INTERNATIONAL STANDARD

ISO/IEC 15938-17

First edition
2022-08

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

Reference number
ISO/IEC 15938-17:2022(E)

Contents

Page

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).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents) or the IEC list of patent declarations received (see https://patents.iec.ch).

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*.

A list of all parts in the ISO/IEC 15938 series can be found on the ISO website 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 (NNEF[1]), 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 (ONNX®)[2], 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 PyTorch®[3] 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 Tensorflow®[4] 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.

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

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.

be useful for the compression of neural networks used in other applications and applied to other types of data.

The International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) draw attention to the fact that it is claimed that compliance with this document may involve the use of patents.

ISO and IEC take no position concerning the evidence, validity and scope of this patent right.

The holder of this patent right has assured ISO and IEC that he/she is willing to negotiate licences under reasonable and non-discriminatory terms and conditions with applicants throughout the world. In this respect, the statement of the holder of this patent right is registered with ISO and IEC. Information may be obtained from the patent database available at www.iso.org/patents.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights other than those in the patent database. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

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.

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, 1996*

NNEF-v1.0.3, Neural Network Exchange Format, The Khronos NNEF Working Group, Version 1.0.3, 2020-06-12 (https://www.khronos.org/registry/NNEF/specs/1.0/nnef-1.0.3.pdf)

3 Terms and definitions

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

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

3.3
decomposition
transformation to express a tensor as product of two tensors

3.4
hyperparameter
parameter whose value is used to control the learning process

3.5
layer
collection of nodes operating together at a specific depth within a neural network

3.6
model parameter
coefficients of the neural network model such as weights and biases

3.7
NNR unit
data structure for carrying (compressed or uncompressed) neural network data and related metadata

3.8
pruning
reduction of parameters in (a part of) the neural network

3.9
sparsification
increase of the number of zero-valued entries of a tensor

3.10
tensor
multidimensional structure grouping related model parameters

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
LSB	Least significant bit
MPS	Model parameter set
MSB	Most significant bit
NN	Neural network
NNEF	Neural network exchange format
NNC	Neural network coding
NNR	Compressed neural network representation
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 of tensor in dimension j and in layer k
$c_j^{k'}$	Derived number of channels of tensor in dimension j and 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 a product of n_j^k
P	Stochastic transition matrix
p	Pruning ratio hyperparameter
p_{ij}	Elements of transition matrix P
q	Sparsification ratio hyperparameter
S	Importance of parameters for pruning
S_j^k	Superblock in layer k
s	Local scaling factors
s_j^k	Size of superblock in layer k
u	Unification ratio hyperparameter
W	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
v_j^k	Width dimension of tensor for layer k.
$w_{l,i}$	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
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
γ_c	Compressibility loss multiplier
γ	Batch normalization parameter
δ	Folded batch normalization parameter
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
π	Equilibrium probability of P
τ	Sparsification pruning 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
$\circ$	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.

÷	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
≥	Greater than or equal to
<	Less than
≤	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.
$x >> y$	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.
$x << y$	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.
array[x, y]	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:

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

$$\mathrm{Min}(x,y)=\begin{cases}x\,;\,x\le y\\ y\,;\,x>y\end{cases}$$

$$\mathrm{Max}(x,y)=\begin{cases}x\,;\,x\ge y\\ y\,;\,x<y\end{cases}$$

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* / *fullRowOfBlocks*

A variable *iOff* is set to *i* % *fullRowOfBlocks*

A variable *currBlockH* is set to Min(*bs*, *h* − *blockY* * *bs*)

A variable *fullBlocks* is set to *bs* * *currBlockH*

A variable *blockX* is set to *iOff* / *fullBlocks*

A variable *blockOff* is set to *iOff* % *fullBlocks*

A variable *currBlockW* is set to Min(*bs*, *w* − *blockX* * *bs*)

A variable *posX* is set to *blockOff* % *currBlockW*

A variable *posY* is set to *blockOff* / *currBlockW*

The variable *x* is set to *blockX* * *bs* + *posX*

The variable y is set to *blockY* * *bs* + *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 << *scan_order*.

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

A variable *w* is set to Prod(*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 << *scan*)) – 1.

If *i* > 0 and *i* % (*w* * (4 << *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.

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* = Size(*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 x, y, and z)
"x++", "x− −"
"!x", "−x" (as a unary prefix operator)
"x^y"
"$x * y$", "x / y", "$x \div y$", "$\frac{x}{y}$", "$x \% y$", "$\prod_{i=x}^{y} f(i)$", "$x \circ y$"
"$x + y$", "$x - y$" (as a two-argument operator), "$\sum_{i=x}^{y} f(i)$"
"$x << y$", "$x >> y$"
"$x < y$", "$x \le y$", "$x > y$", "$x \ge y$"
"$x == y$", "$x != y$"
"$x \& y$"
"$x \mid y$"
"$x \&\& y$"
"$x \mid\mid y$"
"$x ? y : z$"
"$x..y$"
"$x = y$", "$x += y$", "$x -= y$"

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 { {...} {...} }, 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.

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 group 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 (subclause 8.2.1), and structured sparsification using micro-structured sparsification (subclause 8.2.2).

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 weight unification (subclause 8.2.4).

Pruning reduces the number of parameters by eliminating parameters or group 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 (subclause 8.2.3)

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 (subclause 8.2.5).

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 8.2.6) and local scaling adaptation (subclause 8.2.7).

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 9.1.1), codebook-based quantization (subclause 9.1.2) and dependent scalar quantization (subclause 9.1.3).

Entropy coding methods encode the results of parameter quantization methods. This document includes DeepCABAC (subclause 10.1.1) as entropy encoding method.

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.

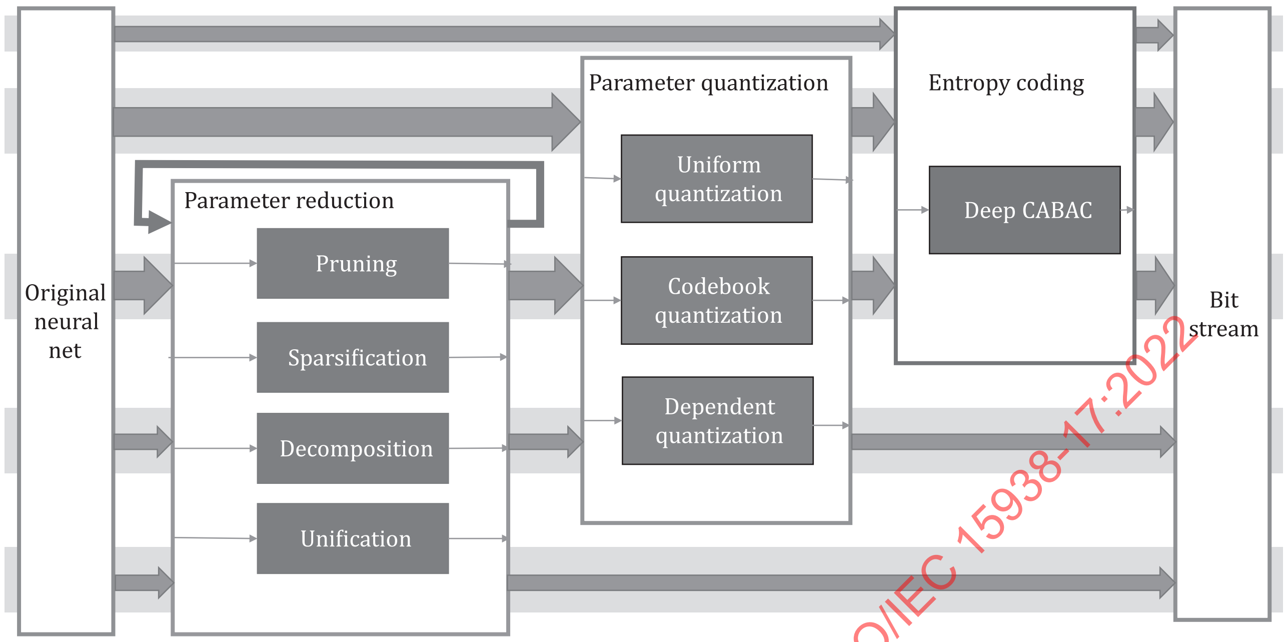

Figure 1 — NNR encoding pipelines.

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

1. Dependent scalar quantization (subclause 9.1.3) – DeepCABAC (subclause 10.1.1)
2. Sparsification (subclause 8.2.1) – Dependent scalar quantization (subclause 9.1.3) – DeepCABAC (subclause 10.1.1)
3. Low-rank decomposition (subclause 8.2.5) – Dependent scalar quantization (subclause 9.1.3)– DeepCABAC (subclause 10.1.1)
4. Codebook-based quantization (subclause 9.1.2) – DeepCABAC (subclause 10.1.1)
5. Unification (subclause 8.2.4) – DeepCABAC (subclause 10.1.1)

This list is non-exhaustive.

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 */	

Table 2 *(continued)*

Syntax	Type/Clause
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	
/* 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 (stringLength + 1) * 8 bit positions, where stringLength is equal to the number of bytes returned.

NOTE The st(v) syntax descriptors is 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.

more_data_in_nnr_unit( ) is specified as follows:

— If more data follow in the current nnr_unit, i.e. the decoded data up to now in the current nnr_unit is less than numBytesInNNRUnit, the return value of more_data_in_nnr_unit( ) is equal to TRUE.

— Otherwise, the return value of more_data_in_nnr_unit( ) is equal to FALSE.

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.

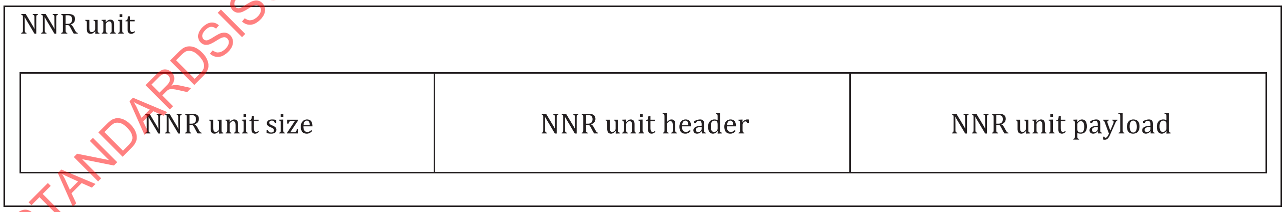

Figure 2 — NNR Unit data structure

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).

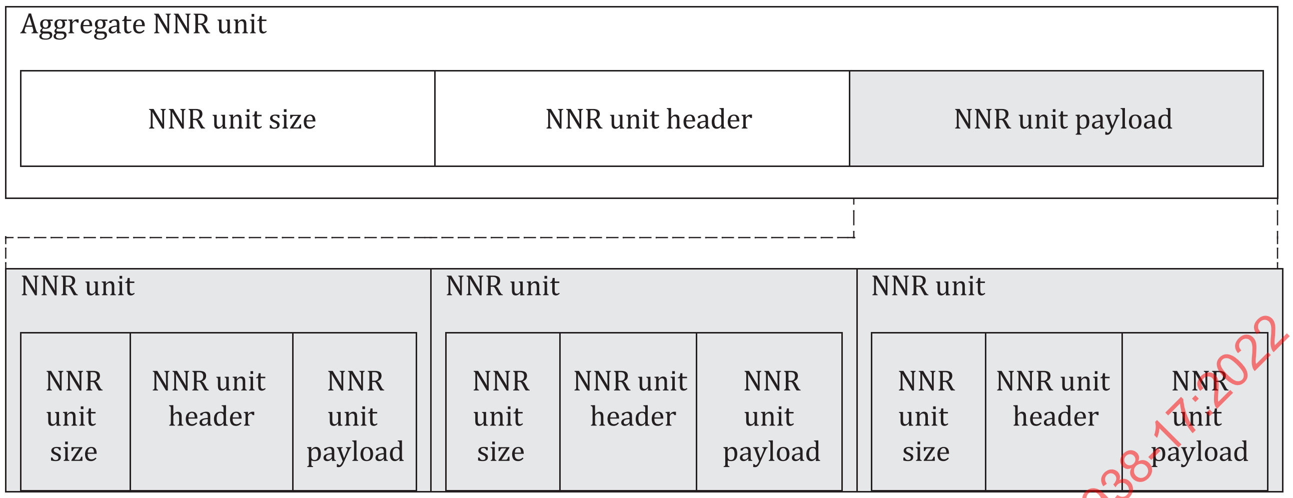

Figure 3 — Aggregate NNR unit data structure

6.2.3 Composition of NNR bitstream

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

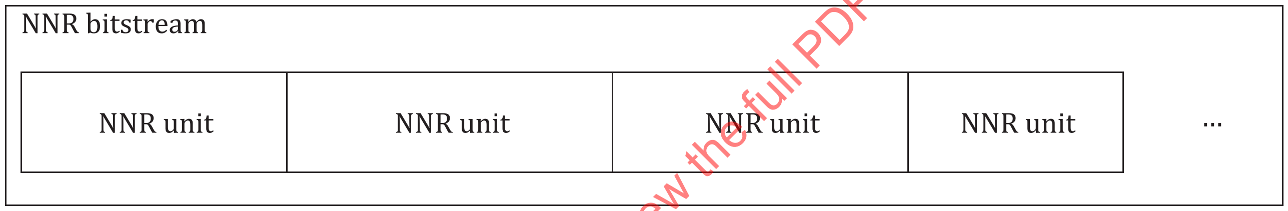

Figure 4 — NNR bitstream data structure

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(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)	
}	

