

**ISO/IEC JTC 1/SC 29/WG 11 N19225**

**ISO/IEC JTC 1/SC 29/WG 11**

**Coding of moving pictures and audio**

**Convenorship: UNI (Italy)**

**Document type: Approved WG 11 document**

**Title: Working Draft 4 of Compression of neural networks for multimedia content description and analysis**

**Status: Approved**

**Date of document: 2020-05-15**

**Source: Video**

**Expected action:**

**No. of pages: 48**

**Email of convenor: leonardo@chiariglione.org**

**Committee URL: mpeg.chiariglione.org**

**INTERNATIONAL ORGANIZATION FOR STANDARDIZATION**

**ORGANISATION INTERNATIONALE DE NORMALISATION**

**ISO/IEC JTC1/SC29/WG11**

**CODING OF MOVING PICTURES AND AUDIO**

**ISO/IEC JTC1/SC29/WG11/N19225**

**April 2020, Alpbach, AT**

**Title: Working Draft 4 of Compression of neural networks for multimedia content description and analysis**

**Source Video Subgroup**

**Status: Approved**

# 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.

This document is the second working draft of compression of neural networks for multimedia content description and analysis. It contains a set of adopted compression tools and baseline compression tools aligned with the release of the test model. The inclusion of compression tools in this WD does at this stage not imply adoption of the respective technology.

# Scope

The scope of this standard is to specify a compressed representation of the parameters/weights of a trained network, complementing the description of the network topology in existing (exchange) formats for neural networks. The standard is specified as a toolbox of compression methods, specifying (where applicable) the resulting elements of the compressed bitstream.

This standard does not aim to define a custom exchange format, but to propose compressed representations that can be added to existing formats. The standard thus defines a high-level syntax that specifies required metadata elements and the semantics of components of the file. In cases where the structure of binary data is to be specified (e.g., sparse matrices) this specification will also provide the actual bitstream syntax of the respective block. Annexes to the standard specify the implementation in a particular target format.

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

# Overview

## Approach

Considering the fact that compression of neural networks is likely to have a hardware dependent and hardware independent component, the standard is designed as a toolbox of compression technologies.

Some of these technologies require specific representations in an exchange format (i.e., sparse representations, adaptive quantisation), and thus a normative specification for representing outputs of these technologies is defined. Others do not at all materialise in a serialised representation (e.g. pruning), and these technologies form an informative part of the standard. However, also for the latter ones it may be useful for some applications to include metadata describing which operations have been applied to the original model.

In order to apply the standard, any non-empty subset of compression tools may be used. The definition of profiles that specify the set of tools supported by a particular implementation can be considered.

## Considerations on implementation

The compression tools for structure pruning produce a modified model as an output, without requiring new elements in the representation (this includes matrix decomposition methods, if the output can be represented as a regular model). They can be specified independent of a particular model representation, however, the implementation needs to be provided for a particular neural network (exchange) format. This step is lossy[[1]](#footnote-1), and there is no decoding step for these methods.

Weight sparsification, quantisation, matrix approximation, and predictive coding methods are defined to work on a set of tensors extracted from the source model. The output is again a set of tensors, that contain the encoded bitstream, and any additional data where applicable. Together these outputs must contain all information to reconstruct the input tensors. The results can then be inserted into a model representation (as defined in the respective annex). This step is typically lossy, and decoding follows the same pattern, implementing the reverse operation. The actual operation of encoding/decoding the set of tensors can be implemented in a way that is agnostic of the neural network representation or framework.

Entropy coding follows the same pattern as the previous group of methods, with the exception that this step is required to be lossless.

# Conventions

## General

This clause 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.

## 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 the IEEE 754-1985 specification.

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.

## 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

*xy* 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.

 Used to denote division in mathematical equations where no truncation or rounding is intended.

 The summation 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* ≥ 0 and *y* > 0.

## 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.

## 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.

## 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.

## 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).

## 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.

## 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

Min( x, y ) =

Max( x, y ) = 

## 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) |
| xy |
| "x \* y", "x / y", "x  y""", "x % y" |
| "x + y", "x − y" (as a two-argument operator), "" |
| "x << y", "x >> y" |
| "x < y", "x ≤ y", "x > y", "x ≥ y" |
| "x = = y", "x != y" |
| "x & y" |
| "x | y" |
| "x && y" |
| "x | | y" |
| "x ? y : z" |
| "x..y" |
| "x = y", "x += y", "x −= y" |

## 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 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 clause 7.3 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 4.2) 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 syx. 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.

# Decoding process

A decoder that complies with this specification shall take an NNR bitstream, as specified in clause 7.3, as input and

* generate decompressed data which complies with an NNR decompressed data format (as defined in Table 2) or
* generate ASCII or compressed data outputs as indicated by using the NNR\_TPL and NNR\_QNT NNR unit payloads (as described in Clause 7.3.4)

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 subclauses and annexes of this specification.

Note: For example, a decoder may 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 specification 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 specification and left to implementation.

## NNR decompressed data formats

Table 2 specifies NNR decompressed data formats that result after decompressing NNR compressed data units.

Table 2. NNR decompressed data formats

|  |  |  |
| --- | --- | --- |
| **Parameter identifier** | **Parameter description** | nnr\_decompressed\_data\_format |
| TENSOR\_INT32 | Tensor of integer values used for representing tensor-shaped signed integer parameters of the model | 0 |
| TENSOR\_FLOAT32 | Tensor of float32 values used for representing tensor-shaped float32 parameters of the model | 1 |
| TBD | Specify all parameter formats required for bitstream format NNR\_FULL. |  |
|  |  |  |

## Decoding methods

This clause specifies the decoding methods of this standard in table Table 3. Each decoding method corresponds to a bitstream format conforming to this standard.

Table 3. NNR compressed data payload types

|  |  |  |  |
| --- | --- | --- | --- |
| **Payload identifier** | **Description** | nnr\_compressed\_data\_unit\_payload\_type | **Clause** |
| NNR\_PT\_INT32 | 32 bit integer parameter tensor | 0 | 5.3.2 |
| NNR\_PT\_FLOAT32 | float32 parameter tensor | 1 | 5.3.3 |
| NNR\_PT\_CB\_FLOAT32 | float32 parameter tensor using a codebook | 2 | 5.3.4 |
| NNR\_PT\_RAW\_FLOAT32 | uncompressed float32 parameter tensor | 3 | 5.2.4 |

### Decoding method for NNR compressed payloads of type NNR\_PT\_INT32

Input to this process are:

* One or more NNR compressed data units which are marked to be decompressed together and nnr\_compressed\_data\_unit\_payload\_type fields are set as NNR\_PT\_INT32.
* a syntax element tensor\_dimensions which is a list of integer values specifying the dimensions of the parameter tensor.
* a syntax element cabac\_unary\_length.

Output of this process is a variable RecParam of type TENSOR\_INT32 as specified in Table 2. The dimensions of RecParam are equal to tensor\_dimensions.

The arithmetic coding engine and context models are initialized as specified in clause 11.2.2.

A syntax unit dq\_flag() according to subclause 11.1.2.3 is decoded from the bitstream.

A syntax unit quant\_tensor\_term( tensor\_dimensions, cabac\_unary\_length ) according to subclause 11.1.2.1 is decoded from the bitstream and RecParam is set equal to QuantParam.

NOTE – Decoding of a bitstream conforming to method NNR\_PT\_INT32 shall only produce values for RecParam that can be represented as 32 bit integer value in two’s complement representation.

### Decoding method for NNR compressed payloads of type NNR\_PT\_FLOAT32

Input to this process are:

* One or more NNR compressed data units which are marked to be decompressed together and their nnr\_compressed\_data\_unit\_payload\_type fields are set as NNR\_PT\_FLOAT32
* a syntax element tensor\_dimensions which is a list of integer values specifying the dimensions of the parameter tensor.
* a syntax element cabac\_unary\_length.
* a syntax element qp\_density.
* a syntax element quantization\_parameter.

Output of this process is a variable RecParam of type TENSOR\_FLOAT32 as specified in Table 2. The dimensions of RecParam are equal to tensor\_dimensions.

