only the syntax format, semantics, associated decoding process requirements, parameter sparsification, parameter transformation methods, parameter quantization, entropy coding method and integration/signalling within existing exchange formats are specified, while other matters such as pre-processing, system signalling and multiplexing, data loss recovery and post-processing are considered to be outside the scope of this document. Additionally, the internal processing steps performed within a decoder are also considered to be outside the scope of this document; only the externally observable output behaviour is required to conform to the specifications of this document.

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 10646, *Information technology — Universal coded character set (UCS)*

ISO/IEC 60559, *Information technology — Microprocessor Systems — Floating-Point arithmetic*

IETF RFC 1950, *ZLIB Compressed Data Format Specification version 3.3*

NNEF-v1.0.3⁵⁾, *Neural Network Exchange Format*, The Khronos NNEF Working Group, Version 1.0.3, 2020

FIPS PUB 180-4:2015, *Secure Hash Standard (SHS)*

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

5) Available from: <https://www.khronos.org/registry/NEF/specs/1.0/nnef-1.0.3.pdf>

ISO and IEC maintain terminology databases for use in standardization at the following addresses:

- ISO Online browsing platform: available at <https://www.iso.org/obp>
- IEC Electropedia: available at <https://www.electropedia.org/>

3.1

aggregate NNR unit

NNR unit which carries multiple NNR units in its payload

3.2

base neural network

neural network serving as reference for a differential update

3.3

compressed neural network representation

NNR

representation of a neural network with model parameters encoded using compression tools

3.4

decomposition

transformation to express a tensor as product of two tensors

3.5

hyperparameter

parameter whose value is used to control the learning process

3.6

layer

collection of nodes operating together at a specific depth within a neural network

3.7

model parameter

coefficients of the neural network model such as weights and biases

3.8

NNR unit

data structure for carrying (compressed or uncompressed) neural network data and related metadata

3.9

parameter identifier

value that uniquely identifies a parameter throughout different incremental updates

Note 1 to entry: Parameters having the same parameter identifier are at the same position in the same tensor in different incremental updates. This means they are co-located.

3.10

pruning

reduction of parameters in (a part of) the neural network

3.11

sparsification

increase of the number of zero-valued entries of a tensor

3.12

tensor

multidimensional structure grouping related model parameters

3.13

updated neural network

neural network resulting from modifying the base neural network

Note 1 to entry: The updated neural network is reconstructed by applying the differential update to the base neural network.

4 Abbreviated terms, conventions and symbols

4.1 General

This subclause contains the definition of operators, notations, functions, textual conventions and processes used throughout this document.

The mathematical operators used in this document are similar to those used in the C programming language. However, the results of integer division and arithmetic shift operations are specified more precisely, and additional operations are specified, such as exponentiation and real-valued division. Numbering and counting conventions generally begin from 0, e.g. "the first" is equivalent to the 0-th, "the second" is equivalent to the 1-th, etc.

4.2 Abbreviated terms

DeepCABAC	Context-adaptive binary arithmetic coding for deep neural networks
LDR	Low displacement rank
LPS	Layer parameter set
LR	Low-rank
LSA	Local scaling adaptation
LSB	Least significant bit
MPS	Model parameter set
MSB	Most significant bit
MSE	Mean square error
NN	Neural network
NNC	Neural network coding
NDU	NNR compressed data unit
NNEF	Neural network exchange format
QP	Quantization parameter
PRE	Predictive residual encoding
SBT	Stochastic binary-ternary quantization
SVD	Singular value decomposition

4.3 List of symbols

This document defines the following symbols:

A	Input tensor
B	Output tensor
B_{jl}^k	Block in superblock j of layer k .
b	Bias parameter
C_i	Number of input channels of a convolutional layer
C_o	Number of output channels of a convolutional layer
c_j^k	Number of channels in dimension j of tensor in layer k
$c_j^{k'}$	Derived number of channels in dimension j of tensor in layer k
d_j^k	Depth dimension of tensor at layer k
e	Parameter of f-circulant matrix Z_e
F	Parameter tensor of a convolutional layer
f	Parameter of f-circulant matrix Zf
G_k	Left-hand side matrix of Low Rank decomposed representation of matrix W_k
H_k	Right-hand side matrix of Low Rank decomposed representation of matrix W_k
h_j^k	Height dimension of tensor for layer k
K	Dimension of a convolutional kernel
L	Loss function
L_c	Compressibility loss
L_d	Diversity loss
L_s	Task loss
L_t	Training loss
M	Feature matrix
M_k	Pruning mask for layer k
m	Sparsification hyperparameter
m_i	i -th row of feature matrix M
n_j^k	Kernel size of tensor at layer k .
n^k	Dimension resulting from a product over n_j^k
P	Stochastic transition matrix
p	Pruning ratio hyperparameter
p_{ij}	Elements of transition matrix P
q	Sparsification ratio hyperparameter

q_b	Binary quantization
q_t	Ternary quantization
S	Importance of parameters for pruning
S_j^k	Superblock j in layer k
s	Local scaling factors
s_j^k	Size of superblock j in layer k
T	Topology element
T^q	Quantizable topology element
u	Unification ratio hyperparameter
W	Parameter tensor
ΔW	Difference of parameter tensor
W_l	Weight tensor of l -th layer
W_k	Parameter tensor of layer k
$\hat{W}_k$	Low Rank approximation of W_k
w	Parameter vector
$w_{l,i}$	Vector of weights for the i -th filter in the l -th layer
$w'_{l,i}$	Vector of normalized weights for the i -th filter in the l -th layer
y, y_{ref}	Coding performance, reference coding performance
y_d	Coding performance difference
X	Input to a batch-normalization layer
Z_e	f -circulant matrix
Z_f	f -circulant matrix
α	Folded batch normalization parameter
α'	Combined value for folded batch normalization parameter and local scaling factors
β	Batch normalization parameter
β_u	Updated batch normalization parameter
γ_c	Compressibility loss multiplier
γ	Batch normalization parameter
γ_u	Updated batch normalization parameter
δ	Folded batch normalization parameter
δ_f	Sparsification threshold (mean of filter means)

δ_s	Scaling factor for sparsification
ϵ	Scalar close to zero to avoid division by zero in batch normalization
λ	Eigenvector
λ_c	Compressibility loss weight
λ_d	Diversity loss weight
μ	Batch normalization parameter
v_j^k	Width dimension of tensor for layer k .
π	Equilibrium probability of P
π_t	Probability of applying ternary quantization
ρ	Parameter
σ	Batch normalization parameter
τ	Threshold (sparsification, ternary-binary quantization)
θ_ρ	Weight magnitude threshold
φ	Smoothing factor

4.4 Number formats and computation conventions

This document defines the following number formats:

integer	Integer number which may be arbitrarily small or large. Integers are also referred to as signed integers.
unsigned integer	Unsigned integer that may be zero or arbitrarily large.
float	Floating point number according to ISO/IEC 60559.