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)	
if(count_tensor_dimensions > 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) {	**Descriptor**
codebook_egk	u(4)
codebook_size	ue(2)
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++) {	
number_of_shifts[k]	ue(1)
for(i = 0; i < number_of_shifts[k] ; i++) {	
shift_index[k][i]	ue(7)
shift_value[k][i]	ue(1)
}	
}	
}	

6.3.3.8 NNR aggregate unit header syntax

nnr_aggregate_unit_header( ) {	**Descriptor**
nnr_aggregate_unit_type	u(8)
entry_points_present_flag	u(1)
nnr_reserved_zero_7bits	u(7)
num_of_nnr_units_minus2	u(16)
if(entry_points_present_flag)	
for(i = 0; i < num_of_nnr_units_minus2 + 2; i++) {	
nnr_unit_type[i]	u(6)
nnr_unit_entry_point[i]	u(34)
}	
for(i = 0; i < num_of_nnr_units_minus2 + 2; i++) {	
quant_bitdepth[i]	u(5)
if(mps_unification_flag || lps_unification_flag){	
ctu_scan_order[i]	u(1)
nnr_reserved_zero_2bits	u(2)
}	
else	
nnr_reserved_zero_3bits	u(3)

}	
}	

6.3.4 NNR unit payload syntax

6.3.4.1 General

nnr_unit_payload( ) {	**Descriptor**
if(nnr_unit_type == NNR_MPS)	
nnr_model_parameter_set_unit_payload( )	
if(nnr_unit_type == NNR_LPS)	
nnr_layer_parameter_set_unit_payload( )	
if(nnr_unit_type == NNR_TPL)	
nnr_topology_unit_payload( )	
if(nnr_unit_type == NNR_QNT)	
nnr_quantization_unit_payload( )	
if(nnr_unit_type == NNR_NDU)	
nnr_compressed_data_unit_payload( )	
if(nnr_unit_type == NNR_STR)	
nnr_start_unit_payload( )	
if(nnr_unit_type == NNR_AGG)	
nnr_aggregate_unit_payload( )	
}	

6.3.4.2 NNR start unit payload syntax

nnr_start_unit_payload( ) {	**Descriptor**
nnr_reserved_zero_0bit	u(0)
}	

6.3.4.3 NNR model parameter set unit payload syntax

nnr_model_parameter_set_unit_payload( ) {	**Descriptor**
topology_carriage_flag	u(1)
mps_sparsification_flag	u(1)
mps_pruning_flag	u(1)
mps_unification_flag	u(1)
mps_decomposition_performance_map_flag	u(1)
mps_quantization_method_flags	u(3)
mps_topology_indexed_reference_flag	u(1)
nnr_reserved_zero_7bits	u(7)
if((mps_quantization_method_flags & NNR_QSU) == NNR_QSU || (mps_quantization_method_flags & NNR_QCB) == NNR_QCB) {	
mps_qp_density	u(3)
mps_quantization_parameter	i(13)
}	

if(mps_sparsification_flag == 1)	
sparsification_performance_map( )	
if(mps_pruning_flag == 1)	
pruning_performance_map( )	
if(mps_unification_flag == 1)	
unification_performance_map( )	
if(mps_decomposition_performance_map_flag == 1)	
decomposition_performance_map( )	
byte_alignment( )	
}	

sparsification_performance_map() is defined as follows:

sparsification_performance_map( ) {	**Descriptor**
spm_count_thresholds	u(8)
for(i = 0; i < (spm_count_thresholds−1); i++) {	
sparsification_threshold[i]	flt(32)
non_zero_ratio[i]	flt(32)
spm_nn_accuracy[i]	flt(32)
spm_count_classes[i]	u(8)
spm_class_bitmask[i]	ue(7)
for (j = 0; j < spm_count_classes[i]; j++)	
spm_nn_class_accuracy[i][j]	flt(32)
}	
}	

pruning_performance_map() is defined as follows:

pruning_performance_map( ) {	**Descriptor**
ppm_count_pruning_ratios	u(8)
for(i = 0; i < (ppm_count_pruning_ratios−1); i++) {	
pruning_ratio[i]	flt(32)
ppm_nn_accuracy[i]	flt(32)
ppm_count_classes[i]	u(8)
ppm_class_bitmask[i]	ue(7)
for(j = 0; j < ppm_count_classes[i]; j++)	
ppm_nn_class_accuracy[i][j]	flt(32)
}	
}	

unification_performance_map() is defined as follows:

unification_performance_map( ) {	**Descriptor**

upm_count_thresholds	u(8)
for(i = 0; i < (upm_count_thresholds−1); i++) {	
count_reshaped_tensor_dimension	ue(1)
for(j = 0; j < (count_reshaped_tensor_dimension−1); j++)	
reshaped_tensor_dimensions[j]	ue(7)
byte_alignment( )	
count_super_block_dimension	u(8)
for(j = 0; j < (count_super_block_dimension−1); j++)	
super_block_dimensions[j]	u(8)
count_block_dimension	u(8)
for(j = 0; j < (count_block_dimension−1); j++)	
block_dimensions[j]	u(8)
unification_threshold[i]	flt(32)
upm_nn_accuracy[i]	flt(32)
upm_count_classes[i]	u(8)
upm_class_bitmask[i]	ue(7)
for(j = 0; j < upm_count_classes[i]; j++)	
upm_nn_class_accuracy[i][j]	flt(32)
}	
}	

Decomposition_performance_map() is defined as follows:

decomposition_performance_map( ) {	**Descriptor**
dpm_count_thresholds	u(8)
for(i = 0; i < (dpm_count_thresholds−1); i++) {	
mse_threshold[i]	flt(32)
dpm_nn_accuracy[i]	flt(32)
nn_reduction_ratio[i]	flt(32)
dpm_count_classes[i]	u(16)
for(j = 0; j < dpm_count_classes[i]; j++)	
dpm_nn_class_accuracy[i][j]	flt(32)
}	
}	

6.3.4.4 NNR layer parameter set unit payload syntax

nnr_layer_parameter_set_unit_payload( ) {	**Descriptor**
nnr_reserved_zero_1bit	u(1)
lps_sparsification_flag	u(1)
lps_pruning_flag	u(1)
lps_unification_flag	u(1)
lps_quantization_method_flags	u(3)
nnr_reserved_zero_1bit	u(1)

if((lps_quantization_method_flags & NNR_QCB) == NNR_QCB || (lps_quantization_method_flags & NNR_QSU) == NNR_QSU) {	
lps_qp_density	u(3)
lps_quantization_parameter	i(13)
}	
if(lps_sparsification_flag == 1)	
sparsification_performance_map( )	
if(lps_pruning_flag == 1)	
pruning_performance_map( )	
if(lps_unification_flag == 1)	
unification_performance_map( )	
byte_alignment( )	
}	

6.3.4.5 NNR topology unit payload syntax

nnr_topology_unit_payload( ) {	**Descriptor**
if(topology_storage_format == NNR_TPL_PRUN)	
nnr_pruning_topology_container()	
else if(topology_storage_format == NNR_TPL_REFLIST)	
topology_elements_ids_list(0)	
else	
topology_data	bs(v)
}	

nnr_pruning_topology_container() is specified as follows:

nnr_pruning_topology_container( ) {	**Descriptor**
nnr_rep_type	u(2)
prune_flag	u(1)
order_flag	u(1)
sparse_flag	u(1)
nnr_reserved_zero_3bits	u(3)
if (prune_flag == 1) {	
if (nnr_rep_type == NNR_TPL_BMSK)	
bit_mask( )	
else if (nnr_rep_type == NNR_TPL_ DICT) {	
count_ids	ue(7)
if (!mps_topology_indexed_reference_flag) {	
byte_alignment()	
for (j = 0; j < count_ids; j++) {	
element_id[j]	st(v)
}	
}	
else {	

for (j = 0; j < count_ids; j++) {	
element_id_index[j]	ue(7)
}	
}	
for (j = 0; j < count_ids; j++) {	
count_dims[j]	ue(1)
for(k = 0; k < count_dims[j]; k++){	
dim[j][k]	ue(7)
}	
}	
byte_alignment( )	
}	
}	
if (sparse_flag == 1) {	
bit_mask( )	
}	
}	

bit_mask() is specified as follows:

bit_mask( ) {	**Descriptor**
count_bits	u(32)
for(j = 0; j < count_bits; j++) {	
bit_mask_value[j]	u(1)
}	
byte_alignment( )	
}	

6.3.4.6 NNR quantization unit payload syntax

nnr_quantization_unit_payload( ) {	**Descriptor**
quantization_data	bs(v)
}	

6.3.4.7 NNR compressed data unit payload syntax

nnr_compressed_data_unit_payload( ) {	**Descriptor**
if(nnr_compressed_data_unit_payload_type == NNR_PT_RAW_FLOAT)	
for(i = 0; i < Prod(TensorDimensions); i++)	
raw_float32_parameter[TensorIndex(TensorDimensions, i , 0)]	flt(32)
decode_compressed_data_unit_payload( )	
}	

decode_compressed_data_unit_payload() invokes the decoding process as specified in subclause 7.3.

6.3.4.8 NNR aggregate unit payload syntax

nnr_aggregate_unit_payload( ) {	Descriptor
for(i = 0; i < num_of_nnr_units_minus2 + 2; i++)	
nnr_unit( )	
}	

6.3.5 Byte alignment syntax

byte_alignment( ) {	Descriptor
alignment_bit_equal_to_one /* equal to 1 */	f(1)
while(!byte_aligned( ))	
alignment_bit_equal_to_zero /* equal to 0 */	f(1)
}	

6.4 Semantics

6.4.1 General

Semantics associated with the syntax structures and elements within these structures are specified in this subclause. When the semantics of a syntax element are specified using a table or a set of tables, any values that are not specified in the table(s) shall not be present in the bitstream unless otherwise specified in this document.

6.4.2 NNR unit size semantics

nnr_unit_size_flag specifies the number of bits used as the data type of the nnr_unit_size. If this value is 0, then nnr_unit_size is a 15 bits unsigned integer value, otherwise it is 31 bits unsigned integer value.

nnr_unit_size specifies the size of the NNR unit, which is the sum of byte sizes of nnr_unit_size(), nnr_unit_header() and nnr_unit_payload().

6.4.3 NNR unit header semantics

6.4.3.1 General

nnr_unit_type specifies the type of the NNR unit, as specified in Table 3.

Table 3 — NNR unit Types

nnr_unit_type	Identifier	NNR unit Type	Description
0	NNR_STR	NNR start unit	Compressed neural network bitstream start indicator
1	NNR_MPS	NNR model parameter set data unit	Neural network global metadata and information
2	NNR_LPS	NNR layer parameter set data unit	Metadata related to a partial representation of neural network
3	NNR_TPL	NNR topology data unit	Neural network topology information
4	NNR_QNT	NNR quantization data unit	Neural network quantization information
5	NNR_NDU	NNR compressed data unit	Compressed neural network data
6	NNR_AGG	NNR aggregate unit	NNR unit with payload containing multiple NNR units
7..31	NNR_RSVD	Reserved	ISO/IEC-reserved range

Table 3 *(continued)*

nnr_unit_type	Identifier	NNR unit Type	Description
32..63	NNR_UNSP	Unspecified	Unspecified range

The values in the range NNR_RSVD are reserved for used in future versions of this or related specifications. Encoders shall not use these values. Decoders conforming to this version of the specification may ignore NNR units using these values. The values in the range NNR_UNSP are not specified, their use is outside the scope of this specification. Decoders conforming to this version of the specification may ignore NNR units using these values.

independently_decodable_flag specifies whether this compressed data unit is independently decodable. A value of 1 indicates an independently decodable NNR unit. A value of 0 indicates that this NNR unit is not independently decodable and its payload should be combined with other NNR units for successful decodability/decompressibility. The value of independently_decodable_flag shall be the same for all NNR units which refer to the same topology_elem_id or topology_elem_id_index value or the same topology_elem_id_list.

partial_data_counter_present_flag equal to 1 specifies that the syntax element partial_data_counter is present in NNR unit header. partial_data_counter_present_flag equal to 0 specifies that the syntax element partial_data_counter is not present in NNR unit header.

partial_data_counter specifies the index of the partial data carried in the payload of this NNR data unit with respect to the whole data for a certain topology element. A value of 0 indicates no partial information (i.e., the data in this NNR unit is all data associated to a topology element and it is complete), a value bigger than 0 indicates the index of the partial information (i.e., data in this NNR unit should be concatenated with the data in accompanying NNR units until partial_data_counter of an NNR unit reaches 1). This counter counts backwards to indicate initially the total number of partitions. If not present, the value of partial_data_counter is inferred to be equal to 0. If the value of independently_decodable_flag is equal to 0, the value of partial_data_counter_present_flag shall be equal to 1 and the value of partial_data_counter shall be greater than 0. If the value of independently_decodable_flag is equal to 1, the values of partial_data_counter_present_flag and partial_data_counter are undefined, in this version of this document.