The arithmetic coding engine and context models are initialized as specified in subclause 11.2.2.

A syntax unit quant\_param( qp\_density ) according to subclause 11.1.2.2 is decoded from the bitstream.

A syntax unit dq\_flag() according to subclause 11.1.2.3 is decoded from the bitstream.

A syntax unit quant\_tensor\_term( tensor\_dimensions, cabac\_unary\_length ) according to subclause 11.1.2.1 is decoded from the bitstream and a variable stepSize is derived as follows:

mul = (1 << qp\_density) + ( (qp + quantization\_parameter) & ( ( 1 << qp\_density ) – 1 ) )

shift = (qp + quantization\_parameter) >> qp\_density

stepSize = mul \* 2shift – qpDensity

RecParam is set as follows:

RecParam = QuantParam \* stepSize

NOTE – Following from the above calculations, RecParam can always be represented as binary fraction. Decoding of a bitstream conforming to method NNR\_PT\_FLOAT32 shall only produce values for RecParam that can be represented as float32 without loss of precision.

### Decoding method for NNR compressed payloads of type of type NNR\_PT\_CB\_FLOAT32

Inputs to this process are:

* One or more NNR compressed data units which are marked to be decompressed together and their nnr\_compressed\_data\_unit\_payload\_type NNR\_PT\_CB\_FLOAT32
* a syntax element tensor\_dimensions which is a list of integer values specifying the dimensions of the parameter tensor.
* a syntax element cabac\_unary\_length.
* a syntax element codebook\_zero\_offset which is an integer
* a syntax element codebook which is a list of float32 values

Output of this process is a variable RecParam of type TENSOR\_FLOAT32 as specified in Table 2. The dimensions of RecParam are equal to tensor\_dimensions.

The arithmetic coding engine and context models are initialized as specified in subclause 11.2.2.

A syntax unit quant\_tensor\_term( tensor\_dimensions, cabac\_unary\_length ) according to subclause 11.1.2.1 is decoded from the bitstream and RecParam is set as follows:

for( i = 0; i < Prod( tensor\_dimensions ); i++ ) {  
 idx = TensorIndex( tensor\_dimensions, i )

RecParam[idx] = codebook[ QuantParam[idx] + codebook\_zero\_offset ]

}

### Decoding method for NNR compressed payloads of type of type NNR\_PT\_RAW\_FLOAT32

Inputs to this process is:

* A syntax element raw\_float32\_parameter.
* a syntax element tensor\_dimensions which is a list of integer values specifying the dimensions of the parameter tensor.

Output of this process is a variable RecParam of type TENSOR\_FLOAT32 as specified in Table 2. The dimensions of RecParam are equal to tensor\_dimensions.

RecParam is set equal to raw\_float32\_parameter.

# Syntax and semantics

## 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 4 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 4. Examples of the syntax specification format

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

## 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.

## 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.

decode\_bit() decodes the next bit from the bitstream using either the arithmetic decoding engine (subclause 13.2.4) or read\_bits( 1 ), as determined by the decoding configuration.

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

Prod( array\_name[] ) returns the product of all elements of array array\_name[].

TensorIndex( tensor\_dimensions[], i ) returns an array with the same number of dimensions as tensor\_dimensions[] where the elements of the array are set to integer values so that the array can be used as an index pointing to the i-th element in row-major scan order of a tensor with dimensions tensor\_dimensions[].

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 11.2.4.3.2.
* ae (t) : arithmetic entropy-coded termination syntax. The parsing process for this data type is specified in subclause 11.2.4.3.5.
* iae(n): signed integer using n arithmetic entropy-coded bits using the bypass mode of CABAC as specified in subclause 11.2.4.3.4. 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 CABAC as specified in subclause 11.2.4.3.4. The read bypass bins are interpreted as a binary representation of an unsigned integer with most significant bit written first.
* 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.
* flt(n): Floating point value using n bits where n may be 16, 32 or 64 in little-endian byte order.
* st(v): null-terminated string encoded as UTF-8 characters as specified in 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) and flt(n) syntax descriptors are only used in this document when the current position in the bitstream is a byte-aligned position.

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.

## Semantics

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

The following definitions apply to the semantics specification.

**unspecified**: A term that may be used to specify some values of a particular *syntax element* to indicate that the values have no specified meaning in this Specification and will not have a specified meaning in the future as an integral part of future versions of this Specification.

**reserved**: A term that may be used to specify that some values of a particular *syntax element* are for future use by ITU-T | ISO/IEC and shall not be used in *bitstreams* conforming to this version of this Specification, but may be used in bitstreams conforming to future extensions of this Specification by ITU‑T | ISO/IEC.]

**nnr\_reserved\_zero\_2bits**, when present, shall be equal to 0 in bitstreams conforming to this version of this Specification. 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 Specification. 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 Specification. 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 Specification. 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.

# High-level syntax definition

## Definition of general bitstream syntax elements

### NNR Unit

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

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

An NNR unit consists of the following data elements:

* **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.

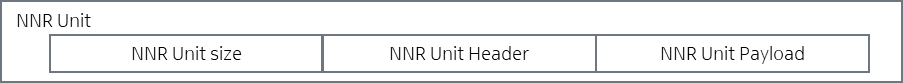


Figure 7‑1 NNR Unit data structure

### 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.

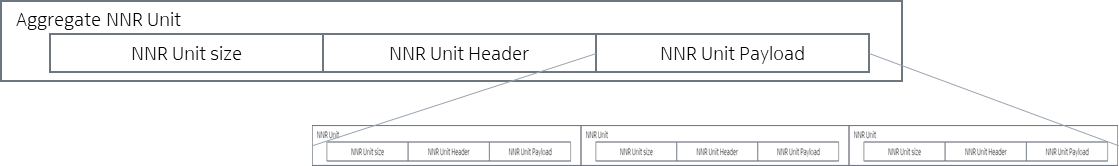


Figure 7‑2 Aggregate NNR unit data structure

## NNR bitstream

### General

NNR bitstream is composed of a sequence of NNR Units. The first NNR unit in an NNR bitstream shall be an NNR start unit (i.e. NNR unit of type NNR\_STR).

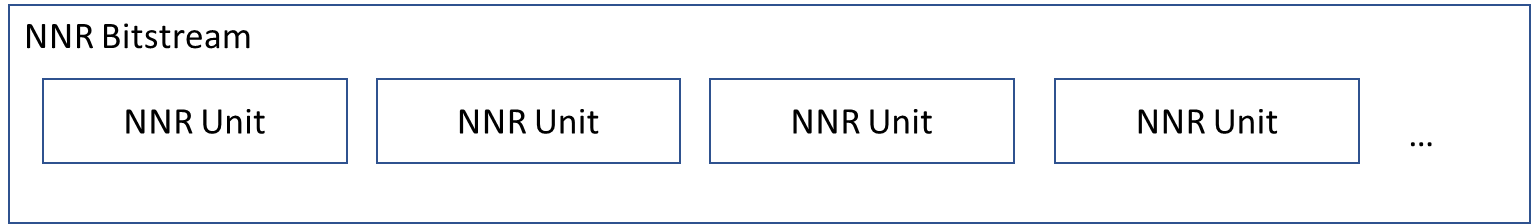


Figure 7‑3 NNR bitstream data structure

### NNR unit syntax

|  |  |
| --- | --- |
| nnr\_unit( numBytesInNNRUnit ) { | **Descriptor** |
| nnr\_unit\_size( ) |  |
| nnr\_unit\_header( ) |  |
| nnr\_unit\_payload( ) |  |
| } |  |

### NNR unit size syntax

|  |  |
| --- | --- |
| nnr\_unit\_size( ) { | **Descriptor** |
| **nnr\_unit\_size\_flag** | u(1) |
| **nnr\_unit\_size** | u(15 + nnr\_unit\_size\_flag\*16) |
| } |  |