If not specified otherwise, outcomes of all operators and mathematical functions are mathematically exact. Whenever an outcome shall be a float, it is explicitly specified.

4.5 Arithmetic operators

The following arithmetic operators are defined:

+	Addition
−	Subtraction (as a two-argument operator) or negation (as a unary prefix operator)
*	Multiplication, including matrix multiplication
◦	Element-wise multiplication of two transposed vectors or element-wise multiplication of a transposed vector with rows of a matrix or Hadamard product of two matrices with identical dimensions
x^y	Exponentiation. Specifies x to the power of y . In other contexts, such notation is used for superscripting not intended for interpretation as exponentiation.

/	Integer division with truncation of the result toward zero. For example, $7 / 4$ and $-7 / -4$ are truncated to 1 and $-7 / 4$ and $7 / -4$ are truncated to -1.
$\div$	Used to denote division in mathematical equations where no truncation or rounding is intended.
$\frac{x}{y}$	Used to denote division in mathematical equations where no truncation or rounding is intended, including element-wise division of two transposed vectors or element-wise division of a transposed vector with rows of a matrix.
$\sum_{i=x}^y f(i)$	The summation of $f(i)$ with i taking all integer values from x up to and including y .
$\prod_{i=x}^y f(i)$	The product of $f(i)$ with i taking all integer values from x up to and including y .
$x \% y$	Modulus. Remainder of x divided by y , defined only for integers x and y with $x \geq 0$ and $y > 0$.

4.6 Logical operators

The following logical operators are defined:

$x \&\& y$	Boolean logical "and" of x and y
$x y$	Boolean logical "or" of x and y
!	Boolean logical "not"
$x ? y : z$	If x is TRUE or not equal to 0, evaluates to the value of y ; otherwise, evaluates to the value of z .

4.7 Relational operators

The following relational operators are defined as follows:

>	Greater than
$\geq$	Greater than or equal to
<	Less than
$\leq$	Less than or equal to
==	Equal to
!=	Not equal to

When a relational operator is applied to a syntax element or variable that has been assigned the value "na" (not applicable), the value "na" is treated as a distinct value for the syntax element or variable. The value "na" is considered not to be equal to any other value.

4.8 Bit-wise operators

The following bit-wise operators are defined as follows:

&	Bit-wise "and". When operating on integer arguments, operates on a two's complement representation of the integer value. When operating on a binary argument that contains fewer bits than another argument, the shorter argument is extended by adding more significant bits equal to 0.
---	---

	Bit-wise "or". When operating on integer arguments, operates on a two's complement representation of the integer value. When operating on a binary argument that contains fewer bits than another argument, the shorter argument is extended by adding more significant bits equal to 0.
^	Bit-wise "exclusive or". When operating on integer arguments, operates on a two's complement representation of the integer value. When operating on a binary argument that contains fewer bits than another argument, the shorter argument is extended by adding more significant bits equal to 0.
$x \gg y$	Arithmetic right shift of a two's complement integer representation of x by y binary digits. This function is defined only for non-negative integer values of y . Bits shifted into the MSBs as a result of the right shift have a value equal to the MSB of x prior to the shift operation.
$x \ll y$	Arithmetic left shift of a two's complement integer representation of x by y binary digits. This function is defined only for non-negative integer values of y . Bits shifted into the LSBs as a result of the left shift have a value equal to 0.
!	Bit-wise not operator returning 1 if applied to 0 and 0 if applied to 1.

4.9 Assignment operators

The following arithmetic operators are defined as follows:

=	Assignment operator
++	Increment, i.e. $x++$ is equivalent to $x = x + 1$; when used in an array index, evaluates to the value of the variable prior to the increment operation.
--	Decrement, i.e. $x--$ is equivalent to $x = x - 1$; when used in an array index, evaluates to the value of the variable prior to the decrement operation.
+=	Increment by amount specified, i.e. $x += 3$ is equivalent to $x = x + 3$, and $x += (-3)$ is equivalent to $x = x + (-3)$.
-=	Decrement by amount specified, i.e. $x -= 3$ is equivalent to $x = x - 3$, and $x -= (-3)$ is equivalent to $x = x - (-3)$.

4.10 Range notation

The following notation is used to specify a range of values:

$x = y..z$	x takes on integer values starting from y to z , inclusive, with x , y , and z being integer numbers and z being greater than y .
<code>array[x,y]</code>	a sub-array containing the elements of array comprised between position x and y included. If x is greater than y , the resulting sub-array is empty.

4.11 Mathematical functions

The following mathematical functions are defined:

$\text{Ceil}(x)$	the smallest integer greater than or equal to x
$\text{Floor}(x)$	the largest integer less than or equal to x
$\text{Log2}(x)$	the base-2 logarithm of x

$$\text{Min}(x, y) = \begin{cases} x & ; x \leq y \\ y & ; x > y \end{cases}$$

$$\text{Max}(x, y) = \begin{cases} x & ; x \geq y \\ y & ; x < y \end{cases}$$

4.12 Array functions

`Size( arrayName[] )` returns the number of elements contained in the array or tensor named `arrayName`. If `arrayName[]` is a tensor this corresponds to the product of all dimensions of the tensor.

`Prod( arrayName[] )` returns the product of all elements of array `arrayName[]`.

`TensorReshape( arrayName[], tensorDimension[] )` returns the reshaped tensor `array_name[]` with the specified `tensorDimension[]`, without changing its data.

`IndexToXY(w, h, i, bs)` returns an array with two elements. The first element is an x coordinate and the second element is a y coordinate pointing into a 2D array of width w and height h . x and y point to the position that corresponds to scan index i when the block is scanned in blocks of size bs times bs . x and y are derived as follows:

A variable `fullRowOfBlocks` is set to $w * bs$

A variable `blockY` is set to $i / \text{fullRowOfBlocks}$

A variable `iOff` is set to $i \% \text{fullRowOfBlocks}$

A variable `currBlockH` is set to $\text{Min}(bs, h - \text{blockY} * bs)$

A variable `fullBlocks` is set to $bs * \text{currBlockH}$

A variable `blockX` is set to $iOff / \text{fullBlocks}$

A variable `blockOff` is set to $iOff \% \text{fullBlocks}$

A variable `currBlockW` is set to $\text{Min}(bs, w - \text{blockX} * bs)$

A variable `posX` is set to $\text{blockOff} \% \text{currBlockW}$

A variable `posY` is set to $\text{blockOff} / \text{currBlockW}$

The variable x is set to $\text{blockX} * bs + \text{posX}$

The variable y is set to $\text{blockY} * bs + \text{posY}$

`TensorIndex( tensorDimensions[], i, scan )` returns an array with the same number of dimensions as `tensorDimensions[]` where the elements of the array are set to integer values so that the array can be used as an index pointing to an element of a tensor with dimensions `tensorDimensions[]` as follows:

If variable `scan` is equal to 0:

The returned array points to the i -th element in row-major scan order of a tensor with dimensions `tensorDimensions[]`.

