

Open Neural Network Exchange (ONNX) — Part 2: Operator sets

Preparatory Standard

Warning for drafts

This document is a draft and is not a standard. It is distributed for review and comment, is subject to change without notice, and may not be referred to as a standard.

Recipients of this draft are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

Table of contents

Foreword.....	vii
Introduction.....	viii
1. Scope.....	1
2. Normative references.....	1
3. Terms and definitions.....	1
4. Required form of an operator specification.....	1
4.1. General.....	1
4.2. Worked example of the required form.....	2
4.3. Operator set versions.....	2
5. The default domain.....	3
5.1. General.....	3
5.2. Abs.....	3
5.3. Acos.....	4
5.4. Acosh.....	4
5.5. Add.....	4
5.6. AffineGrid.....	5
5.7. And.....	6
5.8. ArgMax.....	7
5.9. ArgMin.....	8
5.10.Asin.....	8
5.11.Asinh.....	9
5.12.Atan.....	9
5.13.Atanh.....	10
5.14.Attention.....	10
5.15.AveragePool.....	16
5.16.BatchNormalization.....	18
5.17.Bernoulli.....	19
5.18.BitCast.....	20
5.19.BitShift.....	21
5.20.BitwiseAnd.....	22
5.21.BitwiseNot.....	22
5.22.BitwiseOr.....	23
5.23.BitwiseXor.....	23
5.24.BlackmanWindow.....	24
5.25.Cast.....	25
5.26.CastLike.....	27
5.27.CausalConvWithState.....	28
5.28.Ceil.....	29
5.29.Celu.....	30
5.30.CenterCropPad.....	30
5.31.Clip.....	31
5.32.Col2Im.....	32
5.33.Compress.....	33
5.34.Concat.....	34
5.35.ConcatFromSequence.....	34
5.36.Constant.....	35
5.37.ConstantOfShape.....	36
5.38.Conv.....	37
5.39.ConvInteger.....	38
5.40.ConvTranspose.....	39
5.41.Cos.....	41
5.42.Cosh.....	41
5.43.CumProd.....	42
5.44.CumSum.....	43
5.45.DFT.....	44
5.46.DeformConv.....	45
5.47.DepthToSpace.....	46
5.48.DequantizeLinear.....	47

5.49.Det.....	49
5.50.Div.....	49
5.51.Dropout.....	50
5.52.DynamicQuantizeLinear.....	51
5.53.Einsum.....	52
5.54.Elu.....	53
5.55.Equal.....	53
5.56.Erf.....	54
5.57.Exp.....	54
5.58.Expand.....	55
5.59.EyeLike.....	55
5.60.Flatten.....	56
5.61.Floor.....	57
5.62.GRU.....	57
5.63.Gather.....	60
5.64.GatherElements.....	61
5.65.GatherND.....	63
5.66.Gelu.....	65
5.67.Gemm.....	65
5.68.GlobalAveragePool.....	66
5.69.GlobalLpPool.....	67
5.70.GlobalMaxPool.....	67
5.71.Greater.....	68
5.72.GreaterOrEqual.....	69
5.73.GridSample.....	69
5.74.GroupNormalization.....	71
5.75.HammingWindow.....	72
5.76.HannWindow.....	73
5.77.HardSigmoid.....	73
5.78.HardSwish.....	74
5.79.Hardmax.....	74
5.80.Identity.....	75
5.81.If.....	76
5.82.ImageDecoder.....	77
5.83.InstanceNormalization.....	78
5.84.IsInf.....	79
5.85.IsNaN.....	79
5.86.LRN.....	80
5.87.LSTM.....	81
5.88.LayerNormalization.....	83
5.89.LeakyRelu.....	84
5.90.Less.....	85
5.91.LessOrEqual.....	85
5.92.LinearAttention.....	86
5.93.Log.....	87
5.94.LogSoftmax.....	88
5.95.Loop.....	89
5.96.LpNormalization.....	92
5.97.LpPool.....	92
5.98.MatMul.....	94
5.99.MatMulInteger.....	94
5.100.Max.....	95
5.101.MaxPool.....	96
5.102.MaxRoiPool.....	98
5.103.MaxUnpool.....	98
5.104.Mean.....	100
5.105.MeanVarianceNormalization.....	100
5.106.MelWeightMatrix.....	101
5.107.Min.....	102
5.108.Mish.....	102
5.109.Mod.....	103
5.110.Mul.....	104
5.111.Multinomial.....	104

5.11	Neg	105
5.11	NegativeLogLikelihoodLoss	106
5.11	NonMaxSuppression	108
5.11	NonZero	109
5.11	Not	109
5.11	OneHot	110
5.11	Optional	111
5.11	OptionalGetElement	113
5.12	OptionalHasElement	114
5.12	Or	116
5.12	Relu	116
5.12	Pad	117
5.12	Pow	119
5.12	QLinearConv	120
5.12	QLinearMatMul	122
5.12	QuantizeLinear	123
5.12	RMSNormalization	125
5.12	RNN	125
5.13	RandomNormal	127
5.13	RandomNormalLike	128
5.13	RandomUniform	129
5.13	RandomUniformLike	130
5.13	Range	130
5.13	Reciprocal	132
5.13	ReduceL1	132
5.13	ReduceL2	133
5.13	ReduceLogSum	134
5.13	ReduceLogSumExp	135
5.14	ReduceMax	136
5.14	ReduceMean	137
5.14	ReduceMin	138
5.14	ReduceProd	139
5.14	ReduceSum	140
5.14	ReduceSumSquare	141
5.14	RegexFullMatch	142
5.14	Relu	142
5.14	Reshape	143
5.14	Resize	144
5.15	ReverseSequence	146
5.15	RoiAlign	147
5.15	RotaryEmbedding	148
5.15	Round	151
5.15	STFT	152
5.15	Scan	153
5.15	Scatter	156
5.15	ScatterElements	157
5.15	ScatterND	159
5.15	Selu	161
5.16	SequenceAt	162
5.16	SequenceConstruct	162
5.16	SequenceEmpty	163
5.16	SequenceErase	163
5.16	SequenceInsert	164
5.16	SequenceLength	165
5.16	SequenceMap	166
5.16	Shape	167
5.16	Shrink	168
5.16	Sigmoid	168
5.17	Sign	169
5.17	Sin	169
5.17	Sinh	170
5.17	Size	170
5.17	Slice	171

5.17 5 Softmax.....	172
5.17 6 SoftmaxCrossEntropyLoss.....	173
5.17 7 Softplus.....	175
5.17 8 Softsign.....	175
5.17 9 SpaceToDepth.....	176
5.18 0 Split.....	176
5.18 1 SplitToSequence.....	177
5.18 2 Sqrt.....	178
5.18 3 Squeeze.....	179
5.18 4 StringConcat.....	179
5.18 5 StringNormalizer.....	180
5.18 6 StringSplit.....	181
5.18 7 Sub.....	182
5.18 8 Sum.....	182
5.18 9 SwiGLU.....	183
5.19 0 Swish.....	183
5.19 1 Tan.....	184
5.19 2 Tanh.....	184
5.19 3 TensorScatter.....	185
5.19 4 TfidfVectorizer.....	186
5.19 5 ThresholdedRelu.....	188
5.19 6 Tile.....	188
5.19 7 TopK.....	189
5.19 8 Transpose.....	190
5.19 9 Trilu.....	191
5.20 0 Unique.....	192
5.20 1 Unsqueeze.....	194
5.20 2 Upsample.....	195
5.20 3 Where.....	195
5.20 4 Xor.....	196
6. The ai.onnx.ml domain.....	197
6.1. General.....	197
6.2. ai.onnx.ml.ArrayFeatureExtractor.....	197
6.3. ai.onnx.ml.Binarizer.....	197
6.4. ai.onnx.ml.CastMap.....	198
6.5. ai.onnx.ml.CategoryMapper.....	199
6.6. ai.onnx.ml.DictVectorizer.....	199
6.7. ai.onnx.ml.FeatureVectorizer.....	200
6.8. ai.onnx.ml.Imputer.....	201
6.9. ai.onnx.ml.LabelEncoder.....	202
6.10.ai.onnx.ml.LinearClassifier.....	203
6.11.ai.onnx.ml.LinearRegressor.....	203
6.12.ai.onnx.ml.Normalizer.....	204
6.13.ai.onnx.ml.OneHotEncoder.....	205
6.14.ai.onnx.ml.SVMClassifier.....	206
6.15.ai.onnx.ml.SVMRegressor.....	206
6.16.ai.onnx.mlScaler.....	207
6.17.ai.onnx.ml.TreeEnsemble.....	208
6.18.ai.onnx.ml.TreeEnsembleClassifier.....	209
6.19.ai.onnx.ml.TreeEnsembleRegressor.....	210
6.20.ai.onnx.ml.ZipMap.....	211
Annex A Operator registry.....	212
Annex B Changelog.....	213
Bibliography.....	214

Foreword

This document has been prepared by the ONNX Standardization Working Group.

It is Part 2 of ONNX 1; Part 1 specifies the core and Part 3 the conformance test package.

Status of this document —

This is an unofficial working draft. It has no standing within the ONNX project, the Linux Foundation, IEC, ISO or any other standards development organization, and it MUST NOT be cited as a normative reference.

Introduction

This part carries the operator definitions. It is separate from Part 1 because the two move at different speeds: the core changes with the IR version, roughly once a year, while the operator sets change with every ONNX release, by hundreds of entries.

Separating them means an operator added upstream amends this part and leaves Part 1 untouched: implementers do not re-read the core, and a citation of the core does not go stale because an operator was added.

The seam that makes this work is that Part 1 specifies the **form** an operator definition takes and the rules an operator set obeys, and names no operator. This part supplies the definitions in that form.

Open Neural Network Exchange (ONNX) — Part 2: Operator sets

1. Scope

This document specifies the operator sets of the Open Neural Network Exchange (ONNX) format: the operators of each domain, at each operator set version, in the form required by [ONNX 1-1](#).

This document specifies:

- the form an operator definition takes;
- the operators of the default domain;
- the operators of the `ai.onnx.ml` domain;
- for each operator, its signature, attributes, type constraints, shape inference, semantics and error conditions.

This document does not specify:

- the information model, type system, evaluation semantics, validation rules, numerical obligations, encoding or versioning rules, which are in [ONNX 1-1](#);
- the conformance test vectors, which are in [ONNX 1-3](#).