### NNR unit header syntax

|  |  |
| --- | --- |
| nnr\_unit\_header( ) { | **Descriptor** |
| **nnr\_unit\_type** | u(8) |
| partial\_data\_counter | u(8) |
| independently\_decodable\_flag | u(1) |
| reserved | u(7) |
| 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\_quanization\_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( ) |  |
| } |  |

#### NNR start unit header syntax

|  |  |
| --- | --- |
| nnr\_start\_unit\_header( ) { | Descriptor |
|  |  |
| } |  |

[Ed(EA): as agreed in MPEG 129, this is a placeholder for this syntax element. Further contributions are needed for further develop]

#### NNR model parameter set unit header syntax

|  |  |
| --- | --- |
| nnr\_model\_parameter\_set\_header( ) { | Descriptor |
|  |  |
| } |  |

[Ed(EA): as agreed in MPEG 129, this is a placeholder for this syntax element. Further contributions are needed for further develop]

#### NNR layer parameter set unit header syntax

|  |  |
| --- | --- |
| nnr\_layer\_parameter\_set\_header( ) { | Descriptor |
|  |  |
| } |  |

[Ed(EA): as agreed in MPEG 129, this is a placeholder for this syntax element. Further contributions are needed for further develop]

#### NNR topology unit header syntax

|  |  |
| --- | --- |
| nnr\_topology\_unit\_header( ) { | **Descriptor** |
| **topology\_storage\_format** | u(8) |
| compressed\_topology\_flag | u(1) |
| if (compressed\_topology\_flag) { |  |
| compression\_format | u(7) |
| } |  |
| else { |  |
| byte\_alignment() |  |
| } |  |
| } |  |

#### NNR quantization unit header syntax

|  |  |
| --- | --- |
| nnr\_quantization\_unit\_header( ) { | **Descriptor** |
| **quantization\_storage\_format** | u(8) |
| quantization\_compressed\_flag | u(1) |
| if (quantization\_compressed\_flag) { |  |
| compression\_format | u(7) |
| } |  |
| else { |  |
| byte\_alignment() |  |
| } |  |
|  |  |
| } |  |

#### 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() |  |
| else |  |
| **ref\_id** | st(v) |
| if (nnr\_compressed\_data\_unit\_payload\_type = =  NNR\_PT\_CB\_FLOAT32) { |  |
| **codebook\_zero\_offset** | u(8) |
| **codebook\_size** | u(16) |
| for( j = 0 ; j < codebook\_size; j++ ) { |  |
| **codebook**[j] | flt(32) |
| } |  |
| } |  |
| 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) |
| if (tensor\_dimensions\_flag = = 1) |  |
| tensor\_dimensions() |  |
| If (cabac\_unary\_length\_flag = = 1) |  |
| **cabac\_unary\_length** | u(8) |
| } |  |
| byte\_alignment() |  |
| } |  |

tensor\_dimensions() is defined as follows:

|  |  |
| --- | --- |
|  | **Descriptor** |
| tensor\_dimensions(){ |  |
| **count\_tensor\_dimensions** | u(8) |
| for(j = 0; j < count\_tensor\_dimensions; j++){ |  |
| **tensor\_dimensions**[j] | u(16)[] |
| } |  |
| } |  |

topology\_elements\_ids\_list() is defined as follows:

|  |  |
| --- | --- |
| topology\_elements\_ids\_list () { | **Descriptor** |
| count\_topology\_elements | u(8) |
| for(j = 0; j < count\_topology\_elements; j++ ) { |  |
| ref\_ids[j] | st(v) |
| } |  |
| } |  |

#### 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(8) |
| **nnr\_unit\_entry\_point[ i ]** | u(16) |
| **}** |  |
| if ( nnr\_aggregate\_unit\_type = = NNR\_AGG\_XYZ\_1 || nnr\_aggregate\_unit\_type = =  NNR\_AGG\_XYZ\_2 ) { |  |
| for(i = 0; i < num\_of\_nnr\_units\_minus2 + 2; i++) { |  |
| **quant\_bitdepth[i]** | u(5) |
| if(ctu\_partition\_flag){ |  |
| **ctu\_scan\_order[i]** | u(1) |
| **nnr\_reserved\_zero\_2bits** | u(2) |
| }else{ |  |
| **nnr\_reserved\_zero\_3bits** | u(3) |
| } |  |
| } |  |
| } |  |
| } |  |

[Ed(EA): Based on agreement in MPEG 129 on m51611 and m52352, NNR\_AGG\_XYZ\_1 and NNR\_AGG\_XYZ\_2 are defined as placeholders. Proper type names require further contribution for definition and update nnr\_aggregate\_unit\_type table in section 7.4.2.3.7]

### NNR unit payload syntax

|  |  |
| --- | --- |
| nnr\_unit\_payload() { | **Descriptor** |
| if( nnr\_unit\_type = = NNR\_MPS ) |  |
| nnr\_model\_parameter\_set\_payload( ) |  |
| if(  nnr\_unit\_type = = NNR\_LPS ) |  |
| nnr\_layer\_parameter\_set\_payload( ) |  |
| if(  nnr\_unit\_type = = NNR\_TPL ) |  |
| nnr\_topology\_unit\_payload( ) |  |
| if(  nnr\_unit\_type = = NNR\_QNT ) |  |
| nnr\_quanization\_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( ) |  |
| } |  |

#### NNR start unit payload syntax

#### NNR model parameter set payload syntax

|  |  |
| --- | --- |
| nnr\_model\_parameter\_set\_payload( ) { | **Descriptor** |
| topology\_carriage\_flag | u(1) |
| sparsification\_flag | u(1) |
| quantization\_method\_flags | u(6) |
| if ((quantization\_method\_flags & NNR\_QSU) = = NNR\_QSU) { |  |
| qp\_density | u(3) |
| quantization\_parameter | i(13) |
| } |  |
| If (sparsification\_flag = = 1) { |  |
| sparsification\_performance\_map() |  |
| } |  |
| **ctu\_partition\_flag** | u(1) |
| if(ctu\_partition\_flag){ |  |
| **max\_ctu\_dim\_flag** | u(2) |
| **nnr\_reserved\_zero\_5bits** | u(5) |
| }else{ |  |
| **nnr\_reserved\_zero\_7bits** | u(7) |
| } |  |
| } |  |

Sparsification\_performance\_map() is defined as follows:

|  |  |
| --- | --- |
| **sparsification\_performance\_map () {** | **Descriptor** |
| count\_thresholds | u(8) |
| for (i = 0; i < (count\_thresholds-1); i++ ) { |  |
| sparsification\_threshold | flt(32) |
| non\_zero\_ratio | flt(32) |
| nn\_accuracy | flt(32) |
| count\_classes | u(8) |
| for (j = 0; j < (count\_classes-1); j++ ) { |  |
| nn\_class\_accuracy | flt(32) |
| } |  |
| } |  |
| } |  |

#### NNR layer parameter set unit payload syntax

|  |  |
| --- | --- |
| nnr\_layer\_parameter\_set\_unit\_payload() { | **Descriptor** |
| independently\_decodable\_flag | u(1) |
| sparsification\_flag | u(1) |
| quantization\_method\_flags | u(6) |
| If ((quantization\_method\_flags & NNR\_QSU) = = NNR\_QSU) { |  |
| quantization\_step\_size | u(8) |
| } |  |
| If ((quantization\_method\_flags & NNR\_QCB) = = NNR\_QCB) { |  |
| quantization\_map() |  |
| } |  |
| If (sparsification\_flag = = 1) { |  |
| sparsification\_performance\_map() |  |
| } |  |
| } |  |

#### NNR topology unit payload syntax

|  |  |
| --- | --- |
| nnr\_topology\_unit\_payload( ) { | **Descriptor** |
| **topology\_data\_str** | st(v) |
| } |  |

#### NNR quantization unit payload syntax

|  |  |
| --- | --- |
| nnr\_quantization\_unit\_payload( ) { | **Descriptor** |
| **quantization\_data\_str** | st(v) |
| } |  |