NOTE In future versions of this document, if the value of independently_decodable_flag is equal to 1 and if partial_data_counter_present_flag is equal to 1, partial_data_counter can have non-zero values, based on the assumption that multiple independently decodable NNR units are combined to construct a model.

6.4.3.2 NNR start unit header semantics

general_profile_idc indicates a profile to which NNR bitstream conforms as specified in this document. Reserved for future use.

6.4.3.3 NNR model parameter set unit header semantics

Header elements of the model parameter set (reserved for future use).

6.4.3.4 NNR layer parameter set unit header semantics

lps_self_contained_flag equal to 1 specifies that NNR units that refer to the layer parameter set are a full or partial NN model and shall be successfully reconstructable with the NNR units. A value of 0 indicates that the NNR units that refer to the layer parameter set should be combined with NNR units that refer to other layer parameter sets for successful reconstruction of a full or partial NN model.

6.4.3.5 NNR topology unit header semantics

topology_storage_format specifies the format of the stored neural network topology information, as specified in Figure 4 and Table 4.

Table 4 — Topology storage format identifiers

topology_storage_format value	Identifier	Description
0	NNR_TPL_UNREC	Unrecognized topology format
1		Topology format shall be represented as specified in Annex A.
2..4		See Annexes B to D for further information.
5	NNR_TPL_PRUN	Topology pruning information
6	NNR_TPL_REFLIST	Topology element reference list information
7..127	NNR_TPL_RSVD	ISO/IEC-reserved range
128..255	NNR_TPL_UNSP	Unspecified range

The value NNR_TPL_UNREC indicates that the topology format is unknown. Encoders may use this value if the topology format used is not among the set of formats for which identifiers are specified. Decoders conforming to this version of the specification may ignore NNR units using this value or may attempt to recognize the format by parsing the start of the topology payload.

The values in the range NNR_TPL_RSVD are reserved for used in future versions of this or related specifications. Encoders shall not use these values. Decoders conforming to this version of the specification may ignore NNR units using these values. The values in the range NNR_TPL_UNSP are not specified, their use is outside the scope of this specification. Decoders conforming to this version of the specification may ignore NNR units using these values.

topology_compression_format specifies that one of the compression formats defined in Table 5 is applied on the stored topology data in topology_data.

Table 5 — Topology compression format identifiers

topology_compression_format	Identifier	Description
0x00	NNR_PT_RAW	Uncompressed
0x01	NNR_DFL	Deflate method, shall be implemented according to IETF RFC 1950
0x02-0xFF		Reserved

6.4.3.6 NNR quantization unit header semantics

quantization_storage_format specifies the format of the stored neural network quantization information, as specified in Table 6.

Table 6 — Quantization storage format identifiers

quantization_storage_format value	Identifier	Description
0	NNR_QNT_UNREC	Unrecognized quantization format.
1		Quantisation format shall be represented as specified in Annex A.
2..4		See Annexes B to D for further information.
5..127	NNR_QNT_RSVD	ISO/IEC-reserved range
128..255	NNR_QNT_UNSP	Unspecified range

The value NNR_QNT_UNREC indicates that the quantization format is unknown. Encoders may use this value if the quantization format used is not among the set of formats for which identifiers are specified. Decoders conforming to this version of the specification may ignore NNR units using this value or may attempt to recognize the format by parsing the start of the topology payload.

The values in the range NNR_QNT_RSVD are reserved for used in future versions of this or related specifications. Encoders shall not use these values. Decoders conforming to this version of the specification may ignore NNR units using these values. The values in the range NNR_QNT_UNSP are not specified, their use is outside the scope of this specification. Decoders conforming to this version of the specification may ignore NNR units using these values.

quantization_compression_format specifies that one of the compression formats defined in Table 7 is applied on the stored quantization data in quantization_data.

Table 7 — Quantization compression format identifiers

quantization_compression_format	Identifier	Description
0x00	NNR_PT_RAW	Uncompressed
0x01	NNR_DFL	Deflate method, shall be implemented according to IETF RFC 1950
0x02-0xFF		Reserved

6.4.3.7 NNR compressed data unit header semantics

nnr_compressed_data_unit_payload_type is as defined in Table 14 of subclause 7.3.

nnr_multiple_topology_elements_present_flag specifies whether multiple topology units are present in the bitstream. In case there are multiple units, the list of their IDs is included. When nnr_compressed_data_unit_payload_type is set to NNR_PT_BLOCK, this flag shall be set to 1 and topology_elements_ids_list() in the NNR compressed data unit header shall list the topology elements or topology element indexes of RecWeight, RecWeightG, RecWeightH, RecLS, RecBeta, RecGamma, RecMean, RecVar and RecBias, in the given order and based on their presence as indicated by the value of compressed_parameter_type in the NNR compressed data unit header.

nnr_decompressed_data_format_present_flag specifies whether the data format to be obtained after decompression is present in the bitstream.

input_parameters_present_flag specifies whether the group of elements including tensor dimensions, DeepCABAC unary length and compressed parameter types is present in the bitstream.

topology_elem_id specifies a unique identifier for the topology element to which an NNR compressed data unit refers. The semantic interpretation of this field is context dependent.

topology_elem_id_index specifies a unique index value of a topology element which is signaled in topology information of payload type NNR_TPL_REFLIST. The first index shall be 0 (i.e. 0-indexed).

count_topology_elements_minus2 + 2 specifies the number of topology elements for which this NNR compressed data unit carries data in the payload.

codebook_present_flag specifies whether codebooks are used. If codebook_present_flag is not present, it is inferred to be 0.

dq_flag specifies whether the quantization method is dependent scalar quantization according to subclause 9.1.3 or uniform quantization according to subclause 9.1.1. A dq_flag equal to 0 indicates that the uniform quantization method is used. A dq_flag equal to 1 indicates that the dependent scalar quantization method is used. If dq_flag is not present, it is inferred to be 0.

nnr_decompressed_data_format is defined in Table 13 of subclause 7.2.

tensor_dimensions_flag specifies whether the tensor dimensions are defined in the bitstream. If they are not included in the bitstream, they shall be obtained from the model topology description.

cabac_unary_length_flag specifies whether the length of the unary part in the DeepCABAC binarization is included in the bitstream.

compressed_parameter_types specifies the compressed parameter types present in the current topology element to which an NNR compressed data unit refers. If multiple compressed parameter types are specified, they are combined by OR. The compressed parameter types are defined in Table 8.

Table 8 — Compressed parameter type identifiers

Compressed parameter type	Compressed parameter type ID	Bit in compressed_parameter_types
Decomposition present	NNR_CPT_DC	0x01
Local scaling present	NNR_CPT_LS	0x02
Batch norm parameters present	NNR_CPT_BN	0x04
Bias present	NNR_CPT_BI	0x08

When decomposition is present, the tensors G and H represent the result of decomposing the original tensor. If (compressed_parameter_types & NNR_CPT_DC) != 0 the variables TensorDimensionsG and TensorDimensionsH are derived as follows:

— Variable TensorDimensionsG is set to [g_number_of_rows, decomposition_rank].

— Variable TensorDimensionsH is set to [decomposition_rank, hNumberOfColumns] where hNumberOfColumns is defined as

$$\text{hNumberOfColumns} = \frac{\prod_{i=0}^{\text{count\_tensor\_dimensions}-1} \text{tensor\_dimensions}[i]}{\text{g\_number\_of\_rows}}$$

If (compressed_parameter_types & NNR_CPT_DC) != 0 and if nnr_compressed_data_unit_payload_type != NNR_PT_BLOCK, the NNR unit contains a decomposed tensor G and the next NNR unit in the bitstream contains the corresponding decomposed tensor H.

A variable TensorDimensions is derived as follows:

— If an NNR unit contains a decomposed tensor G and nnr_compressed_data_unit_payload_type != NNR_PT_BLOCK, TensorDimensions is set to TensorDimensionsG.

— Otherwise, if an NNR unit contains a decomposed tensor H and nnr_compressed_data_unit_payload_type != NNR_PT_BLOCK, TensorDimensions is set to TensorDimensionsH.

— Otherwise, TensorDimensions is set to tensor_dimensions.

A variable NumBlockRowsMinus1 is defined as follows:

— If scan_order is equal to 0, NumBlockRowsMinus1 is set to 0.

— Otherwise, if nnr_compressed_data_unit_payload_type == NNR_PT_BLOCK and (compressed_parameter_types & NNR_CPT_DC) != 0, NumBlockRowsMinus1 is set to ((TensorDimensionsG[0] + (4 << scan_order) − 1) >> (2 + scan_order)) + ((TensorDimensionsH[0] + (4 << scan_order) − 1) >> (2 + scan_order)) − 2.

— Otherwise, NumBlockRowsMinus1 is set to ((TensorDimensions[0] + (4 << scan_order) − 1) >> (2 + scan_order)) − 1.

decomposition_rank specifies the rank of the low-rank decomposed weight tensor components relative to tensor_dimensions.

g_number_of_rows specifies the number of rows of matrix G in the case where the reconstruction is performed for decomposed tensors in an NNR unit of type NNR_PT_BLOCK

cabac_unary_length_minus1 specifies the length of the unary part in the DeepCABAC binarization minus 1.

scan_order specifies the block scanning order for parameters with more than one dimension according to the following table:

— 0: No block scanning

— 1: 8x8 blocks

— 2: 16x16 blocks

— 3: 32x32 blocks

— 4: 64x64 blocks

cabac_offset_list specifies a list of values to be used to initialize variable IvlOffset at the beginning of entry points.

dq_state_list specifies a list of values to be used to initialize variable stateId at the beginning of entry points.

bit_offset_delta1 specifies the first element of list BitOffsetList.

bit_offset_delta2 specifies elements of list BitOffsetList except for the first element, as difference to the previous element of list BitOffsetList.

Variable BitOffsetList is a list of bit offsets to be used to set the bitstream pointer position at the beginning of entry points.

codebook_egk specifies the Exp-Golomb parameter k for decoding of syntax elements codebook_delta_left and codebook_delta_right.

codebook_size specifies the number of elements in the codebook.

codebook_centre_offset specifies an offset for accessing elements in the codebook relative to the centre of the codebook. It is used for calculating variable CbZeroOffset.

codebook_zero_value specifies the value of the codebook at position CbZeroOffset. It is involved in creating variable Codebook (the array representing the codebook).

codebook_delta_left specifies the difference between a codebook value and its right neighbour minus 1 for values left to the centre position. It is involved in creating variable Codebook (the array representing the codebook).

codebook_delta_right specifies the difference between a codebook value and its left neighbour minus 1 for values right to the centre position. It is involved in creating variable Codebook (the array representing the codebook).

count_tensor_dimensions specifies a counter of how many dimensions are specified. For example, for a 4-dimensional tensor, count_tensor_dimensions is 4. If it is not included in the bitstream, it shall be obtained from the model topology description.

tensor_dimensions specifies an array or list of dimension values. For example, for a convolutional layer, tensor_dimensions is an array or list of length 4. For NNR units carrying elements G or H of a decomposed tensor, tensor_dimensions is set to the dimensions of the original tensor. The actual tensor dimensions of G and H for the decoding methods are derived from tensor_dimensions, decomposition_rank, and g_number_of_rows. If it is not included in the bitstream, it shall be obtained from the model topology description.

topology_elem_id_list specifies a list of unique identifiers related to the topology element to which an NNR compressed data unit refers. Elements of topology_elem_id_list are semantically equivalent to syntax element topology_elem_id or the index of it when listed in topology payload of type NNR_TPL_REFLIST. The semantic interpretation of this field is context dependent.

topology_elem_id_index_list specifies a list of unique indexes related to the topology elements listed in topology information with payload type NNR_TPL_REFLIST. The first element in the topology shall have the index value of 0.

concatentation_axis_index indicates the 0-based concatenation axis.

split_index[] indicates the tensor splitting index along the concatenation axis indicated by concatentation_axis_index in order to generate each individual tensor which is concatenated.

number_of_shifts[] indicates how many left-shifting operations are to be performed.

shift_index[k][i] indicates the axis index of the kth topology element to be left-shifted.

shift_value[k][i] indicates the amount of left-shift on the axis with index index[k][i].