2. Normative references

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

ONNX 1-1, *Open Neural Network Exchange (ONNX) — Part 1: Core*

ONNX 1-3, *Open Neural Network Exchange (ONNX) — Part 3: Conformance test package*

IEEE 754-2019, *IEEE Standard for Floating-Point Arithmetic*

3. Terms and definitions

No terms and definitions are listed in this document.

The terms and definitions given in [ONNX 1-1](#) apply.

NOTE In particular **operator**, **domain**, **operator set**, **attribute**, **tensor**, **element type** and **shape** are defined in Clause 3 of [ONNX 1-1](#) and are not restated here.

4. Required form of an operator specification

4.1. General

Every operator specified by this document SHALL be specified in the form given in Table 1.

The form is normative. An operator definition that omits a mandatory element is not a definition for the purposes of [ONNX 1-1](#), and an implementation cannot be required to support it.

Table 1 — Required elements of an operator specification

Element	Obligation	Content
Name and domain	mandatory	Identifier of the operator.
Since version	mandatory	Operator set version in which this definition took effect.
Inputs	mandatory	Name, type constraint, and optionality of each formal input.
Outputs	mandatory	Name and type constraint of each formal output.
Attributes	mandatory	Name, type, default value, and obligation of each attribute.
Type constraints	mandatory	Named sets of admissible element types.
Shape inference	mandatory	The output shape as a function of input shapes and attributes.
Semantics	mandatory	The value of each output as a function of the inputs.
Determinism	conditional	Required if the operator is not deterministic; see Clause 11 of ONNX 1-1 .
Errors	mandatory	The conditions under which the operator is not defined.
Test vectors	mandatory	Reference to the vectors in ONNX 1-3 that check this operator.
Examples	optional	Informative.

NOTE The present ONNX operator documentation supplies most of these elements for most operators, but not all of them for all operators. Annex C of [ONNX 1-1](#) records the gaps.

4.2. Worked example of the required form

The following illustrates the required form. It is normative as to **form**; the semantics given are those of the operator as currently defined.

EXAMPLE — ReLU (default domain)

Since version	14
Inputs	X (type constraint T), required.
Outputs	Y (type constraint T).
Attributes	None.
Type constraints	T : FLOAT, DOUBLE, FLOAT16, BFLOAT16, INT8, INT16, INT32, INT64.
Shape inference	The shape of Y equals the shape of X.
Semantics	For every index i into X, $Y_i = \max(X_i, 0)$.
Determinism	Deterministic.
Errors	None. For floating-point element types, the result for a NaN input is NaN.
Test vectors	ai.onnx/ReLU/14 in ONNX 1-3 .

Editorial note —

Confirm the NaN behaviour of the example above against the reference implementation before the committee draft. $\max(\text{NaN}, 0)$ is not well defined by [IEEE 754-2019](#) without stating which of `maxNum` or `maximum` is intended, and Clause 11 of [ONNX 1-1](#) owes that decision once for all operators rather than per definition.

4.3. Operator set versions

Each clause below specifies one domain. Within a domain, an operator definition is identified by its name together with the operator set version in which it took effect.

The evolution rules an operator set obeys are in Clause 14 of [ONNX 1-1](#).

5. The default domain

5.1. General

This clause specifies the operators of the `ai.onnx` domain, in the form required by [Clause 4](#).

Editorial note —

This clause is generated from the upstream operator documentation, which does not supply three of the elements [Clause 4](#) makes mandatory: shape inference, determinism and error conditions. Every operator below records their absence rather than omitting the elements, so that the incompleteness is visible where it matters instead of only in Annex C of [ONNX 1-1](#).

Supplying them is the substance of the work this part represents. They cannot be generated, because the upstream source does not contain them; they have to be written, per operator, and agreed.

Operator prose below also refers to upstream documents that [ONNX 1-1](#) does not yet restate, broadcasting among them. Those references appear as plain text, because a normative cross-reference to a clause that does not exist would be worse than none. Annex C of [ONNX 1-1](#) records the omission.

NOTE Examples and sample implementations are not reproduced. They are informative under [Clause 4](#), and the reference implementation is not restated here; see Annex D of [ONNX 1-1](#). The test vector names extracted from the examples are retained, because [ONNX 1-3](#) needs them.

5.2. Abs

Absolute takes one input data (Tensor<T>) and produces one output data (Tensor<T>) where absolute value, $y = \text{abs}(x)$, is applied to the tensor elementwise.

Domain	<code>ai.onnx</code>
Since version	13
Earlier versions	1, 6
Inputs	X (differentiable) : T — Input tensor
Outputs	Y (differentiable) : T — Output tensor
Attributes	None.
Type constraints	T : <code>tensor(uint8)</code> , <code>tensor(uint16)</code> , <code>tensor(uint32)</code> , <code>tensor(uint64)</code> , <code>tensor(int8)</code> , <code>tensor(int16)</code> , <code>tensor(int32)</code> , <code>tensor(int64)</code> , <code>tensor(float16)</code> , <code>tensor(float)</code> , <code>tensor(double)</code> , <code>tensor(bfloat16)</code> — Constrain input and output types to all numeric tensors.
Test vectors	<code>test_abs</code>
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.3. Acos

Calculates the arccosine (inverse of cosine) of the given input tensor, element-wise.

Domain	ai.onnx
Since version	22
Earlier versions	7
Inputs	input (differentiable) : T—Input tensor
Outputs	output (differentiable) : T—The arccosine of the input tensor computed element-wise
Attributes	None.
Type constraints	T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain input and output types to float tensors.
Test vectors	test_acos_example, test_acos
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.4. Acosh

Calculates the hyperbolic arccosine of the given input tensor element-wise.

Domain	ai.onnx
Since version	22
Earlier versions	9
Inputs	input (differentiable) : T—Input tensor
Outputs	output (differentiable) : T—The hyperbolic arccosine values of the input tensor computed element-wise
Attributes	None.
Type constraints	T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain input and output types to float tensors.
Test vectors	test_acosh_example, test_acosh
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.5. Add

Performs element-wise binary addition (with Numpy-style broadcasting support).

This operator supports **multidirectional (i.e., Numpy-style) broadcasting**; for more details please check the doc.

(Opset 14 change): Extend supported types to include uint8, int8, uint16, and int16.

Domain	ai.onnx
Since version	14
Earlier versions	1, 6, 7, 13
Inputs	A (differentiable) : T — First operand. B (differentiable) : T — Second operand.
Outputs	C (differentiable) : T — Result, has same element type as two inputs
Attributes	None.
Type constraints	T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(float16), tensor(float), tensor(double), tensor(bfloat16) — Constrain input and output types to all numeric tensors.
Test vectors	test_add, test_add_int8, test_add_int16, test_add_uint8, test_add_uint16, test_add_uint32, test_add_uint64, test_add_bcast
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.6. AffineGrid

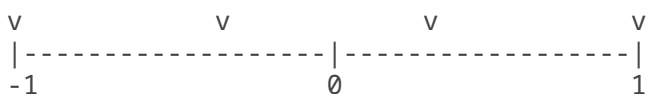
Generates a 2D or 3D flow field (sampling grid), given a batch of affine matrices theta (https://pytorch.org/docs/stable/generated/torch.nn.functional.affine_grid.html). An affine matrix theta is applied to a position tensor represented in its homogeneous expression. Here is an example in 3D:

$$\begin{bmatrix} r_{00} & r_{01} & r_{02} & t_0 \\ r_{10} & r_{11} & r_{12} & t_1 \\ r_{20} & r_{21} & r_{22} & t_2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix}$$

where (x, y, z) is the position in the original space, (x', y', z') is the position in the output space. The last row is always [0, 0, 0, 1] and is not stored in the affine matrix. Therefore we have theta of shape (N, 2, 3) for 2D or (N, 3, 4) for 3D.

Input size is used to define grid of positions evenly spaced in the original 2D or 3D space, with dimensions ranging from -1 to 1. The output grid contains positions in the output space.

When align_corners=1, consider -1 and 1 to refer to the centers of the corner pixels (mark v in illustration).



When align_corners=0, consider -1 and 1 to refer to the outer edge of the corner pixels.



	-1 0 1
Domain	ai.onnx
Since version	20
Inputs	theta (non-differentiable) : T1—input batch of affine matrices with shape (N, 2, 3) for 2D or (N, 3, 4) for 3D size (non-differentiable) : T2—the target output image size (N, C, H, W) for 2D or (N, C, D, H, W) for 3D
Outputs	grid (differentiable) : T1—output tensor of shape (N, H, W, 2) of 2D sample coordinates or (N, D, H, W, 3) of 3D sample coordinates.
Attributes	align_corners : int (default is 0)—if align_corners=1, consider -1 and 1 to refer to the centers of the corner pixels. if align_corners=0, consider -1 and 1 to refer to the outer edge the corner pixels.
Type constraints	T1 : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain grid types to float tensors. T2 : tensor(int64) — Constrain size's type to int64 tensors.
Test vectors	None published for this operator.
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.7. And

Returns the tensor resulted from performing the and logical operation elementwise on the input tensors A and B (with Numpy-style broadcasting support).

This operator supports **multidirectional (i.e., Numpy-style) broadcasting**; for more details please check the doc.

Domain	ai.onnx
Since version	7
Earlier versions	1
Inputs	A (non-differentiable) : T—First input operand for the logical operator. B (non-differentiable) : T—Second input operand for the logical operator.
Outputs	C (non-differentiable) : T1—Result tensor.
Attributes	None.
Type constraints	T : tensor(bool) — Constrain input to boolean tensor. T1 : tensor(bool) — Constrain output to boolean tensor.
Test vectors	test_and2d, test_and3d, test_and4d, test_and_bcast3v1d, test_and_bcast3v2d, test_and_bcast4v2d, test_and_bcast4v3d, test_and_bcast4v4d

Not supplied by the upstream source Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.8. ArgMax

Computes the indices of the max elements of the input tensor's element along the provided axis. The resulting tensor has the same rank as the input if keepdims equals 1. If keepdims equals 0, then the resulting tensor has the reduced dimension pruned. If select_last_index is True (default False), the index of the last occurrence of the max is selected if the max appears more than once in the input. Otherwise the index of the first occurrence is selected. The type of the output tensor is integer.