#### NNR compressed data unit payload syntax

|  |  |
| --- | --- |
| nnr\_compressed\_data\_unit\_payload( ) { | **Descriptor** |
| if( nnr\_compressed\_data\_unit\_payload\_type = = NNR\_PT\_RAW\_FLOAT32 ) { |  |
| for( i = 0; i < Prod( tensor\_dimensions ); i++ ) { |  |
| **raw\_float32\_parameter[**TensorIndex( tensor\_dimensions, i ) **]** | flt(32) |
| } |  |
| } |  |
| **Invoke subclause 5.2** |  |
| } |  |

#### NNR aggregate unit payload syntax

|  |  |
| --- | --- |
| nnr\_aggregate\_unit\_payload() { | **Descriptor** |
| for(i = 0; i < num\_of\_nnr\_units\_minus2 + 2; i++) { |  |
| nnr\_unit( ) |  |
| } |  |
| } |  |

### 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) |
| } |  |

## Semantics

### 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 Specification.

### NNR unit semantics

#### General NNR unit semantics

#### 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().

#### NNR unit header semantics

**nnr\_unit\_type** specifies the type of the NNR unit, as specified in Table 6.

Table 6: 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…127 | NNR\_RSVD | Reserved | MPEG-reserved range |
| 128..255 | NNR\_UNSP | Unspecified | Unspecified range |

**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 0). This counter counts backwards to indicate initially the total number of partitions.

**independently\_decodable\_flag** specifies whether this compressed data unit is independently decodable. A value of 0 indicates independently decodable NNR Unit. A value of 1 indicates that this NNR Unit is not independently decodable and its payload should be combined with other NNR Units for successful decodability/decompressibility.

##### NNR start unit header semantics

##### NNR model parameter set unit header semantics

##### NNR layer parameter set unit header semantics

##### NNR topology unit header semantics

**topology\_storage\_format** specifies the format of the stored neural network topology information, as specified below:

|  |  |  |
| --- | --- | --- |
| **topology\_storage\_format value** | **Identifier** | **Description** |
| 0 | NNR\_NNEF | Neural network topology information is stored in NNEF format as specified in [NNEF] |
| 1 | NNR\_ONNX | Neural network topology information is stored as ONNX messages [ONNX] |
| 2..127 | NNR\_RSVD | MPEG-reserved range |
| 128..255 | NNR\_UNSP | Unspecified range |

**compressed\_topology\_flag,** when set to 1 indicates that the topology information inside the NNR\_TPL units are further compressed. Otherwise, they are stored in an uncompressed format.

**compression\_format** is an enumerated list which takes one of the following values:

|  |  |  |
| --- | --- | --- |
| **compression\_format** | **Type identifier** | **Type enumeration (7 bits)** |
| Uncompressed | NNR\_PT\_RAW | 0x00 |
| Deflate as defined in RFC 1950 | NNR\_DFL | 0x01 |
| Reserved |  | 0x02-0x7F |

##### NNR quantization unit header semantics

**quantization\_storage\_format** specifies the format of the stored neural network topology information, as specified below:

|  |  |  |
| --- | --- | --- |
| **quantization\_storage\_format value** | **Identifier** | **Description** |
| 0 | NNR\_NNEF | Neural network (optional) quantization information is stored in NNEF format as specified in [NNEF] |
| 1 | NNR\_ONNX | Neural network (optional) quantization information is stored as ONNX messages [ONNX] |
| 2..127 | NNR\_RSVD | MPEG-reserved range |
| 128..255 | NNR\_UNSP | Unspecified range |

**quantization\_compressed\_flag,** when set to 1 indicates that the quantization information inside the NNR\_QNT units are further compressed. Otherwise, they are stored in an uncompressed format.

**compression\_format** is as defined in subclause 7.4.2.3.4:

##### NNR compressed data unit header semantics

**ref\_id** specifies a unique identifier for the NNR compressed data unit element. The semantic interpretation of this field is context dependent.

When NNR bitstream is carried in NNEF container organization or when topology\_storage\_format value in NNER topology unit header is equal to NNR\_NNEF, ref\_id shall be interpreted and processed as specified in section 12.3 and section 12.4.

**count\_topology\_elements** specifies the number of topology elements for which this NNR Compressed Data Unit carries data in the payload.

**count\_tensor\_dimensions** specifies a counter of how many dimensions are specified. For example, for a 4-dimensional tensor, count\_dims is 4.

**tensor\_dimensions** specifies an array or list of dimension values. For example, for a convolutional layer, dim is an array or list of length 4.

**cabac\_unary\_length** specifies the length of the unary part in the CABAC binarization.

**codebook\_zero\_offset** specifies an offset for accessing elements in syntax element codebook.

**codebook\_size** specifies the number of elements in syntax element codebook.

**codebook** specifies the codebook as required by nnr\_compressed\_data\_unit\_payload\_type equal to NNR\_PT\_CB\_FLOAT32.

**nnr\_compressed\_data\_unit\_payload\_type** is as defined in Table 3 of subclause 5.2.

##### NNR aggregate unit header semantics

**nnr\_aggregate\_unit\_type** specifies the type of the aggregate NNR unit.

The following NNR aggregate unit types are specified:

|  |  |  |  |
| --- | --- | --- | --- |
| **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\_XYZ\_1 |  |  |
| 2 | NNR\_AGG\_XYZ\_2 |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
| 3..127 | NNR\_RSVD | Reserved | MPEG-reserved range |
| 128..255 | NNR\_UNSP | Unspecified | Unspecified range |

[Ed(EA): Based on agreement in MPEG 129 on m51611 and m52352, NNR\_AGG\_XYZ\_1 and NNR\_AGG\_XYZ\_2 are defined as placeholders. Proper type names and their definitions require further contributions]

**entry\_points\_present\_flag** specifies whether individual NNR unit entry points are present.

**num\_of\_nnr\_units\_minus2** plus 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 bitdepth 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.

#### NNR unit payload semantics

The following NNR unit payload types are specified:

##### NNR start unit payload semantics

##### NNR model parameter set 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 unit types “NNR\_TPL”. If 0, it specifies that topology is provided externally (i.e., out-of-band with respect to the NNR bitstream).

**quantization\_method\_flags** specifies the quantization method(s) used for the whole model. If multiple models are specified, they are combined by OR. The following methods are defined.

|  |  |  |
| --- | --- | --- |
| **Quantization method** | **Quantization method ID** | **Value** |
| Scalar uniform | NNR\_QSU | 0x01 |
| Codebook | NNR\_QCB | 0x02 |
| Reserved |  | 0x03-0xFF |

**qp\_density** specifies density information of syntax element quantization\_parameter.

**quantization\_parameter** specifies the quantization parameter for scalar uniform quantization of parameters of each layer of the neural network for arithmetic coding.

**sparsification\_flag** specifies whether sparsification was applied to the model.

**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.

**count\_thresholds** specifies the number of sparsification thresholds.

**sparsification\_threshold** specifies the threshold which 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 the non-zero ratio that is achieved by applying the sparsification\_threshold to sparsify the weights.

**nn\_accuracy** specifies the overall accuracy of the NN (e.g., classification accuracy by considering all classes).

**count\_classes** specifies number of classes for which separate accuracies are provided for each sparsification thresholds.

**nn\_class\_accuracy** specifies the accuracy for a certain class, when a certain sparsification threshold is applied.

**ctu\_partition\_flag** specifies if the block partitioning is enabled for weight tensor of each layer of the neural network. value 0 indicates that the block partitioning is disabled, value 1 indicates that the block partitioning is enabled where the partitioned block is defined as CTU (coding tree unit).

**max\_ctu\_dim\_flag** specifies the model-wise max CTU dimension for weight tensor of the neural network: gctu\_dim=(64>>max\_ctu\_dim\_flag). tensor-wise max CTU width is scaled by the kernel size of each convolution tensor: max\_ctu\_height=gctu\_dim, max\_ctu\_width=gctu\_dim\*kernel\_size. The height/width of right/bottom CTUs may be less than the max\_ctu\_height/max\_ctu\_width.