6.4.3.8 NNR aggregate unit header semantics

nnr_aggregate_unit_type specifies the type of the aggregate NNR unit.

The NNR aggregate unit types are specified in Table 9.

Table 9 — NNR aggregate unit types

nnr_aggregate_unit_type	Identifier	NNR Aggregate Unit Type	Description
0	NNR_AGG_GEN	Generic NNR aggregate unit	A set of NNR units
1	NNR_AGG_SLF	Self-contained NNR aggregate unit	When extracted and then concatenated with an NNR_STR and NNR_MPS, an NNR_AGG_SLF shall be decodable without any need of additional information and a full or partial NN model shall be successfully reconstructable with it.
2..127	NNR_RSVD	Reserved	ISO/IEC-reserved range
128..255	NNR_UNSP	Unspecified	Unspecified range

The values in the range NNR_ NNR_RSVD are reserved for uses in future versions of this or related specifications. Encoders shall not use these values. Decoders conforming to this version of the specification may ignore NNR units using these values. The values in the range NNR_UNSP are not specified, their use is outside the scope of this specification. Decoders conforming to this version of the specification may ignore NNR units using these values.

entry_points_present_flag specifies whether individual NNR unit entry points are present.

num_of_nnr_units_minus2 + 2 specifies the number of NNR units present in the NNR aggregate unit's payload.

nnr_unit_type[i] specifies the NNR unit type of the NNR unit with index i. This value shall be the same as the NNR unit type of the NNR unit at index i.

nnr_unit_entry_point[i] specifies the byte offset from the start of the NNR aggregate unit to the start of the NNR unit in NNR aggregate unit's payload and at index i. This value shall not be equal or greater than the total byte size of the NNR aggregate unit. nnr_unit_entry_point values can be used for fast and random access to NNR units inside the NNR aggregate unit payload.

quant_bitdepth[i] specify the max bit depth of quantized coefficients for each tensor in the NNR aggregate unit.

ctu_scan_order[i] specify the CTU-wise scan order for each tensor in the NNR aggregate unit. Value 0 indicates that the CTU-wise scan order is raster scan order at horizontal direction, value 1 indicates that the CTU-wise scan order is raster scan order at vertical direction.

6.4.4 NNR unit payload semantics

6.4.4.1 General

The following NNR unit payload types are specified:

6.4.4.2 NNR start unit payload semantics

Start unit payload (reserved for future use).

6.4.4.3 NNR model parameter set unit payload semantics

topology_carriage_flag specifies whether the NNR bitstream carries the topology internally or externally. When set to 1, it specifies that topology is carried within one or more NNR units of type “NNR_TPL”. If 0, it specifies that topology is provided externally (i.e., out-of-band with respect to the NNR bitstream).

mps_sparsification_flag specifies whether sparsification is applied to the model in the NNR compressed data units that utilize this model parameter set.

mps_pruning_flag specifies whether pruning is applied to the model in the NNR compressed data units that utilize this model parameter set.

mps_unification_flag specifies whether unification is applied to the model in the NNR compressed data units that utilize this model parameter set.

mps_decomposition_performance_map_flag equal to 1 specifies that tensor decomposition was applied to at least one layer of the model and a corresponding performance map is transmitted.

mps_quantization_method_flags specifies the quantization method(s) used for the model in the NNR compressed data units that utilize this model parameter set. If multiple models are specified, they are combined by OR. The methods are defined in Table 10.

Table 10 — Quantization method identifiers

Quantization method	Quantization method ID	Value
Scalar uniform	NNR_QSU	0x01
Codebook	NNR_QCB	0x02
Reserved		0x04-0x07

mps_topology_indexed_reference_flag specifies whether topology elements are referenced by unique index. When set to 1, topology elements are represented by their indexes in the topology data defined by the topology payload of type NNR_TPL_REFLIST. If this flag is set to 0, then topology_data of NNR topology unit shall contain the topology information.

mps_qp_density specifies density information of syntax element mps_quantization_parameter in the NNR compressed data units that utilize this model parameter sets.

mps_quantization_parameter specifies the quantization parameter for scalar uniform quantization of parameters of each layer of the neural network for arithmetic coding in the NNR compressed data units that utilize this model parameter set.

sparsification_performance_map() specifies a mapping between different sparsification thresholds and resulting NN inference accuracies. The resulting accuracies are provided separately for different aspects or characteristics of the output of the NN. For a classifier NN, each sparsification threshold is mapped to separate accuracies for each class, in addition to an overall accuracy which considers all classes. Classes are ordered based on the neural network output order, i.e., the order specified during training.

spm_count_thresholds specifies the number of sparsification thresholds. This number shall be non-zero.

sparsification_threshold specifies a list of thresholds where each threshold is applied to the weights of the decoded neural network in order to set the weights to zero. I.e., the weights whose values are less than the threshold are set to zero.

non_zero_ratio specifies a list of non-zero ratio values where each value is the non-zero ratio that is achieved by applying the sparsification_threshold to sparsify the weights.

spm_nn_accuracy specifies a list of accuracy values where each value is the overall accuracy of the NN (e.g. classification accuracy by considering all classes) when sparsification using the corresponding threshold in sparsification_threshold is applied.

spm_count_classes specifies a list of number of classes where each such number is the number of classes for which separate accuracies are provided for each sparsification thresholds.

spm_class_bitmask specifies a subset of classes for which the accuracies are signalled, when a certain sparsification threshold is applied. The order of bits indicates the indexes of classes, with the most significant bit representing the presence of the smallest indexed class.

spm_nn_class_accuracy specifies a list of lists of class accuracies, where each value is accuracy for a certain class, when a certain sparsification threshold is applied.

pruning_performance_map() specifies a mapping between different pruning ratios and resulting NN inference accuracies. The resulting accuracies are provided separately for different aspects or characteristics of the output of the NN. For a classifier NN, each pruning ratio is mapped to separate accuracies for each class, in addition to an overall accuracy which considers all classes. Classes are ordered based on the neural network output order, i.e., the order specified during training.

ppm_count_pruning_ratios specifies the number of pruning ratios. This number shall be non-zero.

pruning_ratio specifies the pruning ratio.

ppm_nn_accuracy specifies a list of accuracy values where each value is the overall accuracy of the NN (e.g. classification accuracy by considering all classes) when pruning using the corresponding ratio in pruning_ratio is applied.

ppm_class_bitmask: specifies a subset of classes for which corresponding accuracies are signalled, when a certain pruning ratio is applied. The order of bits indicates the indexes of classes, with the most significant bit representing the presence of the smallest indexed class.

ppm_count_classes specifies a list of number of classes where each such number is the number of classes for which separate accuracies are provided for each pruning ratio.

ppm_nn_class_accuracy specifies a list of lists of class accuracies, where each value is accuracy for a certain class, when a certain pruning ratio is applied.

unification_performance_map() specifies a mapping between different unification thresholds and resulting NN inference accuracies. The resulting accuracies are provided separately for different aspects or characteristics of the output of the NN. For a classifier NN, each unification threshold is mapped to separate accuracies for each class, in addition to an overall accuracy which considers all classes. Classes are ordered based on the neural network output order, i.e., the order specified during training.

upm_count_thresholds specifies the number of unification thresholds. This number shall be non-zero.

count_reshaped_tensor_dimensions specifies a counter of how many dimensions are specified for reshaped tensor. For example, for a weight tensor reshaped to 3-dimensional tensor, count_reshaped_tensor_dimensions is 3.

reshaped_tensor_dimensions specifies an array or list of dimension values. For example, for a convolutional layer reshaped to 3-dimensional tensor, dim is an array or list of length 3.

count_super_block_dimensions specifies a counter of how many dimensions are specified. For example, for a 3-dimensional superblock, count_super_block_dimensions is 3.

super_block_dimensions specifies an array or list of dimension values. For example, for a 3-dimensional superblock, dim is an array or list of length 3, i.e. [64, 64, kernel_size].

count_block_dimensions specifies a counter of how many dimensions are specified. For example, for a 3-dimensional block, count_block_dimensions is 3.

block_dimensions specifies an array or list of dimension values. For example, for a 3-dimensional block, dim is an array or list of length 3, i.e. [2, 2, 2].

unification_threshold specifies the threshold which is applied to tensor block in order to unify the absolute value of weights in this tensor block.

upm_nn_accuracy specifies the overall accuracy of the NN (e.g. classification accuracy by considering all classes).

upm_count_classes specifies number of classes for which separate accuracies are provided for each unification thresholds.

upm_class_bitmask specifies a subset of classes for which corresponding accuracies are signalled, when a certain unification threshold is applied. The order of bits indicates the indexes of classes, with the most significant bit representing the presence of the smallest indexed class.

upm_nn_class_accuracy specifies the accuracy for a certain class, when a certain unification threshold is applied.

decomposition_performance_map() specifies a mapping between different mean square error (MSE) thresholds between the decomposed tensors and their original version and resulting NN inference accuracies. The resulting accuracies are provided separately for different aspects or characteristics of the output of the NN. For a classifier NN, each MSE threshold is mapped to separate accuracies for each class, in addition to an overall accuracy which considers all classes. Classes are ordered based on the neural network output order, i.e., the order specified during training.

dpm_count_thresholds specifies the number of decomposition MSE thresholds. This number shall be non-zero.

mse_threshold specifies an array of MSE thresholds which are applied to derive the ranks of the different tensors of weights.

dpm_nn_accuracy specifies the overall accuracy of the NN (e.g. classification accuracy by considering all classes).

nn_reduction_ratio[i] specifies the ratio between the total number of parameters after tensor decomposition of the whole model and the number of parameters in the original model.

dpm_count_classes specifies number of classes for which separate accuracies are provided for each decomposition thresholds.

dpm_nn_class_accuracy specifies an array of accuracies for a certain class, when a certain decomposition threshold is applied.

6.4.4.4 NNR layer parameter set unit payload semantics

lps_sparsification_flag specifies whether sparsification was applied to the model in the NNR compressed data units that utilizes this layer parameter set.

lps_pruning_flag specifies whether pruning was applied to the model in the NNR compressed data units that utilizes this layer parameter set.

lps_unification_flag specifies whether unification was applied to the model in the NNR compressed data units that utilizes this layer parameter set.

lps_quantization_method_flags specifies the quantization method used for the data contained in the NNR compressed data units to which this layer parameter set refers. If multiple models are specified, they are combined by OR. The methods are defined in Table 11.

Table 11 — Quantization method identifiers

Quantization method	Quantization method ID	Value
Scalar uniform	NNR_QSU	0x01
Codebook	NNR_QCB	0x02
Reserved		0x04-0x07

lps_qp_density specifies density information of syntax element lps_quantization_parameter in the NNR compressed data units that utilize this model parameter set.

lps_quantization_parameter specifies the quantization parameter for scalar uniform quantization of parameters of each layer of the neural network for arithmetic coding in the NNR compressed data units that utilize this model parameter set.

The variable QpDensity is derived as follows:

— If an active NNR layer parameter set is present, the variable QpDensity is set to lps_qp_density.

— Otherwise, the variable QpDensity is set to mps_qp_density.

The variable QuantizationParameter is derived as follows:

— If an active NNR layer parameter set is present, the variable QuantizationParameter is set to lps_quantization_parameter.

— Otherwise, the variable QuantizationParameter is set to mps_quantization_parameter.

sparsification_performance_map() is as defined in subclause 6.4.4.3.

When lps_sparsification_flag of a certain layer is equal to 1 and mps_sparsification_flag is equal to 0, then the information in sparsification_performance_map() of the layer parameter set is valid when performing sparsification only on that layer. More than one layer can have lps_sparsification_flag equal to 1 in their layer parameter set.

When both mps_sparsification_flag and lps_sparsification_flag are equal to 1, the following shall apply:

— If sparsification is applied on the whole model (i.e., all layers), then the information in sparsification_performance_map() of the model parameter set is valid.

— If sparsification is applied on only one layer, and for that layer lps_sparsification_flag is equal to 1, then the information in sparsification_performance_map() of the layer parameter set of that layer is valid.

pruning_performance_map() is as defined in subclause 6.4.4.3.

When lps_pruning_flag of a certain layer is equal to 1 and mps_pruning_flag is equal to 0, then the information in pruning_performance_map() of the layer parameter set is valid when performing pruning only on that layer. More than one layer can have lps_pruning_flag equal to 1 in their layer parameter set.

When both mps_pruning_flag and lps_pruning_flag are equal to 1, the following shall apply:

— If pruning is applied on the whole model (i.e., all layers), then the information in pruning_performance_map() of the model parameter set is valid.