Domain	ai.onnx
Since version	13
Earlier versions	1, 11, 12
Inputs	data (non-differentiable) : T — An input tensor.
Outputs	reduced (non-differentiable) : tensor(int64) — Reduced output tensor with integer data type.
Attributes	axis : int (default is 0) — The axis in which to compute the arg indices. Accepted range is [-r, r-1] where r = rank(data). keepdims : int (default is 1) — Keep the reduced dimension or not, default 1 means keep reduced dimension. select_last_index : int (default is 0) — Whether to select the last index or the first index if the {name} appears in multiple indices, default is False (first index).
Type constraints	T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(float16), tensor(float), tensor(double), tensor(bfloat16) — Constrain input and output types to all numeric tensors.
Test vectors	test_argmax_default_axis_example, test_argmax_default_axis_random, test_argmax_default_axis_example_select_last_index, test_argmax_default_axis_random_select_last_index, test_argmax_keepdims_example, test_argmax_keepdims_random, test_argmax_keepdims_example_select_last_index, test_argmax_keepdims_random_select_last_index, test_argmax_negative_axis_keepdims_example, test_argmax_negative_axis_keepdims_random, test_argmax_negative_axis_keepdims_example_select_last_index, test_argmax_negative_axis_keepdims_random_select_last_index, test_argmax_no_keepdims_example, test_argmax_no_keepdims_random, test_argmax_no_keepdims_example_select_last_index, test_argmax_no_keepdims_random_select_last_index
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.9. ArgMin

Computes the indices of the min elements of the input tensor's element along the provided axis. The resulting tensor has the same rank as the input if keepdims equals 1. If keepdims equals 0, then the resulting tensor has the reduced dimension pruned. If select_last_index is True (default False), the index of the last occurrence of the min is selected if the min appears more than once in the input. Otherwise the index of the first occurrence is selected. The type of the output tensor is integer.

Domain	ai.onnx
Since version	13
Earlier versions	1, 11, 12
Inputs	data (non-differentiable) : T — An input tensor.
Outputs	reduced (non-differentiable) : tensor(int64) — Reduced output tensor with integer data type.
Attributes	axis : int (default is 0) — The axis in which to compute the arg indices. Accepted range is [-r, r-1] where r = rank(data). keepdims : int (default is 1) — Keep the reduced dimension or not, default 1 means keep reduced dimension. select_last_index : int (default is 0) — Whether to select the last index or the first index if the {name} appears in multiple indices, default is False (first index).
Type constraints	T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(float16), tensor(float), tensor(double), tensor(bfloat16) — Constrain input and output types to all numeric tensors.
Test vectors	test_argmin_default_axis_example, test_argmin_default_axis_random, test_argmin_default_axis_example_select_last_index, test_argmin_default_axis_random_select_last_index, test_argmin_keepdims_example, test_argmin_keepdims_random, test_argmin_keepdims_example_select_last_index, test_argmin_keepdims_random_select_last_index, test_argmin_negative_axis_keepdims_example, test_argmin_negative_axis_keepdims_random, test_argmin_negative_axis_keepdims_example_select_last_index, test_argmin_negative_axis_keepdims_random_select_last_index, test_argmin_no_keepdims_example, test_argmin_no_keepdims_random, test_argmin_no_keepdims_example_select_last_index, test_argmin_no_keepdims_random_select_last_index
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.10. Asin

Calculates the arcsine (inverse of sine) of the given input tensor, element-wise.

Domain	ai.onnx
Since version	22
Earlier versions	7
Inputs	input (differentiable) : T—Input tensor
Outputs	output (differentiable) : T—The arcsine of the input tensor computed element-wise
Attributes	None.
Type constraints	T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain input and output types to float tensors.
Test vectors	test_asin_example, test_asin
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.11. Asinh

Calculates the hyperbolic arcsine of the given input tensor element-wise.

Domain	ai.onnx
Since version	22
Earlier versions	9
Inputs	input (differentiable) : T—Input tensor
Outputs	output (differentiable) : T—The hyperbolic arcsine values of the input tensor computed element-wise
Attributes	None.
Type constraints	T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain input and output types to float tensors.
Test vectors	test_asinh_example, test_asinh
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.12. Atan

Calculates the arctangent (inverse of tangent) of the given input tensor, element-wise.

Domain	ai.onnx
Since version	22

Earlier versions	7
Inputs	input (differentiable) : T—Input tensor
Outputs	output (differentiable) : T—The arctangent of the input tensor computed element-wise
Attributes	None.
Type constraints	T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain input and output types to float tensors.
Test vectors	test_atan_example, test_atan
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.13. Atanh

Calculates the hyperbolic arctangent of the given input tensor element-wise.

Domain	ai.onnx
Since version	22
Earlier versions	9
Inputs	input (differentiable) : T—Input tensor
Outputs	output (differentiable) : T—The hyperbolic arctangent values of the input tensor computed element-wise
Attributes	None.
Type constraints	T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain input and output types to float tensors.
Test vectors	test_atanh_example, test_atanh
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.14. Attention

Computes scaled dot product attention on query, key and value tensors, using an optional attention mask if passed.

This operator covers self and cross variants of the attention operation based on sequence lengths of K, Q and V.

For self attention, kv_sequence_length equals to q_sequence_length.

For cross attention, query and key might have different lengths.

This operator also covers the 3 following variants based on the number of heads: 1) Multi-headed Attention (MHA): Described in the paper <https://arxiv.org/pdf/1706.03762>, $q_num_heads = kv_num_heads$. 2) Group-query Attention (GQA): Described in the paper <https://arxiv.org/pdf/2305.13245>, $q_num_heads > kv_num_heads$, $q_num_heads \% kv_num_heads == 0$. 3) Multi-query Attention (MQA): Described in the paper <https://arxiv.org/pdf/1911.02150>, $q_num_heads > kv_num_heads$, $kv_num_heads=1$.

Attention bias to be added is calculated based on `attn_mask` input and `is_causal` attribute: 1) `attn_mask`: A boolean mask where a value of True indicates that the element should take part in attention or a float mask of the same type as query, key, value that is added to the attention score. 2) If `is_causal` is set to 1, causal masking is applied with bottom-right (offset-aware) alignment: query i attends key j iff $j \leq i + \text{offset}$, as illustrated below.

2D causal mask for Attention (PR onnx/onnx#8068)

$S_q=4$ queries, $S_k=8$ keys

Rule: query i attends key j iff $j \leq i + \text{offset}$
 $\text{offset} = \text{nonpad_kv_seqlen} - S_q$

$\text{nonpad_kv_seqlen}=4$, $\text{offset}=4-4=0$

	k0	k1	k2	k3	k4	k5	k6	k7
q0	##							
q1	##	##						
q2	##	##	##					
q3	##	##	##	##				

$\text{nonpad_kv_seqlen}=8$, $\text{offset}=8-4=4$

	k0	k1	k2	k3	k4	k5	k6	k7
q0	##	##	##	##	##			
q1	##	##	##	##	##	##		
q2	##	##	##	##	##	##	##	
q3	##	##	##	##	##	##	##	##

With $\text{nonpad_kv_seqlen}=4$ ($\text{offset}=0$), the mask is the standard lower-triangular. With $\text{nonpad_kv_seqlen}=8$ ($\text{offset}=4$), the diagonal shifts right by 4, so each query sees the 4 additional valid cached keys.

`offset` is the count of valid keys preceding the current query block: $\text{offset} = \text{past_sequence_length}$ when `past_key` is provided; $\text{offset} = \text{nonpad_kv_seqlen} - q_sequence_length$ (per batch) when an external cache is indicated by `nonpad_kv_seqlen` without `past_key`; $\text{offset} = 0$ when neither is provided (the no-cache case, which reduces to the standard lower-triangular mask). When $\text{offset} < 0$ ($\text{nonpad_kv_seqlen} < q_sequence_length$, i.e. more query tokens than cached keys) the leading query rows have an empty key set (no key satisfies $j \leq i + \text{offset}$) and are fully masked. The causal frontier is computed independently of `attn_mask` and is then composed with it additively: a boolean `attn_mask` intersects the allowed set (its disallowed positions contribute $-\infty$ to the bias), while a float `attn_mask` is added to the attention scores rather than disabling positions. A fully-masked query row (no key attended, including the negative-offset leading rows) produces a zero output row, not NaN, for both Y and the mode-3 `qk_matmul_`

output debug output; the mode-3 qk_matmul_output is emitted at the operator's output precision (T1).

left_window_size and right_window_size independently restrict the keys visible to each query. A query at absolute position $p = \text{offset} + \text{query_index}$ attends keys j satisfying $p - \text{left_window_size} \leq j \leq p + \text{right_window_size}$ for each nonnegative bound. A value of -1 leaves that side unbounded. For example, (left_window_size=2, right_window_size=0) is a causal left-looking window containing the current key and two preceding keys, while (left_window_size=2, right_window_size=1) is an asymmetric bidirectional window. Window bounds are composed with is_causal and attn_mask; when is_causal=1, the causal upper bound still excludes future keys.

2D sliding-window mask for Attention (opset 25)

S_q=4 queries, S_k=6 keys, left_window_size=2, right_window_size=1, offset=0

	k0	k1	k2	k3	k4	k5
q0	##	##				
q1	##	##	##			
q2	##	##	##	##		
q3		##	##	##	##	

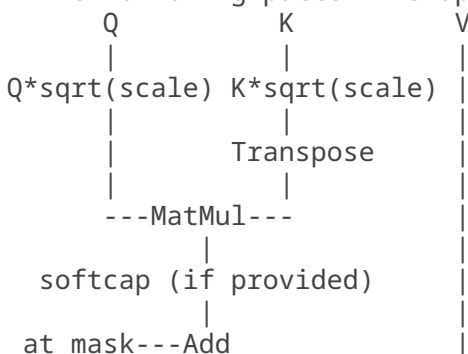
q0 attends {k0,k1}, q1 attends {k0,k1,k2}, q2 attends {k0,k1,k2,k3},
q3 attends {k1,k2,k3,k4}.

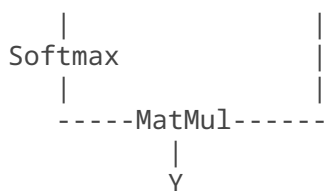
With respect to KV cache update, this operator allows the following two use cases:

1) Cache update happens inside the Attention operator. In this case, the K and V inputs contain only the incoming tokens for the current autoregressive step, and the four optional inputs/outputs past and present key and value are all needed. The Attention op performs a Concat operation on the past and incoming key and value to form the present key and value, respectively. Note that this only works correctly for the special case where the past key and value do not contain padded tokens. 2) Cache update happens outside the Attention operator (for example, through the TensorScatter operator). In this case, the K and V inputs correspond to the entire cache tensor, so the four optional inputs/outputs past and present key and value should not be used. An additional input nonpad_kv_seqlen of shape (batch_size,) may be provided to indicate the number of non-padding tokens in each sample of the batch to save unnecessary computation. Here, the kv_sequence dimension of attn_mask can be shorter than K and V, but still needs to be at least as long as the maximum value of nonpad_kv_seqlen.