##### NNR layer parameter set unit payload semantics

**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 following methods are defined.

|  |  |  |
| --- | --- | --- |
| **Quantization method** | **Quantization method ID** | **Value** |
| Scalar uniform | NNR\_QSU | 0x01 |
| Codebook | NNR\_QCB | 0x02 |
| Reserved |  | 0x03-0xFF |

**quantization\_step\_size** specifies the step interval for scalar uniform quantization method.

**quantization\_map()** specifies a codebook for codebook-based quantization method.

**compressed\_flag** specifies whether the quantization map data is further compressed.

**compression\_format** is an enumerated list which takes one of the following values and indicates the compression method utilized for further compressing the quantization\_map\_data():

|  |  |  |
| --- | --- | --- |
| **compression\_format** | **Type identifier** | **Type enumeration (7 bits)** |
| Uncompressed | NNR\_PT\_RAW | 0x00 |
| Deflate as defined in RFC 1950 | NNR\_DFL | 0x01 |
| Reserved |  | 0x02-0x7F |
|  |  |  |

**index\_precision** specifies the integer type used in the key values of quantization\_map\_data(). A value of 0 indicates 8 bits precision, value of 1 indicates 16 bits precision and a value of 2 indicates 32 bits precision. The other values are reserved.

**quantization\_map\_data()** specifies an array or dictionary of the form {[index<index\_precision>:value<float(32)>]} where index is a quantized value indicator and the second value is a signed floating-point value corresponding to that quantized value index.

**sparsification\_flag** is as defined in subclause 7.4.2.4.3..

**sparsification\_performance\_map()** is as defined in subclause 7.4.2.4.3..

**independently\_decodable\_flag** is as defined in subclause 7.4.2.4.3.

##### NNR topology unit payload semantics

**topology\_data\_str** is a null-terminated string representation of the neural network topology, in the format specified by topology\_storage\_format.

##### NNR quantization unit payload semantics

**quantization\_data\_str** is a null-terminated string representation of the neural network quantization information, in the format specified by quantization\_storage\_format.

##### NNR compressed data unit payload semantics

**raw\_float32\_parameter** is a float32 parameter tensor.

##### 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.

# Weight sparsification

This includes matrix transformations/decompositions that generate additional data (e.g. masks) or require specific sparse representations

## Method(s)

### Sparsification using compressability 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 on the compressibility loss such that the compressibility loss is 𝑚 times the task loss. This   is fixed during the entire training and the following loss is minimized during neural network training:

𝑚 is a hyperparameter of our method and for the use cases we provide multiple working points via changing 𝑚. Note that   is directly inferred from 𝑚 so it is not an additional hyperparameter.

The compressibility loss is defined as follows

where and are and norms of the vector , respectively.

During this data-dependent transformation we always arrange accordingly such that .

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

## Output metadata and bitstream (normative)

Optional metadata elements

* Rsize != Osize (flag), if set
  + Dimension (int) and shape of original tensors (list of int)

# Parameter reduction methods

This includes matrix decomposition methods that change the structure of the model, but of which the output can be represented as a regular model and additional metadata.

## Method(s)

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

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

In the proposed tests, the weight matrices of some fully connected and convolutional layers of pretrained neural networks are approximated as Low Rank or Low Displacement Rank form. are transmitted in the bitstream. A and B are chosen to be f-circulant operators that, denoted Ze and Zf, expressed as .They are coded in the bitstream using the value of e, f, as the structure of 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 weights as a product:

where is a matrix and is matrix that can be derived from an SVD decomposition and retaining the first singular values and vectors.

Convolutional layers, which consist of 4D-tensors of weight, are reshaped into 2D tensors so that the LDR/LR decomposition can be performed. One of 4 reshaping modes are chosen depending on the tensor dimensions.

If the layer is convolutional then the layer parameters are 4-D tensors. The dimensions of these tensors can be decomposed as ,where is the number of input channels, is the number of output channels, and is the size of 2-D the filter kernel. We propose to compress the convolutional layer tensor by reshaping it to a 2-D matrix by using the following function:

,

where ‘m’ is the mode which can take four values 1,2,3, or 4, depending on which the 2-D matrix is returned as described below:

1. Mode : For a fixed , vectorize the matrix to obtain 1-D vectors of size and stack these vectors as columns of the matrix.
2. Mode : For a fixed , vectorize the matrix to obtain 1-D vectors of size and stack these vectors as columns of the matrix.
3. Mode : For a fixed , vectorize the matrix to obtain 1-D vectors of size and stack these vectors as columns of the matrix.
4. Mode : For a fixed vectorize the 3-D tensor to obtain 1-D vectors of size and stack these as rows of the matrix.

## Output metadata and bitstream (normative)

Required metadata elements

* Factors A, B (list of integers)
* Rank (int)
* Dimension (int) and shape of original tensors (list of int)
* Reshaping mode (int)

# Quantisation, Matrix Approximation, Predictive Coding

## Method(s)

### Baseline method

Uniform quantisation is applied to the parameter tensors using a fixed step size represented by parameters qpDensity and qp according to the specification in clause 5.2.2 and a flag, denoted as dq\_flag, equal to zero. The reconstructed values in the decoded tensor are integer multiples of the step size.

### 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 float32 values.

The reconstructed values in the decoded tensors are the values of codebook elements referred to by their index value.

### Dependent scalar quantization method

Dependent scalar quantization is applied to the parameter tensors using a fixed stepsize represented by parameters qpDensity and qp according to the specification in clause 5.3.3 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.

## Output metadata (normative)

### Baseline method

Required metadata

* qpDensity (unsigned integer)
* qp (integer)
* dq\_flag

### Codebook-based method

Required metadata

* codebook\_zero\_offset (int)
* codebook (list of float32 values)

### Dependent scalar quantization method

Required metadata:

* qpDensity (unsigned integer)
* qp (integer)
* dq\_flag

## Output bitstream (where applicable)

# Entropy Coding

## Method(s)

### DeepCABAC

**Binarization**

The encoding method scans the parameter tensor in a row-first manner from the left to the right and the rows from the top to the bottom. Each quantized parameter level is encoded according to the following procedure employing an integer parameter ‘maxNumNoRem’:

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 elements 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 elements abs\_level\_greater\_X is encoded, which indicates, that the absolute value of the quantized parameter level is greater than X. If abs\_level\_greater\_X is equal to 1 and if X is greater than maxNumNoRem, the variable k is increased by 1. Afterwards, 1 << k is added to X and a further abs\_level\_greater\_X is encoded. This procedure is continued until an abs\_level\_greater\_X 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.

**Context modelling**

Context modeling corresponds to associating the three type of flags sig\_flag, sign\_flag, and abs\_level\_greater\_X 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 modeling 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 neighboring quantized parameter level to the left is zero, smaller, or larger than zero.

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

For the abs\_level\_greater\_X flags, each X uses, either one or two separate context models. If X <= maxNumNoRem, two context models are distinguished depending on the sign\_flag. If X > maxNumNoRem, only one context model is used.

### DeepCABAC Syntax

This clause specifies the entropy coding syntax as used by the decoding process of clause 5.

#### Quantized tensor with CABAC termination syntax

|  |  |
| --- | --- |
| **Syntax** | **Type/Subclause** |
| quant\_tensor\_term( dimensions, maxNumNoRem ) { |  |
| quant\_tensor( dimensions, maxNumNoRem ) | 11.1.2.3 |
| **terminating\_one\_bit** | ae(t) |
| 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.

#### Quantization parameter syntax

|  |  |
| --- | --- |
| **Syntax** | **Type/Subclause** |
| quant\_param( qpDensity ) { |  |
| **qp** | iae(6 + qpDensity) |
| } |  |

**qp** is the quantization parameter.

#### Dependent quantization flag syntax

|  |  |
| --- | --- |
| **Syntax** | **Type/Subclause** |
| dq\_flag() { |  |
| **dq\_flag** | uae(1) |
| } |  |

**dq\_flag** specifies whether the quantization method is dependent scalar quantization or baseline method. A dq\_flag equal to 0 indicates that the baseline method is used. A dq\_flag equal to 1 indicated the dependent scalar quantization is used.