If variable `scan` is greater than 0:

A variable `bs` is set to $4 \ll \text{scan_order}$.

A variable `h` is set to `tensorDimensions[0]`.

A variable `w` is set to $\text{Prod}(\text{tensorDimensions}) / h$.

Two variables x and y are set to the first and second element of the array that is returned by calling `IndexToXY(w, h, i, bs)`, respectively.

The returned array is `TensorIndex(tensorDimensions, y * w + x, 0)`.

NOTE Variable *scan* usually corresponds to syntax element *scan_order*.

`GetEntryPointIdx( tensorDimensions[], i, scan )` returns -1 if index *i* doesn't point to the first position of an entry point. If index *i* points to the first position of an entry point, it returns the entry point index within the tensor. To determine the positions and indexes of entry points, the following applies:

A variable *w* is set to `Prod(tensorDimensions) / tensorDimensions[0]`.

A variable *epIdx* is set to $i / (w * (4 \ll scan)) - 1$.

If $i > 0$ and $i \% (w * (4 \ll scan))$ is equal to 0, index *i* points to the first position of an entry point and the entry point index is equal to *epIdx*.

Otherwise, index *i* doesn't point to the first position of an entry point.

`ShiftArrayIndex( inputArray[], shiftIndexPosition )` returns an array *outputArray[]* which is a copy of *inputArray[]* but with the element at position 0 of the *inputArray* shifted to *shiftIndexPosition* as follows:

A variable *outputArray[]* is initialized with a copy of *inputArray[]*.

If *shiftIndexPosition* is greater than 0:

The first element of *outputArray[]* is erased from *outputArray[]*.

The first element of *inputArray[]* is inserted into *outputArray[]* before the element with position *shiftIndexPosition* and after element with position *shiftIndexPosition*-1.

`DimensionShift( inputTensor[], tensorDimensions[], firstDimensionShift )` returns a tensor *reorderedTensor[]* which is a copy of *inputTensor[]* with the same number of dimensions, but where the dimensions are rearranged such that the first dimension of *inputTensor[]* specified by *tensorDimensions[]* is shifted to position *firstDimensionsShift* as follows:

A variable *reorderedTensor* is initialized with dimensions equal to `ShiftArrayIndex( tensorDimensions, firstDimensionShift )`.

The elements of variable *reorderedTensor* are set as follows:

```
for( i = 0; i < Prod( tensorDimensions ); i++ ){
    idxA = TensorIndex( tensorDimensions, i, 0 )
    idxB = ShiftArrayIndex( idxA, firstDimensionShift )
    reorderedTensor[idxB] = inputTensor[idxA]
}
```

`AxisSwap( inputTensor[], tensorDimensions[], numberOfDimensions, axis0, axis1 )` returns a tensor which is derived from *inputTensor* (with dimensions *tensorDimensions* and number of dimensions as *numberOfDimensions*) and where values in the axis indexes *axis0* and *axis1* of the *inputTensor* are swapped.

`TensorSplit( inputTensor[], splitIndices, splitAxis )` returns an array of tensors *subTensors* that is derived by splitting tensor *inputTensor* into $N = \text{Size}(\text{splitIndices}) + 1$ tensors using the provided array of indices *splitIndices* along the provided axis *splitAxis* as follows:

An array *inputDims* is set to the dimensions of tensor *inputTensor*.

An element with value 0 is inserted into *splitIndices* before the first element and an element with value *inputDims[splitAxis]* is inserted into *splitIndices* after the last element.

Tensor *subTensors[x]* (with *x* being an integer from 0 to *N*) is derived as follows:

An array *subTensorDims* is set to *inputDims*.

Element *subTensorDims*[*splitAxis*] is replaced with value *splitIndices*[*x* + 1] – *splitIndices*[*x*].

The elements of *subTensors*[*x*] are set as follows:

```
for( i = 0; i < Prod( subTensorDims ); i++ ) {
    subIdx = TensorIndex( subTensorDims, i, 0 )
    inputIdx = TensorIndex( inputDims, i, 0 )
    inputIdx[splitAxis] += splitIndices[x]
    subTensors[subIdx] = inputTensor[inputIdx]
```

4.13 Order of operation precedence

When the order of precedence in an expression is not indicated explicitly by use of parentheses, the following rules apply:

- Operations of a higher precedence are evaluated before any operation of a lower precedence.
- Operations of the same precedence are evaluated sequentially from left to right.

[Table 1](#) specifies the precedence of operations from highest to lowest; a higher position in the table indicates a higher precedence.

NOTE For those operators that are also used in the C programming language, the order of precedence used in this document is the same as used in the C programming language.

Table 1 — Operation precedence from highest (at top of table) to lowest (at bottom of table)

operations (with operands <i>x</i> , <i>y</i> , and <i>z</i>)
" <i>x</i> ++", " <i>x</i> --"
"! <i>x</i> ", " <i>-x</i> " (as a unary prefix operator)
" <i>x</i> ^{<i>y</i>} "
" <i>x</i> * <i>y</i> ", " <i>x</i> / <i>y</i> ", " <i>x</i> ÷ <i>y</i> ", " $\frac{x}{y}$ ", " <i>x</i> % <i>y</i> ", " $\prod_{i=x}^y f(i)$ ", " <i>x</i> ◦ <i>y</i> "
" <i>x</i> + <i>y</i> ", " <i>x</i> – <i>y</i> " (as a two-argument operator), " $\sum_{i=x}^y f(i)$ "
" <i>x</i> << <i>y</i> ", " <i>x</i> >> <i>y</i> "
" <i>x</i> < <i>y</i> ", " <i>x</i> ≤ <i>y</i> ", " <i>x</i> > <i>y</i> ", " <i>x</i> ≥ <i>y</i> "
" <i>x</i> == <i>y</i> ", " <i>x</i> != <i>y</i> "
" <i>x</i> & <i>y</i> "
" <i>x</i> <i>y</i> "
" <i>x</i> && <i>y</i> "
" <i>x</i> <i>y</i> "
" <i>x</i> ? <i>y</i> : <i>z</i> "
" <i>x</i> . <i>y</i> "
" <i>x</i> = <i>y</i> ", " <i>x</i> += <i>y</i> ", " <i>x</i> -= <i>y</i> "

4.14 Variables, syntax elements and tables

Syntax elements in the bitstream are represented in **bold** type. Each syntax element is described by its name (all lower-case letters with underscore characters), and one data type for its method of coded representation. The decoding process behaves according to the value of the syntax element and to the values of previously decoded syntax elements. When a value of a syntax element is used in the syntax tables or the text, it appears in regular (i.e. not bold) type.