Both past and present state key/values are optional. They shall be used together, and not allowed to use only one of them. The following pattern is applied to the Q, K and V inputs after appropriate reshaping of K and V inputs based on sequence lengths and num heads provided:

The following pattern is applied by this operator:





Domain	ai.onnx
Since version	25
Earlier versions	23, 24
Inputs (3 — 7)	Q : T1 — Query tensor. 4D tensor with shape (batch_size, q_num_heads, q_sequence_length, head_size) or 3D tensor with shape (batch_size, q_sequence_length, q_hidden_size). For cases with a 3D input tensor, $q_hidden_size = q_num_heads * head_size$ K : T1 — Key tensor. 4D tensor with shape (batch_size, kv_num_heads, kv_sequence_length, head_size) or 3D tensor with shape (batch_size, kv_sequence_length, k_hidden_size). For cases with a 3D input tensor, $k_hidden_size = kv_num_heads * head_size$ V : T2 — Value tensor. 4D tensor with shape (batch_size, kv_num_heads, kv_sequence_length, v_head_size) or 3D tensor with shape (batch_size, kv_sequence_length, v_hidden_size). For cases with a 3D input tensor, $v_hidden_size = kv_num_heads * v_head_size$ attn_mask (optional) : U — Attention mask. Shape must be broadcastable to (batch_size, q_num_heads, q_sequence_length, total_sequence_length) where $total_sequence_length = past_sequence_length + kv_sequence_length$. The last dimension can also be shorter than total_sequence_length and will be padded to total_sequence_length with negative infinity. Two types of masks are supported: a boolean mask where a value of True indicates that the element should take part in attention, or a float mask of the same type as query, key, value that is added to the attention score. past_key (optional) : T1 — Past state for key with shape (batch_size, kv_num_heads, past_sequence_length, head_size). Must be used together with past_value input. past_value (optional) : T2 — Past state for value with shape (batch_size, kv_num_heads, past_sequence_length, v_head_size). Must be used together with past_key input. nonpad_kv_seqlen (optional) : tensor(int64) — A vector of integers of shape (batch_size,) that indicates the number of valid (i.e., non-padding) tokens in each sample. A padding mask can be derived from this. This should not be used together with past_key and past_value inputs or present_key and present_value outputs (see the KV cache use cases in the operator description).
Outputs (1 — 4)	Y : T1 — The output tensor. 4D tensor with shape (batch_size, q_num_heads, q_sequence_length, v_head_size) or 3D tensor with shape (batch_size, q_sequence_length, hidden_size). For cases with a 3D input tensor, $hidden_size = q_num_heads * v_head_size$ present_key (optional) : T1 — Updated key cache with shape (batch_size, kv_num_heads, total_sequence_length, head_size) where $total_sequence_length = past_sequence_length + kv_sequence_length$. present_value (optional) : T2 — Updated value cache with shape (batch_size, kv_num_heads, total_sequence_length, v_head_size) where $total_sequence_length = past_sequence_length + kv_sequence_length$.

	qk_matmul_output (optional) : T1 — The output of QK matmul. 4D tensor with shape (batch_size, q_num_heads, q_sequence_length, total_sequence_length) where total_sequence_length = past_sequence_length + kv_sequence_length.
Attributes	is_causal : int (default is 0) — If set to 1, causal masking is applied. For a square Q/K (no cache offset) this is a lower-triangular matrix. In general the mask is bottom-right (offset-aware): query in-block index i attends key j iff $j \leq i + \text{offset}$, where offset is the count of valid keys preceding the query block (past_sequence_length for an internal past_key cache, or nonpad_kv_seqlen - q_sequence_length per batch for an external cache). When offset = 0 this reduces to the lower-triangular (top-left) mask. kv_num_heads : int — Number of heads of key and value. Must be used with 3D inputs of Q, K and V. left_window_size : int (default is -1) — Maximum number of positions to the left of the current absolute query position that may be attended. A value of 0 allows the current position but no preceding position, while -1 leaves the left side unbounded. This bound is composed with is_causal and attn_mask. q_num_heads : int — Number of heads of query. Must be used with 3D inputs of Q, K and V. qk_matmul_output_mode : int (default is 0) — Determines what the optional 4th output contains: 0 (default): raw QK matmul result; 1: after softcap (before bias addition); 2: QK + softcap + bias; 3: post-softmax probabilities (after fully-masked-row guard). In mode 3, a fully-masked query row (every key disallowed) is a zero row, consistent with the corresponding row of the primary output Y. The mode-3 output is emitted at the operator's output precision (T1); when softmax_precision differs from T1 this is a cast of the softmax result to T1. right_window_size : int (default is -1) — Maximum number of positions to the right of the current absolute query position that may be attended. A value of 0 allows the current position but no following position, while -1 leaves the right side unbounded. Set is_causal=0 to use a positive right window. scale : float — Scaling factor applied to $Q \cdot K^T$. Default value is $1/\sqrt{\text{head_size}}$. To prevent numerical overflow, scale Q, K by $\sqrt{\text{scale}}$ before matmul. softcap : float (default is 0.0) — Soft cap for attention logits, applied as $\text{softcap} * \tanh(\text{logits} / \text{softcap})$. Default value of 0.0 means no soft capping is applied. The soft cap is applied before mask / bias addition and softmax. softmax_precision : int — Specifies the precision for softmax computation. If provided, the attention weights will be cast to this type before softmax and then cast back to the original type. Supported values are: 1 (FLOAT), 10 (FLOAT16), 11 (DOUBLE), 16 (BFLOAT16).
Type constraints	T1 : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain Q and K inputs types to float tensors. T2 : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain V input types to float tensors. U : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(bool) — Constrain output 'mask' types to boolean tensors and input types.
Test vectors	test_attention_4d, test_attention_23_boolmask_fullymasked_row_nan_robustness, test_attention_23_fullymasked_qk_matmul_output_mode3_zero, test_attention_24_fullymasked_qk_matmul_output_

mode3_zero, test_attention_24_qk_matmul_output_mode3_softmax_precision, test_attention_3d, test_attention_3d_attn_mask, test_attention_3d_causal, test_attention_3d_causal_bf16, test_attention_3d_diff_heads_sizes, test_attention_3d_diff_heads_sizes_attn_mask, test_attention_3d_diff_heads_sizes_causal, test_attention_3d_diff_heads_sizes_scaled, test_attention_3d_diff_heads_sizes_softcap, test_attention_3d_diff_heads_with_past_and_present, test_attention_3d_gqa, test_attention_3d_gqa_attn_mask, test_attention_3d_gqa_causal, test_attention_3d_gqa_scaled, test_attention_3d_gqa_softcap, test_attention_3d_gqa_with_past_and_present, test_attention_3d_local_window, test_attention_3d_scaled, test_attention_3d_softcap, test_attention_3d_transpose_verification, test_attention_3d_with_past_and_present, test_attention_3d_with_past_and_present_qk_matmul, test_attention_3d_with_past_and_present_qk_matmul_bias, test_attention_3d_with_past_and_present_qk_matmul_softcap, test_attention_3d_with_past_and_present_qk_matmul_softmax, test_attention_4d_causal_nonpad_attn_mask_composition, test_attention_4d_causal_nonpad_batch_prefill, test_attention_4d_causal_nonpad_continued_prefill, test_attention_4d_causal_nonpad_negative_offset_structural_empty, test_attention_4d_causal_with_past_and_present, test_attention_4d_diff_heads_mask4d_padded_kv, test_attention_4d_gqa_causal_nonpad_decode, test_attention_4d_gqa_causal_nonpad_decode_fp16, test_attention_4d_attn_mask_3d, test_attention_4d_attn_mask_3d_causal, test_attention_4d_attn_mask_4d, test_attention_4d_attn_mask_4d_causal, test_attention_4d_attn_mask, test_attention_4d_attn_mask_bool, test_attention_4d_attn_mask_bool_4d, test_attention_4d_attn_mask_causal_bf16, test_attention_bidirectional_window, test_attention_4d_causal, test_attention_4d_causal_bf16, test_attention_causal_boolmask_nan_robustness, test_attention_4d_causal_fp16, test_attention_4d_causal_padded_kv_bf16, test_attention_4d_diff_heads_sizes, test_attention_4d_diff_heads_sizes_attn_mask, test_attention_4d_diff_heads_sizes_causal, test_attention_4d_diff_heads_sizes_scaled, test_attention_4d_diff_heads_sizes_softcap, test_attention_4d_diff_heads_with_past_and_present, test_attention_4d_diff_heads_with_past_and_present_mask3d, test_attention_4d_diff_heads_with_past_and_present_mask4d, test_attention_4d_fp16, test_attention_4d_gqa, test_attention_4d_gqa_attn_mask, test_attention_4d_gqa_causal, test_attention_4d_gqa_scaled, test_attention_4d_gqa_softcap, test_attention_4d_gqa_with_past_and_present, test_attention_4d_gqa_with_past_and_present_fp16, test_attention_local_window, test_attention_local_window_default, test_attention_local_window_ext_cache_float16_mask, test_attention_local_window_ext_cache_rank2_mask, test_attention_local_window_ext_cache_rank3_head_mask, test_attention_local_window_ext_cache_rank4_batch_mask, test_attention_local_window_gqa_rank4_mask, test_attention_local_window_rank1_boolean_mask, test_attention_local_window_with_past, test_attention_4d_padded_kv_bf16, test_attention_4d_scaled, test_attention_4d_softcap, test_attention_4d_softcap_neginf_mask, test_attention_4d_softcap_neginf_mask_poison, test_attention_4d_with_past_and_present, test_attention_4d_with_past_and_present_qk_matmul, test_attention_4d_with_past_and_present_qk_matmul_bias, test_attention_4d_with_past_and_present_qk_matmul_bias_3d_mask, test_attention_4d_with_past_and_present_qk_matmul_bias_3d_mask_causal, test_attention_4d_with_past_and_present_qk_matmul_bias_4d_mask, test_attention_4d_with_past_and_present_qk_matmul_bias_4d_mask_causal, test_attention_4d_with_qk_matmul, test

attention_4d_with_qk_matmul_bias, test_attention_4d_with_qk_matmul_softcapi, test_attention_4d_with_qk_matmul_softmax