#### Quantized tensor syntax

|  |  |
| --- | --- |
| **Syntax** | **Type/Subclause** |
| quant\_tensor( dimensions, maxNumNoRem) { |  |
| stateId = 0 |  |
| for( i = 0; i < Prod( dimensions ); i++ ) { |  |
| idx = TensorIndex( dimensions, i ) |  |
| int\_param( idx, maxNumNoRem, stateId ) | 11.1.2.5 |
| if(dq\_flag) { |  |
| nextSt = StateTransTab[stateId][QuantParam[idx] & 1] |  |
| if( QuantParam[idx] != 0 ) { |  |
| QuantParam[idx] = QuantParam[idx] << 1 |  |
| QuantParam[idx] += QuantParam[idx] < 0 ?  ( stateId & 1 ) : -( stateId & 1 ) |  |
| } |  |
| stateId = nextSt |  |
| } |  |
| } |  |
| } |  |

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} }

#### Quantized parameter syntax

|  |  |
| --- | --- |
| **Syntax** | **Type/Subclause** |
| int\_param( i, maxNumNoRem, stateId ) { |  |
| 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 < maxNumNoRem ) |  |
| if( j == maxNumNoRem ) { |  |
| RemBits = 0 |  |
| j = -1 |  |
| do { |  |
| j++ |  |
| **abs\_level\_greater\_x2[j]** | ae(v) |
| if( abs\_level\_greater\_x2[j] ) { |  |
| RemBits++ |  |
| QuantParam[i] += 1 << RemBits |  |
| } |  |
| } while( abs\_level\_greater\_x2[j] ) |  |
| **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.

## CABAC parsing process for syntax elements

### 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 11.2.3.

The binarization for the syntax element and the sequence of parsed bins determines the decoding process flow as described in subclause 11.2.4.

### Initialization process

#### General

Outputs of this process are initialized CABAC 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 clause 11.2.2.2.

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 11.2.2.3.

#### Initialization process for context variables

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

For each context variable, the two variables pStateIdx0 and pStateIdx1 are both set to 0.

#### Initialization process for the arithmetic decoding engine

Outputs of this process are the initialized decoding engine registers ivlCurrRange and ivlOffset both in 16 bit register precision.

The status of the arithmetic decoding engine is represented by the variables ivlCurrRange and ivlOffset. In the initialization procedure of the arithmetic decoding process, ivlCurrRange is set equal to 510 and ivlOffset is set equal to the value returned from read\_bits( 9 ) interpreted as a 9 bit binary representation of an unsigned integer with the most significant bit written first.

The bitstream shall not contain data that result in a value of ivlOffset being equal to 510 or 511.

### Binarization process

#### General

Input to this process is a request for a syntax element.

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

All syntax elements use fixed-length (FL) binarization process.

The specification of the fixed-length (FL) binarization process is given in subclause 11.2.3.2.

#### Fixed-length binarization process

Input to this process is a request for a fixed-length (FL) binarization.

Output of this process is the FL binarization associating each value symbolVal with a corresponding bin string.

FL binarization is constructed by using the fixedLength‑bit unsigned integer bin string of the symbol value symbolVal, where fixedLength = Ceil( Log2( cMax + 1 ) ). The indexing of bins for the FL binarization is such that the binIdx = 0 relates to the most significant bit with increasing values of binIdx towards the least significant bit.

### Decoding process flow

#### General

Inputs to this process are all bin strings of the binarization of the requested syntax element as specified in clause 11.2.3.

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

This process specifies how each bin of a bin string is parsed for each syntax element. After parsing each bin, the resulting bin string is compared to all bin strings of the binarization of the syntax element and the following applies:

– If the bin string is equal to one of the bin strings, the corresponding value of the syntax element is the output.

– Otherwise (the bin string is not equal to one of the bin strings), the next bit is parsed.

While parsing each bin, the variable binIdx is incremented by 1 starting with binIdx being set equal to 0 for the first bin.

The parsing of each bin is specified by the following two ordered steps:

1. The derivation process for ctxIdx and bypassFlag as specified in clause 11.2.4.2 is invoked with binIdx as input and ctxIdx and bypassFlag as outputs.

2. The arithmetic decoding process as specified in clause 11.2.4.3.2 is invoked with ctxIdx and bypassFlag as inputs and the value of the bin as output.

#### Derivation process for ctxIdx and bypassFlag

##### General

Input to this process is the position of the current bin within the bin string, binIdx.

Outputs of this process ctxIdx and bypassFlag.

The values of ctxIdx and bypassFlag are derived as follows based on the entries for binIdx of the corresponding syntax element inTable 7:

* If the entry in Table 7 is not equal to "bypass" or "na", the values of binIdx are decoded by invoking the DecodeDecision process as specified in clause 11.2.4.3.2 and the following applies:
* The variable ctxInc is specified by the corresponding entry in Table 7 and when more than one value is listed in Table 7 for a binIdx, the assignment process for ctxInc for that binIdx is further specified in the clauses given in parenthesis.
* bypassFlag is set equal to 0.
* Otherwise, if the entry in Table 7 is equal to "bypass", the values of binIdx are decoded by invoking the DecodeBypass process as specified in clause 11.2.4.3.4 and the following applies:
* ctxIdx is set equal to 0.
* bypassFlag is set equal to 1.
* Otherwise (the entry in Table 2 is equal to "na"), the values of binIdx do not occur for the corresponding syntax element.

Table 7: Assignment of ctxInc to syntax elements with context coded bins

| **Syntax element** | **binIdx** | | | | | |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **>= 5** |
| sig\_flag | 0..23 (subclause 11.2.4.2.2) | na | na | na | na | na |
| sign\_flag | 0..2 (subclause 11.2.4.2.3) | na | na | na | na | na |
| abs\_level\_greater\_x[j] | 2\*j+(0..1) (subclause X) | na | na | na | na | na |
| abs\_level\_greater\_x2[j] | j | na | na | na | na | na |

##### Derivation process of ctxInc for the syntax element sig\_flag

Inputs to this process are the sig\_flag decoded before the current sig\_flag, the state value stateId and the associated sign\_flag, if present. If no sig\_flag was decoded before the current sig\_flag, it is assumed to be 0. If no sign\_flag associated with the previously decoded sig\_flag was decoded, it is assumed to be 0.

Output of this process is the variable ctxInc.

The variable ctxInc is derived as follows:

* If sig\_flag is equal to 0, ctxInc is set to stateId\*3.
* Otherwise, if sign\_flag is equal to 0, ctxInc is set to stateId\*3+1.
* Otherwise, ctxInc is set to stateId\*3+2.

##### Derivation process of ctxInc for the syntax element sign\_flag

Inputs to this process are the sig\_flag decoded before the current sig\_flag and the associated sign\_flag, if present. If no sig\_flag was decoded before the current sig\_flag, it is assumed to be 0. If no sign\_flag associated with the previously decoded sig\_flag was decoded, it is assumed to be 0.

Output of this process is the variable ctxInc.

The variable ctxInc is derived as follows:

* If sig\_flag is equal to 0, ctxInc is set to 0.
* Otherwise, if sign\_flag is equal to 0, ctxInc is set to 1.
* Otherwise, ctxInc is set to 2.

##### Derivation process of ctxInc for the syntax element abs\_level\_greater\_x[j]

Inputs to this process are the sign\_flag decoded before the current syntax element abs\_level\_greater\_x[j].

Output of this process is the variable ctxInc.

The variable ctxInc is derived as follows:

* If sign\_flag is equal to 0, ctxInc is set to 2\*j.
* Otherwise, ctxInc is set to 2\*j+1.

#### Arithmetic decoding process

##### General

Inputs to this process are ctxIdx and bypassFlag, as derived in subclause 11.2.4.2, and the state variables ivlCurrRange and ivlOffset of the arithmetic decoding engine.

Output of this process is the value of the bin.

For decoding the value of a bin, the ctxIdx and the bypassFlag are passed to the arithmetic decoding process DecodeBin( ctxIdx, bypassFlag ), which is specified as follows:

– If bypassFlag is equal to 1, DecodeBypass( ) as specified in subclause 11.2.4.3.4 is invoked.