— If pruning is applied on only one layer, and for that layer lps_pruning_flag is equal to 1, then the information in pruning_performance_map() of the layer parameter set of that layer is valid.

unification_performance_map() is as defined in subclause 6.4.4.3.

When lps_unification_flag of a certain layer is equal to 1 and mps_unification_flag is equal to 0, then the information in unification_performance_map() of the layer parameter set is valid when performing unification only on that layer. More than one layer can have lps_unification_flag equal to 1 in their layer parameter set.

When both mps_unification_flag and lps_unification_flag are equal to 1, the following shall apply:

— If unification is applied on the whole model (i.e., all layers), then the information in unification_performance_map() of the model parameter set is valid.

— If unification is applied on only one layer, and for that layer lps_unification_flag is equal to 1, then the information in unification_performance_map() of the layer parameter set of that layer is valid.

6.4.4.5 NNR topology unit payload semantics

topology_storage_format value is as signaled in the corresponding NNR topology unit header of the same NNR unit of type NNR_TPL.

topology_data is a byte sequence of length determined by the NNR unit size describing the neural network topology, in the format specified by topology_storage_format.

If topology_storage_format is set to NNR_TPL_UNREC, definition and identification of the storage format of topology_data is out of scope of this document.

NOTE If topology_storage_format is set to NNR_TPL_UNREC, the (header) structure of topology_data can be used to identify the format.

nnr_rep_type specifies whether pruning information is represented as a bitmask or as a dictionary of references of topology elements. The permitted values are specified in Table 12.

Table 12 — Pruning information representation types

nnr_rep_type value	Identifier	Description
0x00	NNR_TPL_BMSK	Topology related information signaled as bitmask
0x01	NNR_TPL_DICT	Topology related information signaled as dictionary of topology elements
0x02-0x03		Reserved

prune_flag when set to 1 indicates that pruning step is used during parameter reduction and pruning related topology information is present in the payload.

order_flag when set to 1 indicates that the bitmask should be processed row-major order; and column-major otherwise.

sparse_flag when set to 1 indicates that sparsification step is used during parameter reduction and related topology information is present in the payload.

count_ids specifies the number of element ids that are updated. When present, its value shall be greater than zero.

element_id specifies the unique id that is used to reference a topology element

element_id_index specifies the unique index of the topology element which is present in the nnr_topology_unit_payload( ) where topology_storage_format is equal to NNR_TPL_REFLIST.

count_dims specifies the number of dimensions. When present, its value shall be greater than zero.

dim specifies array of dimensions that contain the new dimensions for the specified element. When present, its value shall be greater than zero.

bit_mask_value when set to 1 indicates that this specific neuron's weight is pruned if pruning_flag is set to 1 or is sparisfied (the weight value is 0) if sparse_flag is set to 1.

count_bits specifies the number of bits present in the bit mask information. When present, its value shall be greater than zero.

6.4.4.6 NNR quantization unit payload semantics

quantization_data is a byte sequence of length determined by the NNR unit size describing the neural network quantization information, in the format specified by quantization_storage_format.

If quantization_storage_format is set to NNR_QNT_UNREC, definition and identification of the storage format of quantization_data is out of scope of this document.

NOTE If quantization_storage_format is set to NNR_QNT_UNREC, the (header) structure of quantization_data can be used to identify the format.

6.4.4.7 NNR compressed data unit payload semantics

raw_float32_parameter is a float parameter tensor.

6.4.4.8 NNR aggregate unit payload semantics

NNR aggregate unit payload carries multiple NNR units. num_of_nnr_units_minus2 + 2 parameter in NNR aggregate unit header shall specify how many NNR units are present in the NNR aggregate unit's payload.

7 Decoding process

7.1 General

A decoder that complies with this document shall take an NNR bitstream, as specified in subclause 6.3, as input and

— generate decompressed data which complies with an NNR decompressed data format (as defined in Table 13) or

— generate ASCII or compressed data outputs as indicated by using the NNR_TPL and NNR_QNT NNR unit payloads (as described in subclause 6.3.3)

For the decoding process, the following conditions shall apply:

— Any information that is required for decoding an NNR unit of the NNR bitstream should be signaled as part of the NNR bitstream. If such information is not part of the NNR bitstream, then it shall be provided to the decoding process by other means (e.g. out-of-band topology information or parameters required for decoding but not signaled or carried in the NNR bitstream)

— The decoding process shall be initiated with an NNR unit of type NNR_STR. With the reception of the NNR_STR unit, the decoder shall reset its internal states and get ready to receive an NNR bitstream. The presence and cardinality of preceding NNR units shall be as specified in the relevant clauses and annexes of this document.

NOTE For example, a decoder can be further initialized via an NNR unit of type NNR_MPS in order set global neural network model parameters.

A decoder that complies with this document shall output data structures which comply with the decompressed NNR data formats as soon as it decompresses them. This allows low delay between inputting NNR compressed data units and accessing decompressed data structures from its output. How to establish the relationship between the input NNR units and NNR decompressed output data is out of scope of this document and left to implementation.

7.2 NNR decompressed data formats

Depending on the compression methods used to create a particular bitstream, the NNR decoder is expected to output different decompressed data formats as a result of decoding an NNR data unit. Table 13 specifies these NNR decompressed data formats that result after decompressing NNR compressed data units.

Table 13 — NNR decompressed data formats

Parameter identifier	**Parameter description**	nnr_decompressed_data_format
TENSOR_INT	Tensor of integer values used for representing tensor-shaped signed integer parameters of the model	0
TENSOR_FLOAT	Tensor of float values used for representing tensor-shaped float parameters of the model	1

7.3 Decoding methods

7.3.1 General

This subclause specifies the decoding methods of this document. Depending on the value of nnr_compressed_data_unit_payload_type, one of the subclauses as specified in Table 14 is invoked.

Table 14 — NNR compressed data payload types

Payload identifier	**Description**	nnr_compressed_data_unit_payload_type	**Sub-clause**
NNR_PT_INT	integer parameter tensor	0	7.3.2
NNR_PT_FLOAT	float parameter tensor	1	7.3.3
NNR_PT_RAW_FLOAT	uncompressed float parameter tensor	2	7.3.4
NNR_PT_BLOCK	float parameter tensors including a (optionally decomposed) weight tensor and, optionally, local scaling parameters, biases, and batch norm parameters that form a block in the model architecture	3	7.3.5

If the payload identifier is NNR_PT_INT, NNR_PT_FLOAT, or NNR_PT_FLOAT_RAW and if multiple topology elements are combined (as signaled in the NNR compressed data unit header via nnr_multiple_topology_elements_present_flag), then NNR decompressed tensors shall be further split into multiple tensors after the decoding process as follows:

— Tensor RecParam is split into multiple tensors by invoking TensorSplit(RecParam, split_index, concatenation_axis_index).

— The output of function TensorSplit is the list of split output tensors associated with topology elements as specified by array topology_elem_id_list.

— Output tensors are further processed by swapping their axis as signaled in topology_tensor_dimension_mapping() by invoking AxisSwap().

7.3.2 Decoding method for NNR compressed payloads of type NNR_PT_INT

Input to this process are:

— One or more NNR compressed data units which are marked to be decompressed together by partial_data_counter and nnr_compressed_data_unit_payload_type fields are set as NNR_PT_INT.

Output of this process is a variable RecParam of type TENSOR_INT as specified in Table 13. The dimensions of RecParam are equal to TensorDimensions. Decoding of a bitstream conforming to method NNR_PT_INT shall only produce values for RecParam that can be represented as 32 bit integer value in two's complement representation.

The arithmetic coding engine and context models are initialized as specified in subclause 10.3.2.

A syntax structure shift_parameter_ids(cabac_unary_length_minus1) according to subclause 10.2.1.6 is decoded from the bitstream and the initialization process for probability estimation parameters as specified in subclause 10.3.2.2 is invoked.

A syntax structure quant_tensor(TensorDimensions, cabac_unary_length_minus1, 0) according to subclause 10.2.1.4 is decoded from the bitstream and RecParam is set equal to QuantParam.

A syntax structure terminate_cabac() according to subclause 10.2.1.2 is decoded from the bitstream.

7.3.3 Decoding method for NNR compressed payloads of type NNR_PT_FLOAT

Input to this process are:

— One or more NNR compressed data units which are marked to be decompressed together by partial_data_counter and their nnr_compressed_data_unit_payload_type fields are set as NNR_PT_FLOAT

Output of this process is a variable RecParam of type TENSOR_FLOAT as specified in Table 13. The dimensions of RecParam are equal to TensorDimensions.

The arithmetic coding engine and context models are initialized as specified in subclause 10.3.2.

Subclause 7.3.6 is invoked with TensorDimensions, 0, and (codebook_present_flag ? 0 : -1) as inputs, and the output is assigned to RecParam.

A syntax structure terminate_cabac() according to subclause 10.2.1.2 is decoded from the bitstream.

Decoding of a bitstream conforming to method NNR_PT_FLOAT shall only produce values for RecParam that can be represented as float value without loss of precision.

7.3.4 Decoding method for NNR compressed payloads of type NNR_PT_RAW_FLOAT

Output of this process is a variable RecParam of type TENSOR_FLOAT as specified in Table 13. The dimensions of RecParam are equal to TensorDimensions.

RecParam is set equal to raw_float32_parameter.

7.3.5 Decoding method for NNR compressed payloads of type NNR_PT_BLOCK

Inputs to this process are:

— One or more NNR compressed data units which are marked to be decompressed together by partial_data_counter and their nnr_compressed_data_unit_payload_type fields are set as NNR_PT_BLOCK.

Output of this process are one or more variables of type TENSOR_FLOAT as specified in Table 13 depending on the value of compressed_parameter_types as follows:

If (compressed_parameter_types & NNR_CPT_DC) == 0: RecWeight

If (compressed_parameter_types & NNR_CPT_DC) != 0: RecWeightG, RecWeightH

If (compressed_parameter_types & NNR_CPT_LS) != 0: RecLS

If (compressed_parameter_types & NNR_CPT_BN) != 0: RecBeta, RecGamma, RecMean, RecVar

If (compressed_parameter_types & NNR_CPT_BI) != 0: RecBias

If present, the dimensions of RecWeight are equal to TensorDimensions.

If present, the dimensions of RecWeightG are equal to TensorDimensionsG.

If present, the dimensions of RecWeightH are equal to TensorDimensionsH.

If present, the variables RecLS, RecBeta, RecGamma, RecMean, RecVar, and RecBias are 1D and their length is equal to the first dimension of TensorDimensions.

The arithmetic coding engine and context models are initialized as specified in subclause 10.3.2.

If (compressed_parameter_types & NNR_CPT_LS) != 0, subclause 7.3.6 is invoked with the dimensions of RecLS, -1, and -1 as inputs, and the output is assigned to RecLS.

If (compressed_parameter_types & NNR_CPT_BI) != 0, subclause 7.3.6 is invoked with the dimensions of RecBias,, -1, and -1 as inputs, and the output is assigned to RecBias.

If (compressed_parameter_types & NNR_CPT_BN) != 0, subclause 7.3.6 is invoked with the dimensions of RecBeta, -1, and -1 as inputs, and the output is assigned to RecBeta.

If (compressed_parameter_types & NNR_CPT_BN) != 0, subclause 7.3.6 is invoked with the dimensions of RecGamma, -1, and -1 as inputs, and the output is assigned to RecGamma.

If (compressed_parameter_types & NNR_CPT_BN) != 0, subclause 7.3.6 is invoked with the dimensions of RecMean, -1, and -1 as inputs, and the output is assigned to RecMean.

If (compressed_parameter_types & NNR_CPT_BN) != 0, subclause 7.3.6 is invoked with the dimensions of RecVar, -1, and -1 as inputs, and the output is assigned to RecVar.

If (compressed_parameter_types & NNR_CPT_DC) == 0, subclause 7.3.6 is invoked with the dimensions of RecWeight, 0, and (codebook_present_flag ? 0 : -1) as inputs, and the output is assigned to RecWeight.

If (compressed_parameter_types & NNR_CPT_DC) != 0, the following applies:

Subclause 7.3.6 is invoked with TensorDimensionsG, 0, and (codebook_present_flag ? 0 : -1) as inputs, and the output is assigned to RecWeightG.

Subclause 7.3.6 is invoked with TensorDimensionsH, (TensorDimensionsG[0] + (4 << scan_order) – 1) >> (2 + scan_order)) – 1, and (codebook_present_flag ? 1 : -1) as inputs, and the output is assigned to RecWeightH.

NOTE From the decoded RecWeightG and RecWeightH, the variable RecWeight can be derived as follows:

RecWeight = TensorReshape (RecWeightG * RecWeightH, TensorDimensions)

A syntax structure terminate_cabac() according to subclause 10.2.1.2 is decoded from the bitstream.