Not
supplied
by the
upstream
source

Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.15. AveragePool

AveragePool consumes an input tensor X and applies average pooling across the tensor according to kernel sizes, stride sizes, and pad lengths. average pooling consisting of computing the average on all values of a subset of the input tensor according to the kernel size and downsampling the data into the output tensor Y for further processing. The output spatial shape is calculated differently depending on whether explicit padding is used, where pads is employed, or auto padding is used, where auto_pad is utilized. With explicit padding (<https://pytorch.org/docs/stable/generated/torch.nn.MaxPool2d.html?highlight=maxpool#torch.nn.MaxPool2d>):

```
output_spatial_shape[i] = floor((input_spatial_shape[i] + pad_shape[i] -
dilation[i] * (kernel_shape[i] - 1) - 1) / strides_spatial_shape[i] + 1)
```

or

```
output_spatial_shape[i] = ceil((input_spatial_shape[i] + pad_shape[i] -
dilation[i] * (kernel_shape[i] - 1) - 1) / strides_spatial_shape[i] + 1)
```

if ceil_mode is enabled. pad_shape[i] is the sum of pads along axis i. Sliding windows that would start in the right padded region are ignored.

auto_pad is a DEPRECATED attribute. If you are using them currently, the output spatial shape will be following when ceil_mode is enabled:

```
VALID: output_spatial_shape[i] = ceil((input_spatial_shape[i] - ((kernel_
spatial_shape[i] - 1) * dilations[i] + 1) + 1) / strides_spatial_shape[i])
SAME_UPPER or SAME_LOWER: output_spatial_shape[i] = ceil(input_spatial_shape[i]
/ strides_spatial_shape[i])
```

or when ceil_mode is disabled (https://www.tensorflow.org/api_docs/python/tf/keras/layers/AveragePooling2D):

```
VALID: output_spatial_shape[i] = floor((input_spatial_shape[i] - ((kernel_
spatial_shape[i] - 1) * dilations[i] + 1)) / strides_spatial_shape[i]) + 1
SAME_UPPER or SAME_LOWER: output_spatial_shape[i] = floor((input_spatial_
shape[i] - 1) / strides_spatial_shape[i]) + 1
```

And pad shape will be following if SAME_UPPER or SAME_LOWER:

```
pad_shape[i] = (output_spatial_shape[i] - 1) * strides_spatial_shape[i] +
((kernel_spatial_shape[i] - 1) * dilations[i] + 1) - input_spatial_shape[i]
```

The output of each pooling window is divided by the number of elements (exclude pad when attribute count_include_pad is zero).

Domain	ai.onnx
Since version	22
Earlier versions	1, 7, 10, 11, 19
Inputs	X (differentiable) : T—Input data tensor from the previous operator; dimensions for image case are (N x C x H x W), where N is the

batch size, C is the number of channels, and H and W are the height and the width of the data. For non image case, the dimensions are in the form of (N x C x D1 x D2 ... Dn), where N is the batch size. Optionally, if dimension denotation is in effect, the operation expects the input data tensor to arrive with the dimension denotation of [DATA_BATCH, DATA_CHANNEL, DATA_FEATURE, DATA_FEATURE ...].

Outputs	Y (differentiable) : T— Output data tensor from average or max pooling across the input tensor. Dimensions will vary based on various kernel, stride, and pad sizes. Floor value of the dimension is used
Attributes	auto_pad : string (default is NOTSET) — auto_pad must be either NOTSET, SAME_UPPER, SAME_LOWER or VALID. Where default value is NOTSET, which means explicit padding is used. SAME_UPPER or SAME_LOWER mean pad the input so that $\text{output_shape}[i] = \text{ceil}(\text{input_shape}[i] / \text{strides}[i])$ for each axis i. The padding is split between the two sides equally or almost equally (depending on whether it is even or odd). In case the padding is an odd number, the extra padding is added at the end for SAME_UPPER and at the beginning for SAME_LOWER. ceil_mode : int (default is 0) — Whether to use ceil or floor (default) to compute the output shape. count_include_pad : int (default is 0) — Whether include pad pixels when calculating values for the edges. Default is 0, doesn't count include pad. dilations : list of ints — Dilation value along each spatial axis of filter. If not present, the dilation defaults to 1 along each spatial axis. kernel_shape : list of ints (required) — The size of the kernel along each axis. pads : list of ints — Padding for the beginning and ending along each spatial axis, it can take any value greater than or equal to 0. The value represent the number of pixels added to the beginning and end part of the corresponding axis. pads format should be as follow [x1_begin, x2_begin...x1_end, x2_end,...], where xi_begin the number of pixels added at the beginning of axis i and xi_end, the number of pixels added at the end of axis i. This attribute cannot be used simultaneously with auto_pad attribute. If not present, the padding defaults to 0 along start and end of each spatial axis. strides : list of ints — Stride along each spatial axis. If not present, the stride defaults to 1 along each spatial axis.
Type constraints	T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain input and output types to float tensors.
Test vectors	test_averagepool_1d_default, test_averagepool_2d_ceil, test_averagepool_2d_ceil_last_window_starts_on_pad, test_averagepool_2d_default, test_averagepool_2d_dilations, test_averagepool_2d_pads, test_averagepool_2d_pads_count_include_pad, test_averagepool_2d_precomputed_pads, test_averagepool_2d_precomputed_pads_count_include_pad, test_averagepool_2d_precomputed_same_upper, test_averagepool_2d_precomputed_strides, test_averagepool_2d_same_lower, test_averagepool_2d_same_upper, test_averagepool_2d_strides, test_averagepool_3d_default, test_averagepool_3d_dilations_small
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.16. BatchNormalization

Carries out batch normalization as described in the paper <https://arxiv.org/abs/1502.03167>. Depending on the mode it is being run, There are five required inputs 'X', 'scale', 'B', 'input_mean' and 'input_var'. Note that 'input_mean' and 'input_var' are expected to be the estimated statistics in inference mode (training_mode=False, default), and the running statistics in training mode (training_mode=True). There are multiple cases for the number of outputs, which we list below:

- Output case #1: Y, running_mean, running_var (training_mode=True)
- Output case #2: Y (training_mode=False)

When training_mode=False, extra outputs are invalid. The outputs are updated as follows when training_mode=True:

```
running_mean = input_mean * momentum + current_mean * (1 - momentum)
running_var = input_var * momentum + current_var * (1 - momentum)
```

```
Y = (X - current_mean) / sqrt(current_var + epsilon) * scale + B
```

where:

```
current_mean = ReduceMean(X, axis=all_except_channel_index)
current_var = ReduceVar(X, axis=all_except_channel_index)
```

Notice that ReduceVar refers to the population variance, and it equals to $\text{sum}(\text{sqr}d(x_i - x_{\text{avg}})) / N$ where N is the population size (this formula does not use sample size $N - 1$).

The computation of ReduceMean and ReduceVar uses float to avoid overflow for float16 inputs.

When training_mode=False:

```
Y = (X - input_mean) / sqrt(input_var + epsilon) * scale + B
```

For previous (deprecated) non-spatial cases, implementors are suggested to flatten the input shape to $(N \times C \times D1 \times D2 \times \dots \times Dn)$ before a BatchNormalization Op. This operator has **optional** inputs/outputs. See the doc for more details about the representation of optional arguments. An empty string may be used in the place of an actual argument's name to indicate a missing argument. Trailing optional arguments (those not followed by an argument that is present) may also be simply omitted.

Domain	ai.onnx
Since version	15
Earlier versions	1, 6, 7, 9, 14
Inputs	X (differentiable) : T — Input data tensor from the previous operator; dimensions are in the form of $(N \times C \times D1 \times D2 \dots Dn)$, where N is the batch size, C is the number of channels. Statistics are computed for every channel of C over N and D1 to Dn dimensions. For image data, input dimensions become $(N \times C \times H \times W)$. The op also accepts single dimension input of size N in which case C is assumed to be 1 scale (differentiable) : T1 — Scale tensor of shape @. B (differentiable) : T1 — Bias tensor of shape @. input_mean (differentiable) : T2 — running (training) or estimated (testing) mean tensor of shape @. input_var (differentiable) : T2 — running (training) or estimated (testing) variance tensor of shape @.

Outputs (1 — 3)	Y (differentiable) : T — The output tensor of the same shape as X running_mean (optional, non-differentiable) : T2 — The running mean after the BatchNormalization operator. running_var (optional, non-differentiable) : T2 — The running variance after the BatchNormalization operator. This op uses the population size (N) for calculating variance, and not the sample size N-1.
Attributes	epsilon : float (default is 1e-05) — The epsilon value to use to avoid division by zero. momentum : float (default is 0.9) — Factor used in computing the running mean and variance.e.g., $\text{running_mean} = \text{running_mean} * \text{momentum} + \text{mean} * (1 - \text{momentum})$. training_mode : int (default is 0) — If set to true, it indicates BatchNormalization is being used for training, and outputs 1 and 2 are to be computed.
Type constraints	T : tensor(float16), tensor(float), tensor(double), tensor(bfloat16) — Constrain input and output types to float tensors. T1 : tensor(float16), tensor(float), tensor(double), tensor(bfloat16) — Constrain scale and bias types to float tensors. T2 : tensor(float16), tensor(float), tensor(double), tensor(bfloat16) — Constrain mean and variance types to float tensors.
Test vectors	test_batchnorm_example, test_batchnorm_epsilon, test_batchnorm_example_training_mode, test_batchnorm_epsilon_training_mode
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.17. Bernoulli

Draws binary random numbers (0 or 1) from a Bernoulli distribution. The input tensor should be a tensor containing probabilities p (a value in the range [0,1]) to be used for drawing the binary random number, where an output of 1 is produced with probability p and an output of 0 is produced with probability (1-p).

This operator is non-deterministic and may not produce the same values in different implementations (even if a seed is specified).

Domain	ai.onnx
Since version	22
Earlier versions	15
Inputs	input : T1 — All values in input have to be in the range:[0, 1].
Outputs	output : T2 — The returned output tensor only has values 0 or 1, same shape as input tensor.
Attributes	dtype : int — The data type for the elements of the output tensor. if not specified, we will use the data type of the input tensor. seed : float — (Optional) Seed to the random generator, if not specified we will auto generate one.

Type constraints	T1 : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain input types to float tensors. T2 : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(bool) — Constrain output types to all numeric tensors and bool tensors.
Test vectors	test_bernoulli_double, test_bernoulli_seed, test_bernoulli
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.18. BitCast

Reinterprets the binary representation of a tensor as a different data type, specified by the 'to' attribute. Unlike Cast, BitCast preserves the exact bit pattern without any value conversion.