In some cases the syntax tables may use the values of other variables derived from syntax elements values. Such variables appear in the syntax tables, or text, named by a mixture of lower-case and upper-case letter and without any underscore characters (camel case notation). Variables starting with an upper-case letter are derived for the decoding of the current syntax structure and all depending syntax structures. Variables starting with an upper-case letter may be used in the decoding process for later syntax structures without mentioning the originating syntax structure of the variable. Variables starting with a lower-case letter are only used within the (sub)clause in which they are derived.

In some cases, "mnemonic" names for syntax element values or variable values are used interchangeably with their numerical values. Sometimes "mnemonic" names are used without any associated numerical values. The association of values and names is specified in the text. The names are constructed from one or more groups of letters separated by an underscore character. Each group starts with an upper-case letter and may contain more upper-case letters.

NOTE The syntax is described in a manner that closely follows the C-language syntactic constructs.

Functions that specify properties of the current position in the bitstream are referred to as syntax functions. These functions are specified in [subclause 6.3](#) and assume the existence of a bitstream pointer with an indication of the position of the next bit to be read by the decoding process from the bitstream. Syntax functions are described by their names, which are constructed as syntax element names and end with left and right round parentheses including zero or more variable names (for definition) or values (for usage), separated by commas (if more than one variable).

Functions that are not syntax functions (including mathematical functions specified in [subclause 4.11](#) and array functions specified in [subclause 4.12](#)) are described by their names, which start with an upper-case letter, contain a mixture of lower and upper-case letters without any underscore character, and end with left and right parentheses including zero or more variable names (for definition) or values (for usage) separated by commas (if more than one variable).

A one-dimensional array is referred to as a list. A two-dimensional array is referred to as a matrix. Arrays can either be syntax elements or variables. Subscripts or square parentheses are used for the indexing of arrays. In reference to a visual depiction of a matrix, the first subscript is used as a row (vertical) index and the second subscript is used as a column (horizontal) index. The indexing order is reversed when using square parentheses rather than subscripts for indexing. Thus, an element of a matrix s at horizontal position x and vertical position y may be denoted either as $s[x][y]$ or as s_{yx} . A single column of a matrix may be referred to as a list and denoted by omission of the row index. Thus, the column of a matrix s at horizontal position x may be referred to as the list $s[x]$.

A multi-dimensional array is a variable with a number of dimensions. An element of the multi-dimensional array is either indexed by specifying all required indexes like e.g. `variable[x][y][z]` or by a single index variable that itself is a one-dimensional array specifying the indexes. For example `variable[i]` with i being a one-dimensional array with elements $[x, y, z]$. Multi-dimensional arrays are, for example, used to specify tensors.

A specification of values of the entries in rows and columns of an array may be denoted by $\{ \{ \dots \} \{ \dots \} \}$, where each inner pair of brackets specifies the values of the elements within a row in increasing column order and the rows are ordered in increasing row order. Thus, setting a matrix s equal to $\{ \{ 1 \ 6 \} \{ 4 \ 9 \} \}$ specifies that $s[0][0]$ is set equal to 1, $s[1][0]$ is set equal to 6, $s[0][1]$ is set equal to 4, and $s[1][1]$ is set equal to 9.

Binary notation is indicated by enclosing the string of bit values by single quote marks. For example, '01000001' represents an eight-bit string having only its second and its last bits (counted from the most to the least significant bit) equal to 1.

Hexadecimal notation, indicated by prefixing the hexadecimal number by "0x", may be used instead of binary notation when the number of bits is an integer multiple of 4. For example, 0x41 represents an eight-bit string having only its second and its last bits (counted from the most to the least significant bit) equal to 1.

Numerical values not enclosed in single quotes and not prefixed by "0x" are decimal values.

A value equal to 0 represents a FALSE condition in a test statement. The value TRUE is represented by any value different from zero.

5 Overview

5.1 General

This clause provides an overview of the compression tools defined in this document and describes how they can be combined to encoding pipelines.

5.2 Compression tools

This document contains the following groups of compression tools.

Parameter reduction methods process a model to obtain a compact representation. Examples of such methods include, *parameter sparsification*, *parameter pruning*, *weight unification*, and *decomposition methods*.

Sparsification processes parameters or groups of parameters to produce a sparse representation of the model (e.g. by replacing some weight values with zeros). The sparsification can generate additional metadata (e.g. masks). The sparsification can be structured or unstructured. This document includes methods for unstructured sparsification with compressibility loss (information about encoding provided in [subclause G.1.2](#)), structured sparsification using micro-structured sparsification (information about encoding provided in [subclause G.1.3](#)), unstructured statistics-adaptive sparsification (information about encoding provided in [subclause G.1.5](#)), and structured sparsification (information about encoding provided in [subclause G.1.6](#)).

Unification processes the parameters to produce group of similar parameters. Unification does not eliminate or constrain the weights to be zero, but it lowers the entropy of model parameters by making them similar to each other. This document includes a method for parameter unification (information about encoding provided in [subclause G.1.7](#)).

Pruning reduces the number of parameters by eliminating parameters or groups of parameters. The procedure results in a dense representation which has less parameters in comparison to the original model, e.g. by removing some redundant convolution filters from the layers. This document includes a a method for combined pruning and sparsification (information about encoding provided in [subclause G.1.4](#))

Decomposition performs a matrix decomposition operation to change the structure of the weights of a model. This document includes a method for low rank/low displacement rank for convolutional and fully connected layers (information about encoding provided in [subclause G.1.8](#)).

Along with the reduction methods mentioned above, this document includes decomposition methods that are introduced and tested as part of a parameter quantization technique. Examples of such methods are batchnorm folding ([subclause 8.2.1](#) and information about encoding provided in [subclause G.1.9](#)) and local scaling adaptation (information about encoding provided in [subclause G.1.10](#)).

The parameter reduction methods can be combined or applied in sequence to produce a compact model.

Parameter quantization methods reduce the precision of the representation of parameters. If supported by the inference engine, the quantized representation can be used for more efficient inference. This document includes methods for uniform quantization ([subclause 9.2.1](#)), codebook-based quantization ([subclause 9.2.2](#)), dependent scalar quantization ([subclause 9.2.3](#)), iterative QP optimization (information about encoding provided in [subclause G.2.2](#)) and stochastic binary-ternary quantization (information about encoding provided in [subclause G.2.3](#)).

Predictive residual encoding (PRE, [subclause 9.2.4](#)) enables to code residuals based on a previously decoded update of the model rather than the complete weight update.

Entropy coding methods encode the results of parameter quantization methods. This document includes DeepCABAC ([subclause 10.1.1](#)) as entropy encoding method. Supported extensions for DeepCABAC include Row Skipping and Temporal Context Modeling.