7.3.6 Decoding process for an integer weight tensor

Inputs to this process are:

— A variable tensorDims specifying the dimensions of the tensor to be decoded.

— A variable entryPointOffset indicating whether entry points are present for decoding and, if entry points are present, an entry point offset.

— A variable codebookId indicating whether a codebook is applied and, if a codebook is applied, which codebook shall be used.

Output of this process is a variable recParam of type TENSOR_FLOAT as specified in Table 13 with dimensions equal to tensorDims.

A syntax structure quant_param(QpDensity) according to subclause 10.2.1.3 is decoded from the bitstream.

A syntax structure shift_parameter_ids(cabac_unary_length_minus1) according to subclause 10.2.1.6 is decoded from the bitstream and the initialization process for probability estimation parameters as specified in subclause 10.3.2.2 is invoked.

A syntax structure quant_tensor(tensorDims, cabac_unary_length_minus1, entryPointOffset) according to subclause 10.2.1.4 is decoded from the bitstream and recParam is set as follows:

```
if( codebookId == -1 )
    recParam = QuantParam
else {
    for( i = 0; i < Prod( tensorDims ); i++ ) {
        idx = TensorIndex( tensorDims, i )
        if( codebookId == 0 )
            recParam[idx] = Codebook[ QuantParam[idx] + CbZeroOffset ]
        else
            recParam[idx] = CodebookDC[ QuantParam[idx] + CbZeroOffsetDC ]
    }
}
```

A variable stepSize is derived as follows:

mul = (1 << QpDensity) + ((qp_value + QuantizationParameter) & ((1 << QpDensity) − 1))

shift = (qp_value + QuantizationParameter) >> QpDensity

stepSize = mul * $2^{\text{shift} - \text{QpDensity}}$

Variable recParam is updated as follows:

recParam = recParam * stepSize

NOTE Following from the above calculations, recParam can always be represented as binary fraction.

8 Parameter reduction

8.1 General

This includes methods and techniques that process a model to obtain a compact representation. Examples of such methods include, *parameter sparsification, parameter pruning, weight unification,* and *decomposition methods.*

8.2 Methods

8.2.1 Sparsification using compressibility loss

The method starts from a pretrained neural network (input to the method) and calculates the task loss (categorical cross-entropy for image classification, MSE for image compression, etc.) and the compressibility loss on validation set and arranges the weight λ_c on the compressibility loss such that the compressibility loss is m times the task loss. This λ_c is fixed during the entire training and the following loss is minimized during neural network training:

$$L_t = L_s + \lambda_c L_c$$

m is a hyperparameter of the method. Note that λ_c is directly inferred from m so it is not an additional hyperparameter.

The compressibility loss is defined as follows

$$L_c = \frac{|w|}{w} + \gamma_c \frac{w^2}{|w|}$$

where $|w|$ and w are L_1 and L_2 norms of the vector w, respectively, and the vector w is obtained by flattening all the tensors representing the learnable parameters of the neural network.

During this data-dependent transformation γ_c is arranged accordingly such that $\frac{w^2}{|w|} = \frac{1}{3}\frac{|w|}{w}$.

After the data-dependent transformation is completed, the neural network parameters are flattened to a single vector, and pruned simply by setting the parameters that have smaller absolute values than a threshold τ, to zero.

8.2.2 Sparsification using micro-structured pruning

The method starts from a pretrained neural network (input to the method) and aims at changing (pruning) the pretrained parameters in a structured way.

For each network layer (e.g. the k-th layer) the parameters W_k are represented as a general 5-dimensional (5D) tensor of size $c_1^k \times c_2^k \times n_1^k \times n_2^k \times n_3^k$. The input of the layer is a 4-dimensional (4D) tensor A of size $h_1^k \times v_1^k \times d_1^k \times c_1^k$, and the output of the layer is a 4D tensor B of size $h_2^k \times v_2^k \times d_2^k \times c_2^k$. When any of the sizes $c_1^k, c_2^k, n_1^k, n_2^k, n_3^k, h_1^k, v_1^k, d_1^k, h_2^k, v_2^k, d_2^k$ takes the value 1, the corresponding tensor reduces to a lower dimension. h_1^k, v_1^k, d_1^k (h_2^k, v_2^k, d_2^k) are the height, weight and depth of the input tensor A (output tensor B). c_1^k (c_2^k) is the number of input (output) channel. n_1^k, n_2^k, and n_3^k are the size of the convolution kernel corresponding to the height, weight and depth axes, respectively. The 5D parameter tensor is reshaped into a 3D tensor of size ($c_1^{k'}$, $c_2^{k'}$, n^k), where $c_1^{k'} \times c_2^{k'} \times n^k = c_1^k \times c_2^k \times n_1^k \times n_2^k \times n_3^k$. The size of the 3D tensor is defined as $c_1^{k'} = c_1^k$, $c_2^{k'} = c_2^k$, $n^k = n_1^k \times n_2^k \times n_3^k$.

The reshaped 3D parameter tensor is partitioned into super-blocks of size $s_1^k \times s_2^k \times n^k$. Let S_j^k denote a super-block. Each super-block S_j^k is further partitioned into blocks of size $b_1^k \times b_2^k \times b_3^k$. Let B_{jl}^k denote a block in S_j^k, parameters are pruned block-wise inside S_j^k. For each block B_{jl}^k, a prune distortion loss L(B_{jl}^k) can be computed (e.g. as the L_N norm of the absolute value of the parameters in B_{jl}^k). The prune distortion loss L (S_j^k) of the entire super-block S_j^k is computed by averaging L(B_{jl}^k) across all blocks in S_j^k, i.e., L(S_j^k)=$average_{B_{jl}^k}$ L(B_{jl}^k). When selected to be pruned, all parameters in B_{jl}^k will be set to zero.

When c_1^k cannot be fully divided by s_1^k, or c_2^k cannot be fully divided by s_2^k, the super-blocks along the boundary of the corresponding dimension will be smaller. When n^k can not be fully divided by b_3^k, s_1^k cannot be fully divided by b_1^k, or s_2^k cannot be fully divided by b_2^k, the blocks along the boundary of the corresponding dimension will be smaller. That is, $b_1^k \times b_2^k \times b_3^k$ is the maximum size of the blocks, and $s_1^k \times s_2^k \times n^k$ is the maximum size of the super-blocks.

Given the original task loss L_{train} (categorical cross-entropy for image classification, MSE for image compression, etc.), this method iteratively takes the following two steps:

1. The blocks are ranked based on their loss L(B_{jl}^k) in ascending order. Given a pruning ratio p as a hyperparameter, the top p blocks are selected to be pruned. And the parameters in selected blocks are set to be 0. A parameter pruning mask M_k is maintained throughout the training process. M_k has the same shape as W_k, which records whether a corresponding weight coefficient is pruned or not.

2. The parameters which are marked in M_k as being pruned are fixed, the remaining unfixed weight coefficients of W_k are updated through a neural network training process.

The method will output an updated model with the same model structure as the input model, where part or all of the parameters being structurally removed (pruned). The output model can be directly used in the same way as the input model.

8.2.3 Combined pruning and sparsification

8.2.3.1 General

The method consists of three steps. Starting from a pre-trained neural network, pruning ratio p and sparsification ratio q, this method takes the following steps:

1. Analyse the network to identify the parameters suitable for pruning.

2. Remove the neurons with respect to the pruning ratio p.

3. Apply data dependent-based sparsification with regard to the sparsification ratio q. The sparsification method can be any sparsification method which is defined as a parameter reductions method in this document or a similar method.

Given a configuration setup, the steps 1 to 3 can be performed progressively or in one-shot until a target compression ratio is achieved. In step 2, a sparsification operation may also be applied rather than pruning. In step 3, only task loss may also be applied in order to improve the neural network performance.

8.2.3.2 Estimating the importance of parameters for pruning

The neural network parameters are estimated based on a diffusion process over the layers. Example of pruning of convolution filters is provided below, similar formulation applies to other type of layers and to group of layers.

Each convolution layer consists of a parameter tensor $F \in \mathbb{R}^{C_{out} \times K \times K \times C_\mathrm{i}}$, also denoted filter, where C_o is the number of output channels, K is the dimension of the convolution kernel, and C_i is the number of input channels.

Under constant input the redundancy in a layer output can be modelled by the internal redundant information inside the filter. Thus, by considering an ergodic Markov process between the output channels, a graph diffusion is employed to find the redundancy. To this end, given a convolution filter F, a feature matrix $M \in \mathbb{R}^{C_\mathrm{o} \times m}$ is obtained where $m = K \times K \times C_\mathrm{i}$, via tensor reshape.

Following the ergodic Markov chain with each output channel as one state, the probability of reaching a particular state at the equilibrium is $\pi^T = \pi^T P$ where P is the stochastic transition matrix and π is the equilibrium probability of P, corresponding to the left eigenvector $\lambda = 1$. Under equilibrium, the importance can be defined as

$$S = \exp(-\frac{1}{\varphi \pi}),$$

where φ is a smoothing factor, that can be equal to the number of output channels. The elements p_{ij} of the transition matrix P are determined as

$$p_{ij} = \frac{e^{-D(m_i, m_j)}}{\sum_{z=1}^{C_{out}} e^{-D(m_i, m_z)}},$$

where m_i is the *i*-th row of M and $D(\cdot,\cdot)$ is any distance function of preference. The higher value of S will indicate more dissimilarity, importance and salience for output channel in comparison to the other output channels. To prune the filters, after computing S, less salient channels are removed.

8.2.3.3 Data dependent-based sparsification

While any data dependent sparsification from this document can be used, the following data dependent sparsification is employed. A data-dependent approach which consists of the task loss (e.g. categorical cross-entropy for image classification, MSE for image compression, etc.), the compressibility loss and diversity loss. The loss terms are arranged with the weight λ_c on the compressibility loss and the weight λ_d on the diversity loss, respectively. The λ_c and λ_d are fixed during the entire training and the following loss is minimized during neural network training:

$$L_\mathrm{t} = L_\mathrm{s} + \lambda_\mathrm{c} L_\mathrm{c} - \lambda_\mathrm{d} L_\mathrm{d}$$

The compressibility loss is defined as follows

$$L_\mathrm{c} = \frac{|w|}{w} + \gamma_\mathrm{c} \frac{w^2}{|w|}$$

where $|w|$ and w are L_1 and L_2 norms of the vector w, respectively, and the vector w is obtained by flattening all the tensors representing the learnable parameters of the neural network.

The γ_c is chosen such that $\frac{w^2}{|w|} = \frac{1}{3} \frac{|w|}{w}$.

The diversity loss term encourages the diversity between filters at each layer l for all the layers of a network, L, that is

$$L_d = \sum_{l \in L} \mathrm{Div}(W_l)$$

where W_l is a tensor representing the learnable parameters of the l-th layer. The learnable parameters of each i-th filter in the l-th layer are represented as a weight vector w_{li}. The diversity for the l-th layer is then computed as

$$\mathrm{Div}(W_l) = \sum_{i,j} \left(1 - \left|\left\langle w'_{li}, w'_{lj} \right\rangle\right|\right),$$

where $\langle \cdot,\cdot \rangle$ is the dot product of vectors, w'_{li} and w'_{lj} are the normalized w_{li} and w_{lj}, respectively. The diversity term is bounded, $0 \le 1 - \left|\left\langle w'_{li}, w'_{lj} \right\rangle\right| \le 1$. The value close to 0 indicates highly correlated weight vectors and a value near 1 means uncorrelated filters. The total diversity over all the layers is used as the diversity loss.

8.2.4 Parameter unification

The method starts from a pretrained neural network (input to the method) and aims at changing (unifying) the pretrained parameters in a structured way.

For each network layer (e.g. the k-th layer), its parameters W_k is a general 5-dmensional (5D) tensor of size $c_1^k \times c_2^k \times n_1^k \times n_2^k \times n_3^k$. The input of the layer is a 4-dimensional (4D) tensor A of size $h_1^k \times v_1^k \times d_1^k \times c_1^k$, and the output of the layer is a 4D tensor B of size $h_2^k \times v_2^k \times d_2^k \times c_2^k$. When any of the sizes $c_1^k, c_2^k, n_1^k, n_2^k, n_3^k, h_1^k, v_1^k, d_1^k, h_2^k, v_2^k, d_2^k$ takes the value 1, the corresponding tensor reduces to a lower dimension. h_1^k, w_1^k, d_1^k (h_2^k, v_2^k, d_2^k) are the height, parameter and depth of the input tensor A (output tensor B). c_1^k (c_2^k) is the number of input (output) channel. n_1^k, n_2^k, and n_3^k are the size of the convolution kernel corresponding to the height, parameter and depth axes, respectively. The 5D parameter tensor is reshaped into a 3D tensor of size ($c_1^{k\prime}$, $c_2^{k\prime}$, n^k), where $c_1^{k\prime} \times c_2^{k\prime} \times n^k = c_1^k \times c_2^k \times n_1^k \times n_2^k \times n_3^k$. The size of the 3D tensor is defined as $c_1^{k\prime} = c_1^k$, $c_2^{k\prime} = c_2^k, n^k = n_1^k \times n_2^k \times n_3^k$.