The target data type must have the same bit-width as the input data type. The output tensor has the same shape as the input tensor. All types except string are supported. Implementations must treat the underlying bytes as little endian.

Domain	ai.onnx
Since version	26
Inputs	input (non-differentiable) : T1 — Input tensor to be bitcast.
Outputs	output (non-differentiable) : T2 — Output tensor with the same shape as the input.
Attributes	to : int (required) — The data type to which the input tensor is bitwise reinterpreted. Must be one of the non-string types from DataType enum in TensorProto. The target type must have the same bit-width as the input type.
Type constraints	T1 : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(bool), tensor(complex64), tensor(complex128), tensor(float8e4m3fn), tensor(float8e4m3fnuz), tensor(float8e5m2), tensor(float8e5m2fnuz), tensor(uint4), tensor(int4), tensor(float4e2m1), tensor(float8e8m0), tensor(uint2), tensor(int2) — Constrain input types. Bitcasting from string is not supported. T2 : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(bool), tensor(complex64), tensor(complex128), tensor(float8e4m3fn), tensor(float8e4m3fnuz), tensor(float8e5m2), tensor(float8e5m2fnuz), tensor(uint4), tensor(int4), tensor(float4e2m1), tensor(float8e8m0), tensor(uint2), tensor(int2) — Constrain output types. Bitcasting to string is not supported.

Test vectors `test_bitcast_2d_float32_to_int32, test_bitcast_bool_to_uint8, test_bitcast_float32_to_int32, test_bitcast_float64_to_int64, test_bitcast_int32_to_float32, test_bitcast_int64_to_float64, test_bitcast_int8_to_uint8, test_bitcast_scalar_float32_to_int32, test_bitcast_uint16_to_int16, test_bitcast_uint32_to_int32`

Not supplied by the upstream source Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.19. BitShift

Bitwise shift operator performs element-wise operation. For each input element, if the attribute “direction” is “RIGHT”, this operator moves its binary representation toward the right side. If the attribute “direction” is “LEFT”, bits of binary representation move toward the left side. The input X is the tensor to be shifted and another input Y specifies the amounts of shifting. For example, if “direction” is “RIGHT”, X is [1, 4], and Y is [1, 1], the corresponding output Z would be [0, 2]. If “direction” is “LEFT” with X=[1, 2] and Y=[1, 2], the corresponding output Z would be [2, 8].

For a signed T the right shift is an arithmetic shift (sign-extending). The vacated high bits are filled with copies of the sign bit, so a negative X stays negative. For a signed T a left shift can move bits into and past the sign bit, and bits shifted past the sign bit are discarded.

If Y is negative, or is greater than or equal to the number of bits of T, then the result is whatever the sign bit extension alone produces: -1 for a right shift on a negative X, where the fill is a sign bit of 1, and 0 in every other case. This operator supports **multidirectional (i.e., Numpy-style) broadcasting**; for more details please check the doc.

Domain	<code>ai.onnx</code>
Since version	28
Earlier versions	11
Inputs	X (non-differentiable) : T — First operand, input to be shifted. Y (non-differentiable) : T — Second operand, amounts of shift.
Outputs	Z (non-differentiable) : T — Output tensor
Attributes	direction : string (required) — Direction of moving bits. It can be either “RIGHT” (for right shift) or “LEFT” (for left shift).
Type constraints	T : <code>tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64)</code> — Constrain input and output types to integer tensors.
Test vectors	<code>test_bitshift_left_int16, test_bitshift_left_int32, test_bitshift_left_int32_negative_shift, test_bitshift_left_int32_overflow, test_bitshift_left_int32_shift_ge_width, test_bitshift_left_int64, test_bitshift_left_int8, test_bitshift_left_int8_negative_shift, test_bitshift_left_int8_overflow, test_bitshift_left_int8_shift_ge_width, test_bitshift_left_uint16, test_bitshift_left_uint32, test_bitshift_left_uint64, test_bitshift_left_uint8, test_bitshift_right_int16, test_bitshift_right_int32, test_bitshift_right_int32_negative_input, test_bitshift_right_int32_negative_shift, test_bitshift_right_int32_shift_ge_width, test_bitshift_right_int64, test_bitshift_</code>

```
right_int8, test_bitshift_right_int8_negative_input, test_
bitshift_right_int8_negative_shift, test_bitshift_right_int8_
shift_ge_width, test_bitshift_right_uint16, test_bitshift_right_
uint32, test_bitshift_right_uint64, test_bitshift_right_uint8
```

Not
supplied
by the
upstream
source

Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.20. BitwiseAnd

Returns the tensor resulting from performing the bitwise and operation elementwise on the input tensors A and B (with Numpy-style broadcasting support).

This operator supports **multidirectional (i.e., Numpy-style) broadcasting**; for more details please check the doc.

Domain	ai.onnx
Since version	18
Inputs	A (non-differentiable) : T — First input operand for the bitwise operator. B (non-differentiable) : T — Second input operand for the bitwise operator.
Outputs	C (non-differentiable) : T — Result tensor.
Attributes	None.
Type constraints	T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64) — Constrain input to integer tensors.
Test vectors	test_bitwise_and_i32_2d, test_bitwise_and_i16_3d, test_bitwise_and_ui64_bcast_3v1d, test_bitwise_and_ui8_bcast_4v3d
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.21. BitwiseNot

Returns the bitwise not of the input tensor element-wise.

Domain	ai.onnx
Since version	18
Inputs	X (non-differentiable) : T — Input tensor
Outputs	Y (non-differentiable) : T — Output tensor
Attributes	None.

Type constraints	T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64) — Constrain input/output to integer tensors.
Test vectors	test_bitwise_not_2d, test_bitwise_not_3d, test_bitwise_not_4d
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.22. BitwiseOr

Returns the tensor resulting from performing the bitwise or operation elementwise on the input tensors A and B (with Numpy-style broadcasting support).

This operator supports **multidirectional (i.e., Numpy-style) broadcasting**; for more details please check the doc.

Domain	ai.onnx
Since version	18
Inputs	A (non-differentiable) : T — First input operand for the bitwise operator. B (non-differentiable) : T — Second input operand for the bitwise operator.
Outputs	C (non-differentiable) : T — Result tensor.
Attributes	None.
Type constraints	T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64) — Constrain input to integer tensors.
Test vectors	test_bitwise_or_i32_2d, test_bitwise_or_i16_4d, test_bitwise_or_ui64_bcast_3v1d, test_bitwise_or_ui8_bcast_4v3d
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.23. BitwiseXor

Returns the tensor resulting from performing the bitwise xor operation elementwise on the input tensors A and B (with Numpy-style broadcasting support).

This operator supports **multidirectional (i.e., Numpy-style) broadcasting**; for more details please check the doc.

Domain	ai.onnx
Since version	18

Inputs	A (non-differentiable) : T — First input operand for the bitwise operator. B (non-differentiable) : T — Second input operand for the bitwise operator.
Outputs	C (non-differentiable) : T — Result tensor.
Attributes	None.
Type constraints	T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64) — Constrain input to integer tensors.
Test vectors	test_bitwise_xor_ui64_bcast_3v1d, test_bitwise_xor_ui8_bcast_4v3d, test_bitwise_xor_i32_2d, test_bitwise_xor_i16_3d
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.24. BlackmanWindow

Generates a Blackman window as described in the paper <https://ieeexplore.ieee.org/document/1455106>.

Domain	ai.onnx
Since version	17
Inputs	size (non-differentiable) : T1 — A scalar value indicating the length of the window.
Outputs	output (non-differentiable) : T2 — A Blackman window with length: size. The output has the shape: [size].
Attributes	output_datatype : int (default is 1) — The data type of the output tensor. Strictly must be one of the values from DataType enum in TensorProto whose values correspond to T2. The default value is 1 = FLOAT. periodic : int (default is 1) — If 1, returns a window to be used as periodic function. If 0, return a symmetric window. When 'periodic' is specified, hann computes a window of length size + 1 and returns the first size points. The default value is 1.
Type constraints	T1 : tensor(int32), tensor(int64) — Constrain the input size to int32_t or int64_t. T2 : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(float16), tensor(float), tensor(double), tensor(bfloat16) — Constrain output types to numeric tensors.
Test vectors	test_blackmanwindow, test_blackmanwindow_symmetric
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.25. Cast

The operator casts the elements of a given input tensor to a data type specified by the 'to' argument and returns an output tensor of the same size in the converted type. The 'to' argument must be one of the data types specified in the 'DataType' enum field in the TensorProto message.

Casting from string tensor in plain (e.g., "3.14" and "1000") and scientific numeric representations (e.g., "1e-5" and "1E8") to float types is supported. For example, converting string "100.5" to an integer may yield result 100. There are some string literals reserved for special floating-point values; "+INF" (and "INF"), "-INF", and "NaN" are positive infinity, negative infinity, and not-a-number, respectively. Any string which can exactly match "+INF" in a case-insensitive way would be mapped to positive infinite. Similarly, this case-insensitive rule is applied to "INF" and "NaN". When casting from numeric tensors to string tensors, plain floating-point representation (such as "314.15926") would be used. Converting non-numerical-literal string such as "Hello World!" is an undefined behavior. Cases of converting string representing floating-point arithmetic value, such as "2.718", to INT is an undefined behavior.

Conversion from a numerical type to any numerical type is always allowed. User must be aware of precision loss and value change caused by range difference between two types. For example, a 64-bit float 3.1415926459 may be round to a 32-bit float 3.141592. Similarly, converting an integer 36 to Boolean may produce 1 because we truncate bits which can't be stored in the targeted type.

In more detail, the conversion among numerical types should follow these rules if the destination type is not a float 8 type.

- Casting from floating point to:
 - floating point: +/- infinity if OOR (out of range).
 - fixed point: undefined if OOR.
 - bool: +/- 0.0 to False; all else to True.
- Casting from fixed point to:
 - floating point: /- infinity if OOR. (infinity in the case of uint)
 - fixed point: when OOR, discard higher bits and reinterpret (with respect to two's complement representation for signed types). For example, 200 (int16) -> -56 (int8).
 - bool: zero to False; nonzero to True.
- Casting from bool to:
 - floating point: {1.0, 0.0}.
 - fixed point: {1, 0}.
 - bool: no change.

Float 8 types (E4M3FN, E4M3FNUZ, E5M2, E5M2FNUZ) were introduced to speed up the training of deep models. By default the conversion of a float x obeys to the following rules. $[x]$ means the value rounded to the target mantissa width.