– Otherwise, DecodeDecision( ctxIdx ) as specified in subclause 11.2.4.3.2 is invoked.

NOTE – Arithmetic coding is based on the principle of recursive interval subdivision. Given a probability estimation p( 0 ) and p( 1 ) = 1 − p( 0 ) of a binary decision ( 0, 1 ), an initially given code sub-interval with the range ivlCurrRange will be subdivided into two sub-intervals having range p( 0 ) \* ivlCurrRange and ivlCurrRange − p( 0 ) \* ivlCurrRange, respectively. Depending on the decision, which has been observed, the corresponding sub-interval will be chosen as the new code interval, and a binary code string pointing into that interval will represent the sequence of observed binary decisions. It is useful to distinguish between the most probable symbol(MPS) and the least probable symbol(LPS), so that binary decisions have to be identified as either MPS or LPS, rather than 0 or 1. Given this terminology, each context is specified by the probability pLPS of the LPS and the value of MPS (valMps), which is either 0 or 1. The arithmetic core engine in this Specification has three distinct properties:

– The probability estimation is performed by means of a finite-state machine with a table-based transition process between 64 different representative probability states { pLPS( pStateIdx ) | 0  <=  pStateIdx < 64 } for the LPS probability pLPS. The numbering of the states is arranged in such a way that the probability state with indexpStateIdx = 0 corresponds to an LPS probability value of 0.5, with decreasing LPS probability towards higher state indices.

– The range ivlCurrRange representing the state of the coding engine is quantized to a small set {Q1,…,Q4} of pre-set quantization values prior to the calculation of the new interval range. Storing a table containing all 64x4 pre-computed product values of Qi \* pLPS( pStateIdx ) allows a multiplication-free approximation of the product ivlCurrRange \* pLPS( pStateIdx ).

– For syntax elements or parts thereof for which an approximately uniform probability distribution is assumed to be given a separate simplified encoding and decoding bypass process is used.

##### Arithmetic decoding process for a binary decision

###### General

Inputs to this process are the variables ctxIdx, ivlCurrRange, and ivlOffset.

Outputs of this process are the decoded value binVal, and the updated variables ivlCurrRange and ivlOffset.

For decoding a single decision (DecodeDecision), the following applies:

1. The value of the variable ivlLpsRange is derived as follows:

– Given the current value of ivlCurrRange, the variable qRangeIdx is derived as follows:

qRangeIdx = ivlCurrRange & 0xe0

– Given qRangeIdx, pStateIdx0 and pStateIdx1 associated with ctxIdx and the current syntax element, valMps and ivlLpsRange are derived as follows:

valMps = pStateIdx0 + pStateIdx0 >= 0  
rlps\_table = [128, 112, 97, 84, 74, 65, 57, 50, 45, 39, 34, 30, 27, 23, 20, 18, 15, 14, 12, 11, 10, 9, 7, 7,

5, 5, 4, 4, 3, 3, 2, 2, 142, 125, 108, 93, 82, 72, 63, 56, 50, 43, 38, 33, 30, 26, 22, 20,

17, 16, 13, 12, 11, 10, 8, 8, 6, 6, 5, 5, 3, 3, 2, 2, 156, 137, 119, 103, 90, 79, 70, 61,

55, 48, 42, 37, 33, 28, 24, 22, 19, 17, 15, 13, 12, 11, 9, 9, 6, 6, 5, 5, 4, 4, 2, 2,

171, 150, 130, 112, 99, 87, 76, 67, 60, 52, 46, 40, 36, 31, 27, 24, 21, 19, 16, 15, 13, 12, 10, 10,

7, 7, 6, 6, 4, 4, 3, 3, 185, 162, 141, 121, 107, 94, 82, 73, 65, 56, 50, 43, 39, 34, 29, 26,

22, 21, 17, 16, 14, 13, 11, 11, 8, 8, 6, 6, 4, 4, 3, 3, 199, 175, 152, 131, 115, 101, 89, 78,

70, 61, 54, 47, 42, 36, 31, 28, 24, 22, 19, 17, 15, 14, 12, 12, 8, 8, 7, 7, 5, 5, 3, 3,

213, 187, 163, 140, 123, 108, 95, 84, 75, 65, 58, 50, 45, 39, 33, 30, 26, 24, 20, 18, 16, 15, 13, 13,

9, 9, 7, 7, 5, 5, 3, 3, 228, 200, 174, 150, 132, 116, 102, 90, 80, 70, 62, 54, 48, 42, 36, 32,

28, 26, 22, 20, 18, 16, 14, 14, 10, 10, 8, 8, 6, 6, 4, 4]  
ivlLpsRange = rps\_table[(abs((pStateIdx0 + pStateIdx1) >> 7)) + qRangeIdx]

1. The variable ivlCurrRange is set equal to ivlCurrRange − ivlLpsRange and the following applies:

– If ivlOffset is greater than or equal to ivlCurrRange, the variable binVal is set equal to 1 − valMps, ivlOffset is decremented by ivlCurrRange, and ivlCurrRange is set equal to ivlLpsRange.

– Otherwise, the variable binVal is set equal to valMps.

Given the value of binVal, the state transition isperformed as specified in subclause 11.2.4.3.2.2. Depending on the current value of ivlCurrRange, renormalization is performed as specified in subclause 11.2.4.3.3.

###### State transition process

Inputs to this process are the current pStateIdx0 and pStateIdx1, and the decoded value binVal.

Outputs of this process are the updated pStateIdx0 and pStateIdx1 of the context variable associated with ctxIdx.

The variables shift0 is set to 1 and shift1 is set to 4.

Depending on the decoded value binVal, the update of the two variables pStateIdx0 and pStateIdx1 associated with ctxIdx and with the syntax element is derived as follows:

transition\_table = [2512, 2288, 2064, 1840, 1616, 1392, 1168, 944, 720, 560, 464, 368, 272, 208, 144, 80, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 0]

sign = 2 \* binVal - 1

pStateIdx0 += sign \* (transition\_table[16 + (sign \* pStateIdx0 >> 3)] >> shift0)

pStateIdx1 += sign \* (transition\_table[16 + (sign \* pStateIdx0 >> 7)] >> shift1)

##### Renormalization process in the arithmetic decoding engine

Inputs to this process are bits from slice data and the variables ivlCurrRange and ivlOffset.

Outputs of this process are the updated variables ivlCurrRange and ivlOffset.

The current value of ivlCurrRange is first compared to 256 and further steps are specified as follows:

– If ivlCurrRange is greater than or equal to 256, no renormalization is needed and the process is finished;

– Otherwise (ivlCurrRange is less than 256), the renormalization loop is entered. Within this loop, the value of ivlCurrRange is doubled, i.e. left-shifted by 1 and a single bit is shifted into ivlOffset by using read\_bits( 1 ).

The bitstream shall not contain data that result in a value of ivlOffset being greater than or equal to ivlCurrRange upon completion of this process.

##### Bypass decoding process for binary decisions

Inputs to this process are bits from slice data and the variables ivlCurrRange and ivlOffset.

Outputs of this process are the updated variable ivlOffset and the decoded value binVal.

First, the value of ivlOffset is doubled, i.e. left-shifted by 1 and a single bit is shifted into ivlOffset by using read\_bits( 1 ). Then, the value of ivlOffset is compared to the value of ivlCurrRange and further steps are specified as follows:

– If ivlOffset is greater than or equal to ivlCurrRange, the variable binVal is set equal to 1 and ivlOffset is decremented by ivlCurrRange.

– Otherwise (ivlOffset is less than ivlCurrRange), the variable binVal is set equal to 0*.*

The bitstream shall not contain data that result in a value of ivlOffset being greater than or equal to ivlCurrRange upon completion of this process.

##### Decoding process for binary decisions before termination

Inputs to this process are bits from slice data and the variables ivlCurrRange and ivlOffset.

Outputs of this process are the updated variables ivlCurrRange and ivlOffset, and the decoded value binVal.