Row Skipping reduces the number of bins to be decoded and also the bitstream size by skipping decoding of matrix rows that are entirely zero. The method is described in [subclause 10.1.1.2](#). *Temporal Context*

Modeling uses information from previously decoded incremental updates to improve the context modeling of DeepCABAC and thus increases the coding efficiency. The method is described in [subclause 10.1.1.4](#).

5.3 Creating encoding pipelines

The compression tools in this document can be combined to form different encoding pipelines. Some of the tools are alternatives for addressing neural network models with different types of characteristics, while other tools are designed to work in sequence.

[Figure 1](#) shows an overview of encoding pipelines that can be assembled using the compression tools in this document. From the group of parameter transformation tools, multiple tools can be applied in sequence. Parameter quantization can be applied to source models as well as to the outputs of transformation with parameter reduction methods. Entropy coding is usually applied to the output of quantization. Raw outputs of earlier steps without applying entropy coding can be serialized if needed.

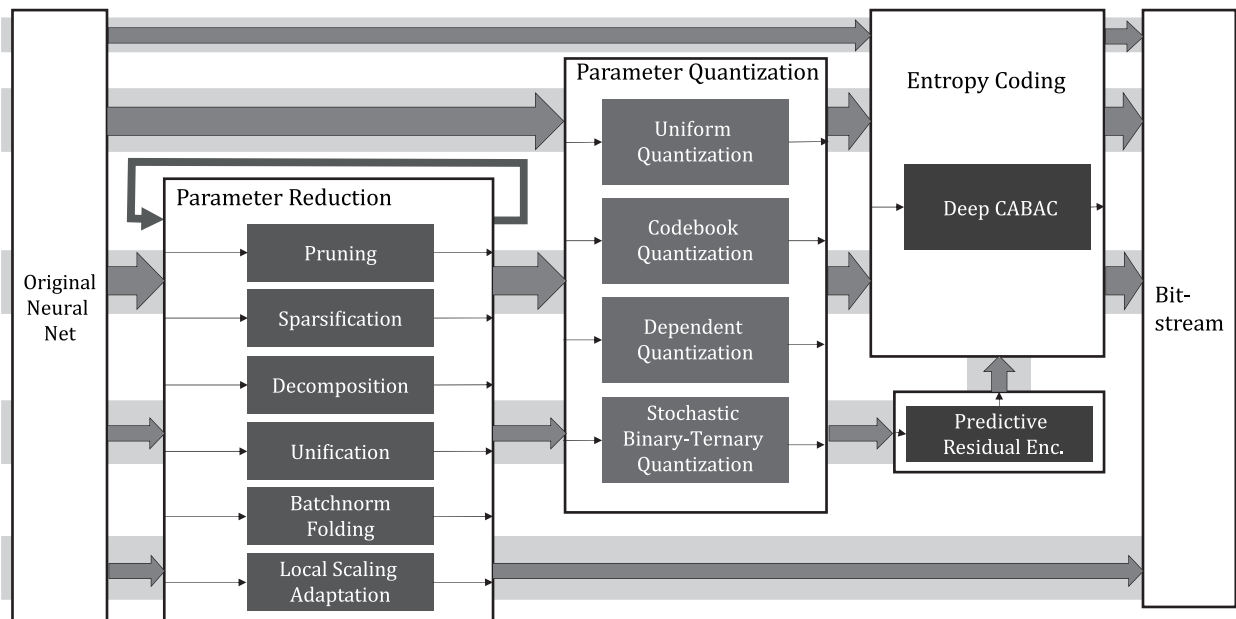


Figure 1 — NNR encoding pipelines

The following encoding pipelines are considered typical examples of using this document:

1. Dependent scalar quantization ([subclause 9.2.3](#)) – DeepCABAC ([subclause 10.1.1](#))
2. Sparsification (information about encoding provided in [subclause G.1.2](#)) – Dependent scalar quantization ([subclause 9.2.3](#)) – DeepCABAC ([subclause 10.1.1](#))
3. Low-rank decomposition (information about encoding provided in [subclause G.1.8](#)) – Dependent scalar quantization ([subclause 9.2.3](#)) – DeepCABAC ([subclause 10.1.1](#))
4. Codebook-based quantization ([subclause 9.2.2](#)) – DeepCABAC ([subclause 10.1.1](#))
5. Unification (information about encoding provided in [subclause G.1.7](#)) – DeepCABAC ([subclause 10.1.1](#))
6. Stochastic binary-ternary quantization (information about encoding provided in [subclause G.2.3](#)) – Predictive residual encoding ([subclause 9.2.4](#)) – DeepCABAC ([subclause 10.1.1](#))

This list is non-exhaustive.

The following coding tools are only applicable to updates of neural network parameters:

- Predictive residual encoding ([subclause 9.2.4](#));

— Temporal Context Modeling ([subclause 10.1.1.4](#)).

6 Syntax and semantics

6.1 Specification of syntax and semantics

6.1.1 Method of specifying syntax in tabular form

The syntax tables specify a superset of the syntax of all allowed bitstreams. Additional constraints on the syntax may be specified, either directly or indirectly, in other clauses.

[Table 2](#) lists examples of the syntax specification format. When **syntax_element** appears, it specifies that a syntax element is parsed from the bitstream and the bitstream pointer is advanced to the next position beyond the syntax element in the bitstream parsing process.

Table 2 — Examples of the syntax specification format

Syntax	Type/Clause
/* A statement can be a syntax element with an associated data type or can be an expression used to specify conditions for the existence, type and quantity of syntax elements, as in the following two examples */	
syntax_element	st(v)
conditioning statement	
/*A group of statements enclosed in curly brackets is a compound statement and is treated functionally as a single statement. */	
{	
statement	
statement	
...	
}	
/* A "while" structure specifies a test of whether a condition is true, and if true, specifies evaluation of a statement (or compound statement) repeatedly until the condition is no longer true */	
while(condition)	
statement	
/* A "do ... while" structure specifies evaluation of a statement once, followed by a test of whether a condition is true, and if true, specifies repeated evaluation of the statement until the condition is no longer true */	
do	
statement	
while(condition)	
/* An "if ... else" structure specifies a test of whether a condition is true and, if the condition is true, specifies evaluation of a primary statement, otherwise, specifies evaluation of an alternative statement. The "else" part of the structure and the associated alternative statement is omitted if no alternative statement evaluation is needed */	
if(condition)	
primary statement	
else	
alternative statement	

Table 2 (continued)

Syntax	Type/Clause
/* A "for" structure specifies evaluation of an initial statement, followed by a test of a condition, and if the condition is true, specifies repeated evaluation of a primary statement followed by a subsequent statement until the condition is no longer true. */	
for(initial statement; condition; subsequent statement)	
primary statement	

6.1.2 Bit ordering

For bit-oriented delivery, the bit order of syntax fields in the syntax tables is specified to start with the MSB and proceed to the LSB.