Table 2 — Table from the upstream description of Cast

x	E4M3FN	E4M3FNUZ	E5M2	E5M2FNUZ
0	0	0	0	0
-0	-0	0	-0	0
NaN	NaN	NaN	NaN	NaN
Inf	FLT_MAX	FLT_MAX	FLT_MAX	FLT_MAX
-Inf	-FLT_MAX	-FLT_MAX	-FLT_MAX	-FLT_MAX
$[x] > \text{FLT_MAX}$	FLT_MAX	FLT_MAX	FLT_MAX	FLT_MAX
$[x] < -\text{FLT_MAX}$	-FLT_MAX	-FLT_MAX	-FLT_MAX	-FLT_MAX
else	RNE	RNE	RNE	RNE

The behavior changes if the parameter 'saturate' is set to False. The rules then become:

Table 3 — Table from the upstream description of Cast

x	E4M3FN	E4M3FNUZ	E5M2	E5M2FNUZ
0	0	0	0	0
-0	-0	0	-0	0
NaN	NaN	NaN	NaN	NaN
-NaN	-NaN	NaN	-NaN	NaN
Inf	NaN	NaN	Inf	NaN
-Inf	-NaN	NaN	-Inf	NaN
[x] > FLT_MAX	NaN	NaN	Inf	NaN
[x] < -FLT_MAX	NaN	NaN	-Inf	NaN
else	RNE	RNE	RNE	RNE

FLOAT8E8M0 type was introduced to enable [Microscaling \(MX\) formats](#). When casting to FLOAT8E8M0, the rounding behavior can be specified using the `round_mode` and `saturate` attributes. The current CUDA behavior is to round up and saturate. Casting negative values to FLOAT8E8M0 gives undefined behavior. The following table describes the casting behavior of special values to FLOAT8E8M0 in the two most common cases.

Table 4 — Table from the upstream description of Cast

x	saturate + up	non-saturate + nearest
0	0	NaN
-0	Unspecified	Unspecified
NaN	NaN	NaN
Inf	E8M0_MAX	NaN
x > E8M0_MAX	E8M0_MAX	NaN
x < E8M0_MIN	E8M0_MIN	NaN
x < 0	Unspecified	Unspecified
Domain	ai.onnx	
Since version	28	
Earlier versions	1, 6, 9, 13, 19, 21, 23, 24, 25	
Inputs	input (differentiable) : T1—Input tensor to be cast.	
Outputs	output (differentiable) : T2—Output tensor with the same shape as input with type specified by the 'to' argument	
Attributes	<code>round_mode</code> : string (default is up) — Rounding mode for conversion to float8e8m0. It only applies to casting to float8e8m0 and is up by default. up: round to nearest value away from zero, down: round to nearest value towards zero, nearest: round to nearest value and ties round up. <code>saturate</code> : int (default is 1) — The parameter defines how the conversion behaves if an input value is out of range of the destination type. It only applies for float 8 conversion (float8e4m3fn, float8e4m3fnuz, float8e5m2, float8e5m2fnuz, float8e8m0). It is true by default. All cases are fully described in the tables inserted in the operator description. <code>to</code> : int (required) — The data type to which the elements of the input tensor are cast. Strictly must be one of the types from <code>DataType</code> enum in <code>TensorProto</code>	
Type constraints	T1 : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(float8e4m3fn), tensor(float8e4m3fnuz), tensor(float8e5m2), tensor(float8e5m2fnuz), tensor(uint4), tensor(int4), tensor(float4e2m1), tensor(float8e8m0), tensor(uint2), tensor(int2), tensor(float6e2m3),	

tensor(float6e3m2) — Constrain input types. Casting from complex is not supported.

T2 : tensor(uint8), tensor(uint16), tensor(uint32),
 tensor(uint64), tensor(int8), tensor(int16), tensor(int32),
 tensor(int64), tensor(bfloat16), tensor(float16),
 tensor(float), tensor(double), tensor(string),
 tensor(bool), tensor(float8e4m3fn), tensor(float8e4m3fnuz),
 tensor(float8e5m2), tensor(float8e5m2fnuz), tensor(uint4),
 tensor(int4), tensor(float4e2m1), tensor(float8e8m0),
 tensor(uint2), tensor(int2), tensor(float6e2m3),
 tensor(float6e3m2) — Constrain output types. Casting to complex is not supported.

Test vectors test_cast_, test_cast_e8m0_, test_cast_no_saturate_

Not supplied by the upstream source Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.26. CastLike

The operator casts the elements of a given input tensor (the first input) to the same data type as the elements of the second input tensor. See documentation of the Cast operator for further details.

Domain ai.onnx

Since version 25

Earlier versions 15, 19, 21, 23, 24

Inputs input (differentiable) : T1 — Input tensor to be cast.
 target_type (non-differentiable) : T2 — The (first) input tensor will be cast to produce a tensor of the same type as this (second input) tensor.

Outputs output (differentiable) : T2 — Output tensor produced by casting the first input tensor to have the same type as the second input tensor.

Attributes round_mode : string (default is up) — Rounding mode for conversion to float8e8m0. It only applies to casting to float8e8m0 and is up by default. up: round to nearest value away from zero, down: round to nearest value towards zero, nearest: round to nearest value and ties round up. Please refer to operator Cast description for further details.
 saturate : int (default is 1) — The parameter defines how the conversion behaves if an input value is out of range of the destination type. It only applies for float 8 conversion (float8e4m3fn, float8e4m3fnuz, float8e5m2, float8e5m2fnuz, float8e8m0). It is true by default. Please refer to operator Cast description for further details.

Type constraints T1 : tensor(uint8), tensor(uint16), tensor(uint32),
 tensor(uint64), tensor(int8), tensor(int16), tensor(int32),
 tensor(int64), tensor(bfloat16), tensor(float16),
 tensor(float), tensor(double), tensor(string),
 tensor(bool), tensor(float8e4m3fn), tensor(float8e4m3fnuz),
 tensor(float8e5m2), tensor(float8e5m2fnuz), tensor(uint4),
 tensor(int4), tensor(float4e2m1), tensor(float8e8m0),
 tensor(uint2), tensor(int2) — Constrain input types. Casting from complex is not supported.

T2 : tensor(uint8), tensor(uint16), tensor(uint32),
 tensor(uint64), tensor(int8), tensor(int16), tensor(int32),
 tensor(int64), tensor(bfloat16), tensor(float16),
 tensor(float), tensor(double), tensor(string),
 tensor(bool), tensor(float8e4m3fn), tensor(float8e4m3fnuz),
 tensor(float8e5m2), tensor(float8e5m2fnuz), tensor(uint4),
 tensor(int4), tensor(float4e2m1), tensor(float8e8m0),
 tensor(uint2), tensor(int2) — Constrain output types. Casting to
 complex is not supported.

Test vectors test_castlike_, test_castlike_no_saturate_

Not supplied by the upstream source Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.27. CausalConvWithState

Stateful causal 1D depthwise convolution.

Used by Gated DeltaNet (Qwen3.5) and Mamba (Jamba, FalconMamba) as a preprocessing step. Replaces the 3-op pattern (Concat + Conv + Slice) with a single fused operation.

The convolution is causal (looks only at current and past positions) and depthwise (each channel is convolved independently with its own kernel).

The input, weight, past_state, output, and present_state tensors are rank-3 with shape (batch_size, channels, length). The optional bias input is rank-1 with shape (channels). For higher-dimensional data, use Reshape nodes before and after this operator to pack extra dimensions into the batch or channel axis.

Weight layout: (channels, 1, k) for depthwise convolution. The carry state stores the last (k-1) positions for incremental decode.

The optional activation attribute supports fused SiLU/Swish activation.

Domain ai.onnx

Since version 27

Inputs (2 — 4) input (differentiable) : T — Input tensor with shape (batch_size, channels, length). Channels-first layout.
 weight (differentiable) : T — Depthwise convolution kernel with shape (channels, 1, k) where k is the kernel size. The middle dim of size 1 follows the ONNX Conv weight layout (M, C/group, k1, ..., kn): since this op is always depthwise, group = channels, so C/group = 1. Keeping this layout makes the weight tensor a drop-in for a depthwise Conv(group=channels) weight, so Conv CausalConvWithState rewrites require no reshape.
 bias (optional, differentiable) : T — Optional per-channel bias with shape (channels).
 past_state (optional, non-differentiable) : T — Carry state from previous step with shape (batch_size, channels, k — 1). If not provided, padding is zero.

Outputs output (differentiable) : T — Convolution output with same shape as input.

	<code>present_state</code> (non-differentiable) : T — Updated carry state with shape (batch_size, channels, $k - 1$). Contains the last ($k - 1$) values of the effective padded/concatenated sequence along the causal axis, including any values from <code>past_state</code> or zero-padding when the current input is shorter than $k - 1$.
Attributes	<code>activation</code> : string (default is none) — Fused activation function. One of: 'silu', 'swish', 'none'. Default is 'none'.
Type constraints	T : <code>tensor(float)</code> , <code>tensor(float16)</code> , <code>tensor(bfloat16)</code> — Constrain input and output types to float tensors.
Test vectors	<code>test_causal_conv_with_state_b1_c1_degenerate</code> , <code>test_causal_conv_with_state_basic</code> , <code>test_causal_conv_with_state_decode_step</code> , <code>test_causal_conv_with_state_fp16</code> , <code>test_causal_conv_with_state_kernel_size_one</code> , <code>test_causal_conv_with_state_short_input_no_past_state</code> , <code>test_causal_conv_with_state_silu</code> , <code>test_causal_conv_with_state_silu_fp16</code> , <code>test_causal_conv_with_state_silu_with_past_state</code> , <code>test_causal_conv_with_state_swish_alias</code> , <code>test_causal_conv_with_state_with_bias</code> , <code>test_causal_conv_with_state_with_bias_and_past_state</code> , <code>test_causal_conv_with_state_with_past_state</code>
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.28. Ceil

Ceil takes one input data (Tensor<T>) and produces one output data (Tensor<T>) where the ceil is, $y = \text{ceil}(x)$, is applied to the tensor elementwise. If x is integral, +0, -0, NaN, or infinite, x itself is returned.