The reshaped 3D parameter tensor is partitioned into super-blocks of size $s_1^k \times s_2^k \times n^k$. Let S_j^k denote a super-block. Each super-block S_j^k is further partitioned into blocks of size $b_1^k \times b_2^k \times b_3^k$. Let B_{jl}^k denote a block in S_j^k, parameter unification happens inside S_j^k blocks, there can be different ways to unify parameter coefficients in B_{jl}^k. Given a parameter unifying method, the parameter unifier can unify parameters in B_{jl}^k using the method with an associated unification distortion loss L(B_{jl}^k). The unification distortion loss L(S_j^k) of the entire super-block S_j^k is computed by averaging L(B_{jl}^k) across all blocks in S_j^k, i.e., L(S_j^k)=$average_{B_{jl}^k}$ L(B_{jl}^k). All parameters in B_{jl}^k can be set to have the same absolute value, while keeping the original signs. In such a case, the L_N norm of the absolute of parameters in B_{jl}^k can be used to measure L(B_{jl}^k).

When c_1^k cannot be fully divided by s_1^k, or c_2^k cannot be fully divided by s_2^k, the super-blocks along the boundary of the corresponding dimension will be smaller. When n^k can not be fully divided by b_3^k, s_1^k cannot be fully divided by b_1^k, or s_2^k cannot be fully divided by b_2^k, the blocks along the boundary of the corresponding dimension will be smaller. That is, $b_1^k \times b_2^k \times b_3^k$ is the maximum size of the blocks, and $s_1^k \times s_2^k \times n^k$ is the maximum size of the super-blocks.

Given the original task loss L_s (categorical cross-entropy for image classification, MSE for image compression, etc.), this method iteratively takes the following two steps:

1. The super-blocks are ranked based on their unification loss $L(S_j^k)$ in ascending order. Given a unification ratio u as a hyperparameter, the top u super-blocks are selected to be unified. And the parameter unifier unifies the blocks in the selected super-blocks. A parameter unifying mask M_k is maintained throughout the training process. M_k has the same shape as W_k, which records whether a corresponding parameter coefficient is unified or not.
2. The parameter coefficients which are marked in M_k as being unified are fixed, the remaining unfixed parameters of W_k are updated through a neural network training process.

The method will output an updated model with the same model structure as the input model, where part or all of the parameters being structurally changed (unified). The output model can be directly used in the same way as the input model.

8.2.5 Low rank/low displacement rank for convolutional and fully connected layers

This method aims at reducing the size of parameter tensors, while keeping the accuracy high. It enables using reduced tensors at inference, involving less multiplication and memory load.

The parameter matrices of some fully connected and convolutional layers of pretrained neural networks are approximated as low rank or low displacement rank form. G_k ,H_k are transmitted in the bitstream.

Z_e and Z_f, are chosen to be *f*-circulant operators, expressed as $Z_f = \begin{pmatrix} & f \\ I_{n-1} & \end{pmatrix}$. They are coded in the bitstream using the value of *e*, *f*, as the structure of Z_f is known by the decoder. Finally, the rank and parameters *e* and *f* have to be transmitted in the bitstream.

In the same way, low rank approximations represent an original matrix of parameters as a product:

$$\hat{W}_k = G_k * H_k$$

where G_k is a $m \times r_k$ matrix and H_k is $r_k \times n$ matrix that can be derived from an SVD $\hat{W}_k = U\Sigma V^T$ and retaining the first r_k singular values and vectors.

8.2.6 Batchnorm folding

When a batch-normalization layer follows either a convolutional or fully-connected layer, then the following re-parametrization (denoted batchnorm folding) of the parameters can be applied. It requires that the combination of a convolutional or fully-connected layer with the batch-normalization layer can be expressed as

$$BN(X) = \frac{W * X + b - \mu}{\sqrt{\sigma^2 + \epsilon}} \circ \gamma + \beta$$

where X is the input, $BN(X)$ is the output, W is a weight tensor of the convolutional or fully-connected layer (represented as 2D matrix), b is a bias parameter, and where the remaining parameters are batch-normalization parameters. Note that b, μ, σ^2, γ, and β have the same shape as X and that X is shaped as a transposed vector. Parameter ϵ is a scalar close to zero.

If batchnorm folding is applied, the above equation is expressed as

$$BN(X) = \alpha \circ W * X + \delta$$

where $\alpha = \frac{\gamma}{\sqrt{\sigma^2 + \epsilon}}$ and where $\delta = \frac{(b - \mu) \circ \gamma}{\sqrt{\sigma^2 + \epsilon}} + \beta$.

Parameter α can be present in NNR compressed payloads of type NNR_PT_BLOCK as output variable RecLS and it is quantized using either uniform quantization or dependent scalar quantization.

Parameter δ can be present in NNR compressed payloads of type NNR_PT_BLOCK as output variable RecBias and it is quantized using either uniform quantization or dependent scalar quantization.

Note that the four batchnorm parameters RecBeta, RecGamma, RecMean, and RecVar can be recreated from RecLS and RecBias according to the following equations:

RecBeta = RecBias

RecGamma = RecLS

RecMean = 0

RecVar = $1 - \epsilon$

In case the four batchnorm parameters have been recreated, RecBias is set to 0 and RecLS is set to 1. If RecBias and RecLS are not required, they can simply be ignored.

8.2.7 Local scaling adaptation

This method aims at increasing the capacity of the neural network by introducing a multiplicative scaling factor to each output element of the linear component of a convolutional or fully-connected layer. That is, in the case of fully-connected layers, a unique scaling factor is multiplied to each output neuron before the bias is added. Analogously, at convolutional layers each output feature map is assigned a unique scaling factor which is multiplied to all elements of the feature map, before the bias is added respectively.

This method allows to increase the capacity of the network and thus, compensate for the quantization error induced by quantizing the weight tensors of the convolutional and fully-connected layers.

The scaling factors s can be present in NNR compressed payloads of type NNR_PT_BLOCK as output variable RecLS and they are quantized using either uniform quantization or dependent scalar quantization.

When batchnorm folding is applied together with local scaling adaptation, the scaling factors s are merged with parameter α of the batchnorm folding operation as follows:

$$\alpha' = \alpha \circ s$$

The resulting variable α' can be present in NNR compressed payloads of type NNR_PT_BLOCK as output variable RecLS and it is quantized using either uniform quantization or dependent scalar quantization.

Note that the decoder needs not to be aware of whether RecLS contains only folded batchnorm parameters or only scaling factors or both.

The recommended usage of the scaling factors is to derive and add them after quantization of the weight tensors has been performed. The scaling factors are initialized with the value of 1, and then adapted by means of backpropagation so that the prediction performance of the quantized neural network is increased. Notably, this particular manner of introducing and calculating the scaling factors requires access to data. However, having access to only a small dataset usually suffices for attaining good results with this method, comparable to the size of a typical validation set (approx. 5 % of the training set size).

8.3 Syntax and semantics

8.3.1 Sparsification using compressibility loss

The presence and semantics of syntax elements are specified in Table 15.

Table 15 — Syntax and semantics for sparsification using compressibility loss.

Syntax element	condition	semantics
tensor_dimensions	present	Dimension and shape of original tensors

8.3.2 Sparsification using micro-structured pruning

The presence and semantics of syntax elements are specified in Table 16.

Table 16 — Syntax and semantics for sparsification using miro-structured pruning.

Syntax element	condition	semantics
count_tensor_dimension	present	counter of how many dimensions of reshaped weight tensor
reshaped_tensor_dimensions[]	present	dimensions of reshaped weight tensor
count_super_block_dimension	present	counter of how many dimensions of superblock
super_block_dimensions[]	present	dimensions of superblock
count_block_dimension	present	counter of how many dimensions of block
block_dimensions[]	present	dimensions of block

8.3.3 Combined pruning and sparsification

The presence and semantics of syntax elements are specified in Table 17.

Table 17 — Syntax and semantics for combined pruning and sparsification.

Syntax element and functions	condition	semantics
nnr_rep_type	present	The flag to indicate what type of output is produced
prune_flag	present	The flag to indicate pruning is applied
order_flag	present	The flag to indicate the order of processing of information in row-major or column-major
sparse_flag	present	The flag to indicate sparsification is applied
count_ids	(prune_flag == 1) && (nnr_rep_type == NNR_TPL_ DICT)	The number of elements that are pruned
element_id[]	(prune_flag == 1) && (nnr_rep_type == NNR_TPL_ DICT)	The IDs of the elements that are pruned
count_dims[]	(prune_flag == 1) && (nnr_rep_type == NNR_TPL_ DICT)	The number of dimensions of each pruned element
dim[][]	(prune_flag == 1) && (nnr_rep_type == NNR_TPL_ DICT)	The new dimensions of the pruned elements

Table 17 *(continued)*

Syntax element and functions	condition	semantics
bit_mask()	sparse_flag == 1	A bitmask to indicate which matrix elements are preserved during sparsification. A bit value of 1 shall indicate that the corresponding element is preserved and a bit value of 0 shall indicate that the corresponding element is sparsified
	(prune_flag == 1) && (nnr_rep_type == NNR_TPL_BMSK)	A bitmask to indicate which matrix elements or output channels are preserved during pruning. A bit value of 1 shall indicate that the corresponding element is preserved and a bit value of 0 shall indicate that the corresponding element is pruned

8.3.4 Weight unification

The presence and semantics of syntax elements are specified in Table 18.

Table 18 — Syntax and semantics for weight unification.

Syntax element	condition	semantics
count_tensor_dimension	present	counter of how many dimensions of reshaped weight tensor
reshaped_tensor_dimensions[]	present	dimensions of reshaped weight tensor
count_super_block_dimension	present	counter of how many dimensions of superblock
super_block_dimensions[]	present	dimensions of superblock
count_block_dimension	present	counter of how many dimensions of block
block_dimensions[]	present	dimensions of block

8.3.5 Low rank/low displacement rank for convolutional and fully connected layers

The presence and semantics of syntax elements are specified in Table 19.

Table 19 — Syntax and semantics for low rank/low displacement rank.

Syntax element	condition	semantics
compressed_parameter_types	(compressed_parameter_types && NNR_CPT_DC) != 0	One bit indicating whether decomposition is present
decomposition_rank	present	rank
g_number_of_rows	present	rows of G
tensor_dimensions	present	dimensions of original tensor

8.3.6 Batchnorm folding

The presence and semantics of syntax elements are specified in Table 20.

Table 20 — Syntax and semantics for batchnorm folding.

Syntax element/Variable	condition	semantics
compressed_parameter_types	(compressed_parameter_types && NNR_CPT_BN) != 0	One bit indicating whether batchnorm parameters are present
QpDensity	present	unsigned integer

Table 20 *(continued)*

Syntax element/Variable	condition	semantics
QuantizationParameter	present	integer
qp_value	present	integer
dq_flag	present	flag

8.3.7 Local scaling

The presence and semantics of syntax elements are specified in Table 21.

Table 21 — Syntax and semantics for local scaling.

Syntax element/Variable	condition	semantics
compressed_parameter_types	(compressed_parameter_types && NNR_CPT_LS) != 0	One bit indicating whether a local scaling parameter is present
QpDensity	present	unsigned integer
QuantizationParameter	present	integer
qp_value	present	integer
dq_flag	present	flag

9 Parameter quantization

9.1 Methods

9.1.1 Uniform quantization method

Uniform quantization is applied to the parameter tensors using a fixed step size represented by parameters mps_qp_density (or lps_qp_density, if present) and qp_value according to the specification in subclause 7.3.3 and a flag, denoted as dq_flag, equal to zero. The reconstructed values in the decoded tensor are integer multiples of the step size.

9.1.2 Codebook-based method

The parameter tensors are represented as a codebook and tensors of indices, the latter having the same shape as the original tensors. The size of the codebook is chosen at the encoder and is transmitted as a metadata parameter. The indices have integer values, they will be further entropy coded. The codebook is composed of integer values that are strictly monotonically increasing.

The reconstructed integer tensors are the values of codebook elements referred to by their index value and the reconstructed tensors are derived by multiplying the reconstructed integer tensors with a step size that is derived from parameters mps_qp_density (or lps_qp_density, if present) and qp_value.