6.1.3 Specification of syntax functions and data types

The functions presented here are used in the syntactical description. These functions are expressed in terms of the value of a bitstream pointer that indicates the position of the next bit to be read by the decoding process from the bitstream.

byte_aligned() is specified as follows:

- If the current position in the bitstream is on a byte boundary, i.e. the next bit in the bitstream is the first bit in a byte, the return value of byte_aligned() is equal to TRUE.
- Otherwise, the return value of byte_aligned() is equal to FALSE.

read_bits(n) reads the next n bits from the bitstream and advances the bitstream pointer by n bit positions. When n is equal to 0, read_bits(n) is specified to return a value equal to 0 and to not advance the bitstream pointer.

get_bit_pointer() returns the position of the bitstream pointer relative to the beginning of the current NNR unit as unsigned integer value. get_bit_pointer() >> 3 points to the current byte of the bitstream pointer. get_bit_pointer() & 7 points to the current bit in the current byte of the bitstream pointer where a value of 0 indicates the most significant bit.

set_bit_pointer(pos) sets the position of the bitstream pointer such that get_bit_pointer() equals pos.

The following data types specify the parsing process of each syntax element:

- ae(v): context-adaptive arithmetic entropy-coded syntax element. The parsing process for this data type is specified in [subclause 10.3.4.3.2](#).
- at(v) : arithmetic entropy-coded termination syntax. The parsing process for this data type is specified in [subclause 10.3.4.3.5](#).
- iae(n): signed integer using n arithmetic entropy-coded bits using the bypass mode of DeepCABAC as specified in [subclause 10.3.4.3.4](#). The read bypass bins are interpreted as a two's complement integer representation with most significant bit written first.
- uae(n): unsigned integer using n arithmetic entropy-coded bits using the bypass mode of DeepCABAC as specified in [subclause 10.3.4.3.4](#). The read bypass bins are interpreted as a binary representation of an unsigned integer with most significant bit written first. When n=0, uae(n) does not decode any bins and returns 0.
- f(n): fixed-pattern bit string using n bits written (from left to right) with the left bit first. The parsing process for this data type is specified by the return value of the function read_bits(n).
- i(n): signed integer using n bits. When n is "v" in the syntax table, the number of bits varies in a manner dependent on the value of other syntax elements. The parsing process for this data type is specified by

the return value of the function `read_bits( n )` interpreted as a two's complement integer representation with most significant bit written first.

- `u(n)`: unsigned integer using n bits. When n is “v” in the syntax table, the number of bits varies in a manner dependent on the value of other syntax elements. The parsing process for this data type is specified by the return value of the function `read_bits( n )` interpreted as a binary representation of an unsigned integer with most significant bit written first.
- `ue(k)`: unsigned integer k -th order Exp-Golomb-coded syntax element. The parsing process for this descriptor is according to the following pseudo-code with x as result:

```
x = 0
bit = 1
while( bit ) {
    bit = 1 - u( 1 )
    x += bit << k
    k += 1
}
k -= 1
if( k > 0 )
    x += u( k )
```

- `ie(k)`: signed integer k -th order Exp-Golomb-coded syntax element. The parsing process for this descriptor is according to the following pseudo-code with x as result:

```
val = ue( k )
if( (val & 1) != 0 )
    x = ((val+1)>>1)
else
    x = - (val>>1)
```

- `flt(n)`: Floating point value using n bits where n may be 32, 64, or 128 in little-endian byte order as specified in ISO/IEC 60559 as `binary32`, `binary64`, or `binary128`, respectively.
- `st(v)`: null-terminated string, which shall be encoded as UTF-8 characters in accordance with ISO/IEC 10646. The parsing process is specified as follows: `st(v)` begins at a byte-aligned position in the bitstream and reads and returns a series of bytes from the bitstream, beginning at the current position and continuing up to but not including the next byte-aligned byte that is equal to `0x00`, and advances the bitstream pointer by $(\text{stringLength} + 1) * 8$ bit positions, where `stringLength` is equal to the number of bytes returned.

NOTE The `st(v)` and `flt(n)` syntax descriptors are only used in this document when the current position in the bitstream is a byte-aligned position.

- `bs(v)`: Byte-sequence specifies a sequence of bytes of variable length, starting at byte-aligned position. The length of the sequence is determined from the size of the NNR unit containing the byte sequence.

6.1.4 Semantics

Semantics associated with the syntax structures and with the syntax elements within each structure are specified in a subclause following the subclause containing the syntax structures.

The following definitions apply to the semantics specification.

unspecified is used to specify some values of a particular *syntax element* to indicate that the values have no specified meaning in this document and will not have a specified meaning in the future as an integral part of future versions of this document.

reserved is used to specify that some values of a particular *syntax element* are for future use by ISO/IEC and shall not be used in *bitstreams* conforming to this version of this document, but may be used in bitstreams conforming to future extensions of this document by ISO/IEC.

nnr_reserved_zero_0bit shall be an element of length 0. Decoders shall ignore the value of **nnr_reserved_zero_0bit**.

nnr_reserved_zero_1bit, when present, shall be equal to 0 in bitstreams conforming to this version of this document. Other values for **nnr_reserved_zero_1bit** are reserved for future use by ISO/IEC. Decoders shall ignore the value of **nnr_reserved_zero_1bit**.

nnr_reserved_zero_2bits, when present, shall be equal to 0 in bitstreams conforming to this version of this document. Other values for **nnr_reserved_zero_2bits** are reserved for future use by ISO/IEC. Decoders shall ignore the value of **nnr_reserved_zero_2bits**.

nnr_reserved_zero_3bits, when present, shall be equal to 0 in bitstreams conforming to this version of this document. Other values for **nnr_reserved_zero_3bits** are reserved for future use by ISO/IEC. Decoders shall ignore the value of **nnr_reserved_zero_3bits**.

nnr_reserved_zero_5bits, when present, shall be equal to 0 in bitstreams conforming to this version of this document. Other values for **nnr_reserved_zero_5bits** are reserved for future use by ISO/IEC. Decoders shall ignore the value of **nnr_reserved_zero_5bits**.

nnr_reserved_zero_7bits, when present, shall be equal to 0 in bitstreams conforming to this version of this document. Other values for **nnr_reserved_zero_7bits** are reserved for future use by ISO/IEC. Decoders shall ignore the value of **nnr_reserved_zero_7bits**.

6.2 General bitstream syntax elements

6.2.1 NNR unit

NNR unit is the data structure for carrying neural network data and related metadata which is compressed or represented using this document.

NNR units carry compressed or uncompressed information about neural network metadata, topology information, complete or partial layer data, filters, kernels, biases, quantized weights, tensors or alike.

An NNR unit consists of the following data elements (shown in [Figure 2](#)):

- **NNR unit size**: This data element signals the total byte size of the NNR unit, including the NNR unit size.
- **NNR unit header**: This data element contains information about the NNR unit type and related metadata.
- **NNR unit payload**: This data element contains compressed or uncompressed data related to the neural network.