The decoding process is specified as follows:

First, the value of ivlCurrRange is decremented by 2. Then, the value of ivlOffset is compared to the value of ivlCurrRange and further steps are specified as follows:

– If ivlOffset is greater than or equal to ivlCurrRange, the variable binVal is set equal to 1, no renormalization is carried out, and CABAC decoding is terminated. The last bit inserted in register ivlOffset is equal to 1. When decoding slice\_one\_bit, end\_of\_tile\_one\_bit, and end\_of\_subset\_one\_bit, this last bit inserted in register ivlOffset is interpreted as rbsp\_stop\_one\_bit.

– Otherwise (ivlOffset is less than ivlCurrRange), the variable binVal is set equal to 0 and renormalization is performed as specified in subclause 11.2.4.3.3.

# Usage (informative)

This section will contain examples of pipelines that can be generated

# Annex A: implementation for NNEF

## General

Neural Network Exchange Format, as specified in [NNEF] is a data format for exchanging information about (trained) neural networks. This annex specifies how compressed representation of neural networks, as specified in this specification, are utilized in NNEF context.

In this version of Specification, [NNEF] is taken as reference for Neural Network Exchange Format (NNEF).

In NNEF context, NNR bitstream can be stored and exchanged via two methods:

* Carriage of NNEF data in NNR bitstream
* Carriage of NNR bitstream in NNEF container organization (as defined in section 5.1 of [NNEF]).

## Definitions for use in NNR bitstream (normative)

**NNEF topology** is a textual information describing the structure of the neural network according to the syntax as specified in section 3.2 of [NNEF]. This information is stored as a textual file under NNEF-specified file directory tree structure.

Other NNEF specific data structures and acronyms are used as specified in [NNEF].

## Carriage of NNEF data in NNR bitstream (normative)

If NNEF topology information is provided in-band of the NNR bitstream, then the following constraints shall apply:

* NNEF topology information shall be carried in the NNR topology unit’s topology\_data\_str parameter.
* NNR topology unit shall precede any NNR compressed data unit.
* topology\_storage\_format value of NNR topology unit header shall be set to NNR\_NNEF.

If NNEF optional quantization information as specified in section 5.1 of [NNEF] is provided in-band of the NNR bitstream, then the following constraints shall apply:

* Optional NNEF quantization information shall be carried in the NNR quantization unit’s quantization\_data\_str parameter.
* NNR quantization unit shall precede any NNR compressed data unit.
* quantization\_storage\_format value of NNR quantization unit header shall be set to NNR\_NNEF.

The following constraints shall apply to NNR compressed data units:

* ref\_id in NNR compressed data unit header shall contain the related NNEF variable label as specified in [NNEF]. This enables mapping of a uniquely identifiable NNEF data structure to an NNR compressed data unit.

## Recommendation for carriage of NNR bitstream in NNEF container organization (to be specified within the NNEF format specification)

If NNR bitstream is carried in NNEF container organization, then NNR compressed bitstreams of uniquely identifiable NNEF variables shall be stored in corresponding “.dat” files in the directory tree structure as specified in [NNEF].

For NNR compressed data payload types NNR\_PT\_INT32 and NNR\_PT\_FLOAT32, the syntax element cabac\_unary\_length should be set to 10 and the syntax element tensor\_dimensions should be as signaled in the relevant NNR compressed data unit payload.

For NNR compressed data payload type NNR\_PT\_FLOAT32, the input variable qpDensity should be set to 2.

The payload of the NNEF Tensor File Format will contain only the information to decode a single tensor.

For such NNR bitstreams, the following constraints shall apply:

* NNR topology units and NNR quantization units shall not be present in the stored NNR bitstream
* ref\_id field of NNR compressed data unit headers shall contain the label of uniquely identifiable NNEF variable which corresponds to the location in the directory tree structure as specified in [NNEF]

[Ed (EA): Should storage format related examples be added to an informative annex in order to clarify the carriage scenarios?]

# Annex B: Recommended implementation for ONNX (informative)

## General

Open Neural Network Exchange, as specified in [ONNX] is a data format for exchanging information about (trained) neural networks. This annex specifies how compressed representation of neural networks, as specified in this specification, are utilized in ONNX context.

In this version of Specification, [ONNX] is taken as reference for Open Neural Network Exchange (ONNX).

In ONNX context, NNR bitstream can be stored and exchanged via two methods:

* Carriage of ONNX data in NNR bitstream
* Carriage of NNR bitstream in ONNX container organization (requires new DataType in TensorProto).

## Definitions for use in NNR bitstream

**ONNX topology** is described in a GraphProto using NodeProto protobuf messages that contain information about the structure of the neural network. This information is stored a under ONNX-specified GraphProto at model.graph.node.

Other ONNX data structures and acronyms are used as specified in [ONNX].

**ONNX weights** are optionally stored in TensorProtos as specified in [ONNX] under model.graph.initializer.

## Carriage of ONNX data in NNR bitstream

If ONNX topology information is provided in-band of the NNR bitstream, then the following constraints shall apply:

* ONNX topology information shall be carried in the NNR topology unit’s topology\_data\_str parameter.
* NNR topology unit shall precede any NNR compressed data unit.
* topology\_storage\_format value of NNR topology unit header shall be set to NNR\_ONNX.

If ONNX optional quantization information as specified in GraphProto and TensorAnnotation of [ONNX] is provided in-band of the NNR bitstream, then the following constraints shall apply:

* Optional ONNX quantization information shall be carried in the NNR quantization unit’s quantization\_data\_str parameter.
* NNR quantization unit shall precede any NNR compressed data unit.
* quantization\_storage\_format value of NNR quantization unit header shall be set to NNR\_ONNX.

The following constraints shall apply to NNR compressed data units:

* ref\_id in NNR compressed data unit header shall contain the related ONNX variable label as specified in [ONNX] NodeProto.input. This enables mapping of a uniquely identifiable ONNX data structure to an NNR compressed data unit.

## Recommendation for carriage of NNR coded bitstream inside ONNX (to be specified within the ONNX format specification)

If an NNR bitstream is carried in ONNX container organization, an nnr\_unit shall be stored under field raw\_data in TensorProto and the following applies:

Syntax element qp\_density shall be inferred to equal 2.

Syntax element quantization\_parameter shall be inferred to equal 0.

Syntax element nnr\_unit\_type shall equal NNR\_NDU.

Syntax element nnr\_compressed\_data\_unit\_payload\_type shall equal NNR\_PT\_FLOAT32.

Syntax element tensor\_dimensions\_flag shall be equal to 0.

Syntax element cabac\_unary\_length\_flag shall be equal to 0.

Syntax element cabac\_unary\_length shall be euqal to 10.

Syntax element tensor\_dimenstions shall be set to integer list TensorProto.dims.

The following value is added to enum DataType of TensorProto:

ISO\_IEC\_15938\_17

The field data\_type in TensorProto (which is one of the values specified in enum DataType of TensorProto) shall be set to ISO\_IEC\_15938\_17.

The payload will contain only the information to decode a single tensor. TensorProtos are located at model.graph.initializer.

NNR bitstream can be also provided externally, also for specific (compressed) layers only as specified in TensorProto.DataLocation.

For such NNR bitstreams, the following constraints shall apply:

* TensorProto.data\_type shall be set to ISO\_IEC\_15938\_17
* TensorProto.external\_data field shall contain the filesystem path to NNR bitstream and an offset to the corresponding bitstream format as specific in TensorProto.external\_data [ONNX]

# References

[NNEF] Neural Network Exchange Format, The Khronos NNEF Working Group, Version 1.0, Revision 3, 2018-06-13 (<https://www.khronos.org/registry/NNEF/specs/1.0/nnef-1.0.pdf>)

[ONNX] Open Neural Network Exchange, VERSION 6, 2019-09-19 (https://github.com/onnx/onnx/blob/master/onnx/onnx.proto)

1. The terms lossy and lossless are used w.r.t. to a reconstruction of the encoded weights/parameters. Depending on the application and/or model lossy compression may correspond to a loss in terms of the respective performance metric. [↑](#footnote-ref-1)