9.1.3 Dependent scalar quantization method

Dependent scalar quantization is applied to the parameter tensors using a fixed stepsize represented by parameters mps_qp_density (or lps_qp_density, if present) and qp_value according to the specification in subclause 7.3.3 and a state transition table of size 8, whenever a flag, denoted as dq_flag, is equal to one. The reconstructed values in the decoded tensor are integer multiples of the step size.

9.2 Syntax and semantics

9.2.1 Uniform quantization method

The presence and semantics of syntax elements are specified in Table 22.

Table 22 — Syntax and semantics for uniform quantization method.

Syntax element/Variable	condition	semantics
QpDensity	present	unsigned integer
QuantizationParameter	present	integer
qp_value	present	integer
dq_flag	dq_flag == 0	flag

9.2.2 Codebook-based method

The presence and semantics of syntax elements are specified in Table 23.

Table 23 — Syntax and semantics for codebook-based method.

Syntax element/Variable	condition	semantics
QpDensity	present	unsigned integer
QuantizationParameter	present	integer
qp_value	present	integer
codebook_egk	present	unsigned integer
codebook_size	present	unsigned integer
codebook_centre_offset	present	integer
codebook_zero_value	present	integer
codebook_delta_left	present	unsigned integer, multiple instances thereof
codebook_delta_right	present	unsigned integer, multiple instances thereof

9.2.3 Dependent scalar quantization method

The presence and semantics of syntax elements are specified in Table 24.

Table 24 — Syntax and semantics for dependent scalar quantization method.

Syntax element	condition	semantics
QpDensity	present	unsigned integer
QuantizationParameter	present	integer
qp_value	present	integer
dq_flag	dq_flag == 1	flag

10 Entropy coding

10.1 Methods

10.1.1 DeepCABAC

10.1.1.1 Binarization

The encoding method scans the parameter tensor in a manner as defined by function TensorIndex(). Each quantized parameter level is encoded according to the following procedure employing an integer parameter 'maxNumNoRemMinus1':

In the first step, a binary syntax element sig_flag is encoded for the quantized parameter level, which specifies whether the corresponding level is equal to zero. If the sig_flag is equal to one, a further binary syntax element sign_flag is encoded. The bin indicates if the current parameter level is positive or negative. Next, a unary sequence of bins is encoded, followed by a fixed length sequence as follows:

A variable k is initialized with zero and X is initialized with 1 << k. A syntax element abs_level_greater_x/x2 is encoded, which indicates, that the absolute value of the quantized parameter level is greater than x. If abs_level_greater_x/x2 is equal to 1 and if x is greater than maxNumNoRemMinus1, the variable k is increased by 1. Afterwards, 1 << k is added to x and a further abs_level_greater_x/x2 is encoded. This procedure is continued until an abs_level_greater_x/x2 is equal to 0. Now, it is clear that X must be one of the values (x, x − 1, ... X − (1 << k) + 1). A code of length k is encoded, which points to the values in the list which is absolute quantized parameter level.

10.1.1.2 Context modelling

Context modelling corresponds to associating the three type of flags sig_flag, sign_flag, and abs_level_greater_x/x2 with context models. In this way, flags with similar statistical behavior should be associated with the same context model so that the probability estimator (inside of the context model) can adapt to the underlying statistics.

The context modelling of the presented approach is as follows:

Twenty-four context models are distinguished for the sig_flag, depending on the state value and whether the neighbouring quantized parameter level to the left is zero, smaller, or larger than zero.

If dq_flag is 0, only the first three context models are used.

Three other context models are distinguished for the sign_flag depending on whether the neighbouring quantized parameter level to the left is zero, smaller, or larger than zero.

For the abs_level_greater_x/x2 flags, each x uses either one or two separate context models. If x <= maxNumNoRemMinus1, two context models are distinguished depending on the sign_flag. If x > maxNumNoRemMinus1, only one context model is used.

10.2 Syntax and semantics

10.2.1 DeepCABAC syntax

10.2.1.1 General

This subclause specifies the entropy coding syntax as used by the decoding process of clause 7.

10.2.1.2 DeepCABAC termination syntax

terminate_cabac( ) {	**Descriptor**
terminating_one_bit	at(v)
while(!byte_aligned())	
nesting_zero_bit	f(1)
}	

terminating_one_bit specifies a terminating bit equal to 1.

nesting_zero_bit is one bit set to 0.

10.2.1.3 Quantization parameter syntax

quant_param(qpDensity) {	**Descriptor**

qp_value	iae(6 + qpDensity)
}	

qp_value is the quantization parameter.

10.2.1.4 Quantized tensor syntax

quant_tensor(dimensions, maxNumNoRemMinus1, entryPointOffset) {	**Descriptor**
stateId = 0	
bitPointer = get_bit_pointer( )	
lastOffset = 0	
for(i = 0; i < Prod(dimensions); i++) {	
idx = TensorIndex(dimensions, i, scan_order)	
if(entryPointOffset != -1 && GetEntryPointIdx(dimensions, i, scan_order) != -1 && scan_order > 0) {	
IvlCurrRange = 256	
j = entryPointOffset + GetEntryPointIdx(dimensions, i, scan_order)	
IvlOffset = cabac_offset_list[j]	
if(dq_flag)	
stateId = dq_state_list[j]	
set_bit_pointer(bitPointer + lastOffset + BitOffsetList[j])	
lastOffset = BitOffsetList[j]	
init_prob_est_param( )	
}	
int_param(idx, maxNumNoRemMinus1, stateId)	10.2.1.5
if(dq_flag) {	
nextSt = StateTransTab[stateId][QuantParam[idx] & 1]	
if(QuantParam[idx] != 0) {	
QuantParam[idx] = QuantParam[idx] << 1	
if(QuantParam[idx] < 0)	
QuantParam[idx] += stateId & 1	
else	
QuantParam[idx] += − (stateId & 1)	
}	
stateId = nextSt	
}	
}	
}	

init_prob_est_param() invokes the initialization process specified in subclause 10.3.2.2.

The 2D integer array StateTransTab[][] specifies the state transition table for dependent scalar quantization and is as follows:

StateTransTab[][] = { {0, 2}, {7, 5}, {1, 3}, {6, 4}, {2, 0}, {5, 7}, {3, 1}, {4, 6} }

10.2.1.5 Quantized parameter syntax

int_param(i, maxNumNoRemMinus1, stateId) {	**Descriptor**
QuantParam[i] = 0	
sig_flag	ae(v)
if(sig_flag) {	
QuantParam[i]++	
sign_flag	ae(v)
j = −1	
do {	
j++	
abs_level_greater_x[j]	ae(v)
QuantParam[i] += abs_level_greater_x[j]	
} while(abs_level_greater_x[j] == 1 && j < maxNumNoRemMinus1)	
if(abs_level_greater_x[j] == 1) {	
RemBits = 0	
j = −1	
do {	
j++	
abs_level_greater_x2[j]	ae(v)
if(abs_level_greater_x2[j]) {	
QuantParam[i] += 1 << RemBits	
RemBits++	
}	
} while(abs_level_greater_x2[j] && j < 30)	
abs_remainder	uae(RemBits)
QuantParam[i] += abs_remainder	
}	
QuantParam[i] = sign_flag ? −QuantParam[i] : QuantParam[i]	
}	
}	

sig_flag specifies whether the quantized weight QuantParam[i] is nonzero. A sig_flag equal to 0 indicates that QuantParam[i] is zero.

sign_flag specifies whether the quantized weight QuantParam[i] is positive or negative. A sign_flag equal to 1 indicates that QuantParam[i] is negative.

abs_level_greater_x[j] indicates whether the absolute level of QuantParam[i] is greater j + 1.

abs_level_greater_x2[j] comprises the unary part of the exponential Golomb remainder.

abs_remainder indicates a fixed length remainder.

10.2.1.6 Shift parameter indices syntax

shift_parameter_ids(maxNumNoRemMinus1) {	**Descriptor**
for(i = 0; i < (dq_flag ? 24:3); i++) {	
shift_idx(i, ShiftParameterIdsSigFlag)	10.2.1.7
}	
for(i = 0; i < 3; i++) {	
shift_idx(i, ShiftParameterIdsSignFlag)	10.2.1.7
}	
for(i = 0; i < 2*(maxNumNoRemMinus1+1); i++) {	
shift_idx(i, ShiftParameterIdsAbsGrX)	10.2.1.7
}	
for(i = 0; i < 31; i++) {	
shift_idx(i, ShiftParameterIdsAbsGrX2)	10.2.1.7
}	
}	

10.2.1.7 Shift parameter syntax

shift_idx(ctxId, shiftParameterIds) {	**Descriptor**
shiftParameterIds[ctxId] = 0	
shift_idx_minus_1_present_flag	ae(v)
if(shift_idx_minus_1_present_flag) {	
shift_idx_minus_1	uae(3)
shiftParameterIds[ctxId] += shift_idx_minus_1 + 1	
}	
}	

shift_idx_minus_1_present_flag specifies whether the shift parameter index shiftParameterIds[ctxId] is present. A shift_idx_minus_1_present_flag equal to zero indicates that shiftParameterIds[ctxId] is zero.

shift_idx_minus_1 specifies the absolute value of the shift parameter index shiftParameteIds[ctxId] minus one. The shift parameter index is shiftParameIds[ctxId] = shift_idx_minus_1 + 1

10.3 Entropy decoding process

10.3.1 General

Inputs to this process are a request for a value of a syntax element and values of prior parsed syntax elements.

Output of this process is the value of the syntax element.

The parsing of syntax elements proceeds as follows:

For each requested value of a syntax element a binarization is derived as specified in subclause 10.3.3.

The binarization for the syntax element and the sequence of parsed bins determines the decoding process flow as described in subclause 10.3.4.

10.3.2 Initialization process

10.3.2.1 General

Outputs of this process are initialized DeepCABAC internal variables.

The context variables of the arithmetic decoding engine are initialized as follows:

The initialization process for context variables is invoked as specified in subclause 10.3.2.3.

The decoding engine registers IvlCurrRange and IvlOffset both in 16 bit register precision are initialized by invoking the initialization process for the arithmetic decoding engine as specified in subclause 10.3.2.4.

10.3.2.2 Initialization process for probability estimation parameters

Outputs of this process are the initialized probability estimation parameters shift0, shift1, pStateIdx0, and pStateIdx1 for each context model of syntax elements sig_flag, sign_flag, abs_level_greater_x, and abs_level_greater_x2.

The 2D array CtxParameterList [][] is initialized as follows:

CtxParameterList[][] = { {1, 4, 0, 0}, {1, 4, -41, -654}, {1, 4, 95, 1519}, {0, 5, 0, 0}, {2, 6, 30, 482}, {2, 6, 95, 1519}, {2, 6, -21, -337}, {3, 5, 0, 0}, {3, 5, 30, 482}}

If dq_flag is equal to 1, for each of the 24 context models of syntax element sig_flag, the associated context parameter shift0 is set to CtxParameterList[setId][0], shift1 is set to CtxParameterList[setId][1], pStateIdx0 is set to CtxParameterList[setId][2], and pStateIdx1 is set to CtxParameterList[setId][3], where i is the index of the context model and where setId is equal to ShiftParameterIdsSigFlag[i].

Otherwise, (dq_flag == 0), for each of the first 2 context models of syntax element sig_flag, the associated context parameter shift0 is set to CtxParameterList[setId][0], shift1 is set to CtxParameterList[setId][1], pStateIdx0 is set to CtxParameterList[setId][2], and pStateIdx1 is set to CtxParameterList[setId][3], where i is the index of the context model and where setId is equal to ShiftParameterIdsSigFlag[i].

For each of the 3 context models of syntax element sign_flag, the associated context parameter shift0 is set to CtxParameterList[setId][0], shift1 is set to CtxParameterList[setId][1], pStateIdx0 is set to CtxParameterList[setId][2], and pStateIdx1 is set to CtxParameterList[setId][3], where i is the index of the context model and where setId is equal to ShiftParameterIdsSignFlag[i].

For each of the 2 * (cabac_unary_length_minus1 + 1) context models of syntax element abs_level_greater_x, the associated context parameter shift0 is set to CtxParameterList[setId][0], shift1 is set to CtxParameterList[setId][1], pStateIdx0 is set to CtxParameterList[setId][2], and pStateIdx1 is set to CtxParameterList[setId][3], where i is the index of the context model and where setId is equal to ShiftParameterIdsAbsGrX[i].

For each of the 31 context models of syntax element abs_level_greater_x2, the associated context parameter shift0 is set to CtxParameterList[setId][0], shift1 is set to CtxParameterList[setId][1], pStateIdx0 is set to CtxParameterList[setId][2], and pStateIdx1 is set to CtxParameterList[setId][3], where i is the index of the context model and where setId is equal to ShiftParameterIdsAbsGrX2[i].

10.3.2.3 Initialization process for context variables

Outputs of this process are the initialized DeepCABAC context variables distinguished by the associated syntax element and by ctxIdx.