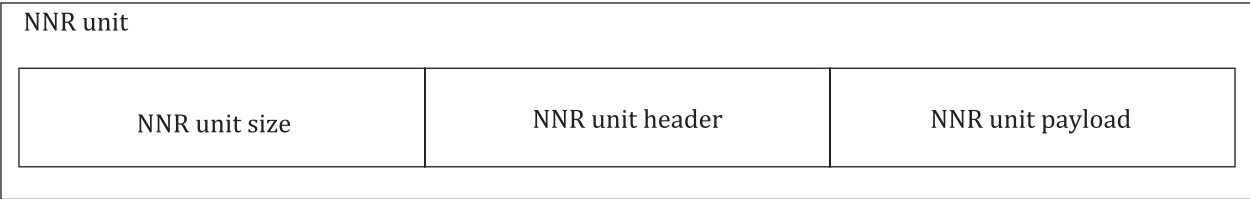


Figure 2 — NNR Unit data structure

6.2.2 Aggregate NNR unit

An aggregate NNR unit is an NNR unit which carries multiple NNR units in its payload. Aggregate NNR units provide a grouping mechanism for several NNR units which are related to each other and benefit from aggregation under a single NNR unit (shown in [Figure 3](#)).

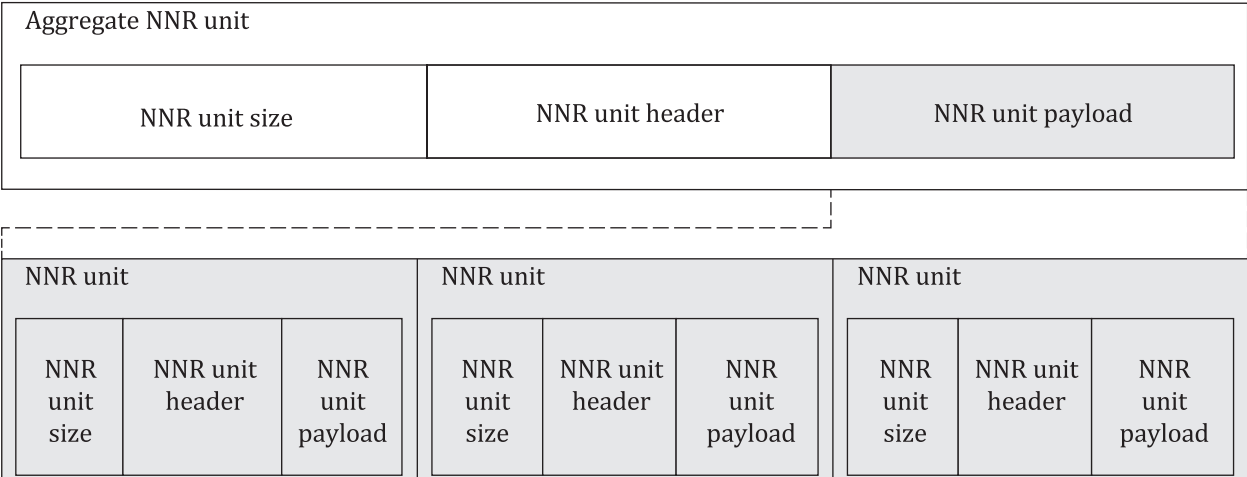


Figure 3 — Aggregate NNR unit data structure

6.2.3 Composition of NNR bitstream

NNR bitstream is composed of a sequence of NNR units (shown in [Figure 4](#)).

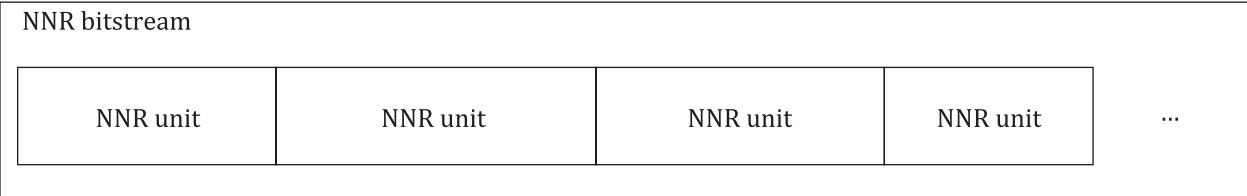


Figure 4 — NNR bitstream data structure

In an NNR bitstream; the following constraints apply unless otherwise stated in this document or defined by NNR profiles:

(NNR_STR, NNR_MPS, NNR_NDU, NNR_LPS, NNR_TPL and NNR_QNT are NNR unit types as specified in [Table 3](#) of [subclause 6.4.3](#))

- An NNR bitstream shall start with an NNR start unit (NNR_STR) ([subclause 6.4.3](#))
- There shall be a single NNR model parameter set (NNR_MPS) ([subclause 6.4.3](#)) in an NNR bitstream which shall precede any NNR_NDU ([subclause 6.4.3](#)) in the NNR bitstream
- NNR layer parameter sets (NNR_LPS) shall be active until the next NNR layer parameter set in the NNR bitstream or until the boundary of an Aggregate NNR unit is reached.
- **topology_elem_id** and **topology_elem_id_index** ([subclause 6.4.3.7](#)) values shall be unique in the NNR bitstream.
- NNR_TPL or NNR_QNT units; if present in the NNR bitstream; shall precede any NNR_NDUs that reference their data structures (e.g. **topology_elem_id**).

6.3 NNR bitstream syntax

6.3.1 NNR unit syntax

nnr_unit() {	Descriptor
nnr_unit_size()	
nnr_unit_header()	
nnr_unit_payload()	
}	

6.3.2 NNR unit size syntax

nnr_unit_size() {	Descriptor
nnr_unit_size_flag	u(1)
nnr_unit_size	u(15 + nnr_unit_size_flag*16)
}	

6.3.3 NNR unit header syntax

6.3.3.1 General

nnr_unit_header() {	Descriptor
nnr_unit_type	u(6)
independently_decodable_flag	u(1)
partial_data_counter_present_flag	u(1)
if(partial_data_counter_present_flag)	
partial_data_counter	u(8)
if(nnr_unit_type == NNR_MPS)	
nnr_model_parameter_set_unit_header()	
if(nnr_unit_type == NNR_LPS)	
nnr_layer_parameter_set_unit_header()	
if(nnr_unit_type == NNR_TPL)	
nnr_topology_unit_header()	
if(nnr_unit_type == NNR_QNT)	
nnr_quantization_unit_header()	
if(nnr_unit_type == NNR_NDU)	
nnr_compressed_data_unit_header()	
if(nnr_unit_type == NNR_STR)	
nnr_start_unit_header()	
if(nnr_unit_type == NNR_AGG)	
nnr_aggregate_unit_header()	
}	

6.3.3.2 NNR start unit header syntax

nnr_start_unit_header() {	Descriptor
general_profile_idc	u(8)
}	

6.3.3.3 NNR model parameter set unit header syntax

nnr_model_parameter_set_unit_header() {	Descriptor
nnr_reserved_zero_0bit	u(0)
}	

6.3.3.4 NNR layer parameter set unit header syntax

nnr_layer_parameter_set_unit_header() {	Descriptor
lps_self_contained_flag	u(1)
nnr_reserved_zero_7_bits	u(7)
}	

6.3.3.5 NNR topology unit header syntax

nnr_topology_unit_header() {	Descriptor
topology_storage_format	u(8)
topology_compression_format	u(8)
}	

6.3.3.6 NNR quantization unit header syntax

nnr_quantization_unit_header() {	Descriptor
quantization_storage_format	u(8)
quantization_compression_format	u(8)
}	

6.3.3.7 NNR compressed data unit header syntax

nnr_compressed_data_unit_header() {	Descriptor
nnr_compressed_data_unit_payload_type	u(5)
nnr_multiple_topology_elements_present_flag	u(1)
nnr_decompressed_data_format_present_flag	u(1)
input_parameters_present_flag	u(1)
if(nnr_multiple_topology_elements_present_flag == 1)	
topology_elements_ids_list(mps_topology_indexed_reference_flag)	
else {	
if(!mps_topology_indexed_reference_flag)	
topology_elem_id	st(v)
else	
topology_elem_id_index	ue(7)
}	

if(general_profile_idc == 1) {	
node_id_present_flag	u(1)
if(node_id_present_flag) {	
device_id	ue(1)
parameter_id	ue(5)
put_node_depth	ue(4)
}	
if(mps_parent_signalling_enabled_flag == 1) {	
parent_node_id_present_flag	u(1)
if(parent_node_id_present_flag) {	
parent_node_id_type	u(2)
temporal_context_modeling_flag	u(1)
if(parent_node_id_type == ICNN_NDU_ID) {	
parent_device_id	ue(1)
if(!node_id_present_flag) {	
parameter_id	ue(5)
put_node_depth	ue(4)
}	
} else if(parent_node_id_type == ICNN_NDU_PL_SHA256)	
parent_node_payload_sha256	u(256)
else if(parent_node_id_type == ICNN_NDU_PL_SHA512)	
parent_node_payload_sha512	u(512)
}	
}	
}	
if(nnr_compressed_data_unit_payload_type == NNR_PT_FLOAT nnr_compressed_data_unit_payload_type == NNR_PT_BLOCK) {	
codebook_present_flag	u(1)
if(codebook_present_flag)	
integer_codebook(CbZeroOffset, Codebook, CbSize)	
}	
if(nnr_compressed_data_unit_payload_type == NNR_PT_INT nnr_compressed_data_unit_payload_type == NNR_PT_FLOAT nnr_compressed_data_unit_payload_type == NNR_PT_BLOCK)	
dq_flag	u(1)
if(nnr_decompressed_data_format_present_flag == 1)	
nnr_decompressed_data_format	u(7)
if(input_parameters_present_flag == 1) {	
tensor_dimensions_flag	u(1)
cabac_unary_length_flag	u(1)
compressed_parameter_types	u(4)
if((compressed_parameter_types & NNR_CPT_DC) != 0) {	
decomposition_rank	ue(3)
g_number_of_rows	ue(3)
}	

if(tensor_dimensions_flag == 1)	
tensor_dimension_list()	
if (nnr_compressed_data_unit_payload_type != NNR_PT_BLOCK)	
if(nnr_multiple_topology_elements_present_flag == 1)	
topology_tensor_dimension_mapping()	
if(cabac_unary_length_flag == 1)	
cabac_unary_length_minus1	u(8)
}	
if(nnr_compressed_data_unit_payload_type == NNR_PT_BLOCK && (compressed_parameter_types & NNR_CPT_DC) != 0 && codebook_present_flag)	
integer_codebook(CbZeroOffsetDC, CodebookDC, CbSizeDC)	
if(count_tensor_dimensions > 1) {	
if(general_profile_idc == 1)	
first_tensor_dimension_shift	ue(1)
scan_order	u(4)
if(scan_order > 0) {	
for(j=0; j < NumBlockRowsMinus1; j++) {	
cabac_offset_list[j]	u(8)
if(dq_flag)	
dq_state_list[j]	u(3)
if(j == 0) {	
bit_offset_delta1	ue(11)
BitOffsetList[j] = bit_offset_delta1	
}	
else {	
bit_offset_delta2	ie(7)
BitOffsetList[j] = BitOffsetList[j-1] + bit_offset_delta2	
}	
}	
}	
}	
byte_alignment()	
}	

integer_codebook() is defined as follows:

integer_codebook[cbZeroOffset, integerCodebook, cbSize] {	Descriptor
codebook_egk	u(4)
codebook_size	ue(2)
cbSize = codebook_size	
codebook_centre_offset	ie(2)
cbZeroOffset = (codebook_size >> 1) + codebook_centre_offset	
codebook_zero_value	ie(7)
integerCodebook[cbZeroOffset] = codebook_zero_value	
previousValue = integerCodebook[cbZeroOffset]	

for(j = cbZeroOffset - 1; j >= 0; j--) {	
codebook_delta_left	ue(codebook_egk)
integerCodebook[j] = previousValue - codebook_delta_left - 1	
previousValue = integerCodebook[j]	
}	
previousValue = integerCodebook[cbZeroOffset]	
for(j = cbZeroOffset + 1; j < codebook_size; j++) {	
codebook_delta_right	ue(codebook_egk)
integerCodebook[j] = previousValue + codebook_delta_right + 1	
previousValue = integerCodebook[j]	
}	
}	

tensor_dimension_list() is defined as follows:

tensor_dimension_list() {	Descriptor
count_tensor_dimensions	ue(1)
for(j = 0; j < count_tensor_dimensions; j++)	
tensor_dimensions[j]	ue(7)
}	

topology_elements_ids_list(topologyIndexedFlag) is defined as follows:

topology_elements_ids_list(topologyIndexedFlag) {	Descriptor
count_topology_elements_minus2	ue(7)
if(topologyIndexedFlag == 0)	
byte_alignment()	
for(j = 0; j < count_topology_elements_minus2 + 2; j++) {	
if (topologyIndexedFlag == 0) {	
topology_elem_id_list[j]	st(v)
}	
else {	
topology_elem_id_index_list[j]	ue(7)
}	
}	
if (topologyIndexedFlag == 1)	
byte_alignment()	
}	

topology_tensor_dimension_mapping() is defined as follows:

topology_tensor_dimension_mapping () {	Descriptor
concatentation_axis_index	u(8)
for(j = 0; j < count_topology_elements_minus2 + 1 ; j++) {	
split_index[j]	ue(7)
}	
for(k = 0; k < count_topology_elements_minus2 + 2 ; k++) {	