Domain	<code>ai.onnx</code>
Since version	13
Earlier versions	1, 6
Inputs	X (non-differentiable) : T — Input tensor
Outputs	Y (non-differentiable) : T — Output tensor
Attributes	None.
Type constraints	T : <code>tensor(float16)</code> , <code>tensor(float)</code> , <code>tensor(double)</code> , <code>tensor(bfloat16)</code> — Constrain input and output types to float tensors.
Test vectors	<code>test_ceil_example</code> , <code>test_ceil</code>
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.29. Celu

Continuously Differentiable Exponential Linear Units: Perform the linear unit element-wise on the input tensor X using formula:

$$\max(0, x) + \min(0, \alpha * (\exp(x/\alpha) - 1))$$

Domain	ai.onnx
Since version	28
Earlier versions	12
Inputs	X (differentiable) : T—Input tensor
Outputs	Y (differentiable) : T—Output tensor
Attributes	alpha : float (default is 1.0) — The Alpha value in Celu formula which control the shape of the unit. The default value is 1.0.
Type constraints	T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain input and output types to float tensors.
Test vectors	test_celu, test_celu_bfloat16, test_celu_float16
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.30. CenterCropPad

Center crop or pad an input to given dimensions.

The crop/pad dimensions can be specified for a subset of the axes; unspecified dimensions will remain unchanged.

If the input dimensions are larger than the target crop dimensions, a centered cropping window will be extracted from the input. The starting value for the cropping window is rounded down, which means that if the difference between the input shape and the crop shape is odd, the cropping window will be shifted half a pixel to the left of the input center.

If the input dimensions are smaller than the target crop dimensions, the input will be padded equally on both sides to center it in the output. In cases where the total number of padding pixels is odd, an additional pixel will be added to the right side.

The padding value used is zero.

Domain	ai.onnx
Since version	18
Inputs	input_data (differentiable) : T—Input to extract the centered crop from. shape (non-differentiable) : Tind — 1-D tensor representing the cropping window dimensions.

Outputs	output_data (differentiable) : T — Output data.
Attributes	axes : list of ints — If provided, it specifies a subset of axes that 'shape' refer to. If not provided, all axes are assumed [0, 1, ..., r-1], where r = rank(data). Negative value means counting dimensions from the back. Accepted range is [-r, r-1], where r = rank(data). Behavior is undefined if an axis is repeated.
Type constraints	T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128) — Constrain input and output types to all tensor types. Tind : tensor(int32), tensor(int64) — Constrain indices to integer types
Test vectors	test_center_crop_pad_crop, test_center_crop_pad_crop_and_pad, test_center_crop_pad_crop_axes_chw, test_center_crop_pad_crop_axes_hwc, test_center_crop_pad_crop_negative_axes_hwc, test_center_crop_pad_pad
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.31. Clip

Clip operator limits the given input within an interval. The interval is specified by the inputs 'min' and 'max'. They default to `numeric_limits::lowest()` and `numeric_limits::max()`, respectively. When 'min' is greater than 'max', the clip operator sets all the 'input' values to the value of 'max'. Thus, this is equivalent to `'Min(max, Max(input, min))'`.

Domain	ai.onnx
Since version	13
Earlier versions	1, 6, 11, 12
Inputs (1 — 3)	input (differentiable) : T — Input tensor whose elements to be clipped min (optional, non-differentiable) : T — Minimum value, under which element is replaced by min. It must be a scalar(tensor of empty shape). max (optional, non-differentiable) : T — Maximum value, above which element is replaced by max. It must be a scalar(tensor of empty shape).
Outputs	output (differentiable) : T — Output tensor with clipped input elements
Attributes	None.
Type constraints	T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(float16), tensor(float),

	<code>tensor(double)</code> , <code>tensor(bfloat16)</code> — Constrain input and output types to all numeric tensors.
Test vectors	<code>test_clip_example</code> , <code>test_clip</code> , <code>test_clip_inbounds</code> , <code>test_clip_outbounds</code> , <code>test_clip_splitbounds</code> , <code>test_clip_min_greater_than_max</code> , <code>test_clip_default_min</code> , <code>test_clip_default_max</code> , <code>test_clip_default_inbounds</code> , <code>test_clip_default_int8_min</code> , <code>test_clip_default_int8_max</code> , <code>test_clip_default_int8_inbounds</code>
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.32. Col2Im

The operator rearranges column blocks back into a multidimensional image

Col2Im behaves similarly to PyTorch's fold <https://pytorch.org/docs/stable/generated/torch.nn.Fold.html>, but it only supports *batched* multi-dimensional image tensors. Another implementation in Python with N-dimension support can be found at <https://github.com/f-dangel/unfoldNd/>.

NOTE: Although specifying `image_shape` looks redundant because it could be calculated from convolution formulas, it is required as input for more advanced scenarios as explained at PyTorch's implementation (<https://github.com/pytorch/pytorch/blob/master/aten/src/ATen/native/Col2Im.cpp#L10>)

Domain	<code>ai.onnx</code>
Since version	18
Inputs	<code>input</code> (differentiable) : T — Input data tensor to be rearranged from column blocks back into an image. This is a 3-dimensional tensor containing $[N, C * \text{n-ary-product}(\text{block_shape}), L]$, where N is batch dimension, C is image channel dimension and L is number of blocks. The blocks are enumerated in increasing lexicographic-order of their indices. For example, with an image-size 1020 and block-size 918, there would be 2*3 blocks, enumerated in the order <code>block(0, 0)</code>, <code>block(0, 1)</code>, <code>block(0, 2)</code>, <code>block(1, 0)</code>, <code>block(1, 1)</code>, <code>block(1, 2)</code>. <code>image_shape</code> (non-differentiable) : <code>tensor(int64)</code> — The shape of the spatial dimensions of the image after rearranging the column blocks. This is a 1-dimensional tensor with size of at least 2, containing the value $[H_img, W_img]$ for a 2-D image or $[dim_i1, dim_i2, ..., dim_iN]$ for a N-D image. <code>block_shape</code> (non-differentiable) : <code>tensor(int64)</code> — The shape of the block to apply on the input. This is a 1-dimensional tensor of size of at least 2, containing the value $[H_block, W_block]$ for a 2-D image or $[dim_b1, dim_b2, ..., dim_bN]$ for a N-D block. This is the block-shape before dilation is applied to it.
Outputs	<code>output</code> (differentiable) : T — Output tensor produced by rearranging blocks into an image.
Attributes	<code>dilations</code> : list of ints — 1-dimensional tensor with dilation value along each spatial axis of the image. If not present, the dilation defaults to 1 along each spatial axis of the image. <code>pads</code> : list of ints — 1-dimensional tensor with padding value for the beginning and ending along each spatial axis, it can take any value greater than or equal to 0. The value represent the number of pixels

added to the beginning and end part of the corresponding axis. pads format should be as follow [x1_begin, x2_begin...x1_end, x2_end,...], where xi_begin is the number of pixels added at the beginning of axis i and xi_end is the number of pixels added at the end of axis i. If not present, the padding defaults to 0 along start and end of each spatial axis.
 strides : list of ints — 1-dimensional tensor with stride value along each spatial axis. If not present, the stride defaults to 1 along each spatial axis.

Type constraints	T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128) — Constrain input and output types to all numeric tensor types.
Test vectors	test_col2im, test_col2im_5d, test_col2im_dilations, test_col2im_pads, test_col2im_strides
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.33. Compress

Selects slices from an input tensor along a given axis where condition evaluates to True for each axis index. In case axis is not provided, input is flattened before elements are selected. Compress behaves like `numpy.compress`: <https://docs.scipy.org/doc/numpy/reference/generated/numpy.compress.html>

Domain	ai.onnx
Since version	28
Earlier versions	9, 11
Inputs	input (differentiable) : T — Tensor of rank $r \geq 1$. condition (non-differentiable) : T1 — Rank 1 tensor of booleans to indicate which slices or data elements to be selected. Its length can be less than the input length along the axis or the flattened input size if axis is not specified. In such cases data slices or elements exceeding the condition length are discarded.
Outputs	output (differentiable) : T — Tensor of rank r if axis is specified. Otherwise output is a Tensor of rank 1.
Attributes	axis : int — (Optional) Axis along which to take slices. If not specified, input is flattened before elements being selected. Negative value means counting dimensions from the back. Accepted range is $[-r, r-1]$ where $r = \text{rank}(\text{input})$.
Type constraints	T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128) — Constrain input and output types to all tensor types.

T1 : tensor(bool) — Constrain to boolean tensors.

Test vectors test_compress_0, test_compress_1, test_compress_bfloat16, test_compress_default_axis, test_compress_negative_axis

Not supplied by the upstream source Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.34. Concat

Concatenate a list of tensors into a single tensor. All input tensors must have the same shape, except for the dimension size of the axis to concatenate on.

Domain ai.onnx

Since version 13

Earlier versions 1, 4, 11

Inputs (1 — unbounded) inputs (variadic, differentiable) : T — List of tensors for concatenation

Outputs concat_result (differentiable) : T — Concatenated tensor

Attributes axis : int (required) — Which axis to concat on. A negative value means counting dimensions from the back. Accepted range is [-r, r-1] where r = rank(inputs)..

Type constraints T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128) — Constrain output types to any tensor type.

Test vectors test_concat_

Not supplied by the upstream source Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.35. ConcatFromSequence

Concatenate a sequence of tensors into a single tensor. All input tensors must have the same shape, except for the dimension size of the axis to concatenate on. By default 'new_axis' is 0, the behavior is similar to numpy.concatenate. When 'new_axis' is 1, the behavior is similar to numpy.stack.

Domain ai.onnx

Since version 11

Inputs input_sequence : S — Sequence of tensors for concatenation

Outputs concat_result : T — Concatenated tensor

Attributes	<code>axis</code> : <code>int</code> (required) — Which axis to concat on. Accepted range in $[-r, r - 1]$, where r is the rank of input tensors. When <code>new_axis</code> is 1, accepted range is $[-r - 1, r]$. <code>new_axis</code> : <code>int</code> (default is 0) — Insert and concatenate on a new axis or not, default 0 means do not insert new axis.
Type constraints	<code>S</code> : <code>seq(tensor(uint8))</code>, <code>seq(tensor(uint16))</code>, <code>seq(tensor(uint32))</code>, <code>seq(tensor(uint64))</code>, <code>seq(tensor(int8))</code>, <code>seq(tensor(int16))</code>, <code>seq(tensor(int32))</code>, <code>seq(tensor(int64))</code>, <code>seq(tensor(float16))</code>, <code>seq(tensor(float))</code>, <code>seq(tensor(double))</code>, <code>seq(tensor(string))</code>, <code>seq(tensor(bool))</code>, <code>seq(tensor(complex64))</code>, <code>seq(tensor(complex128))</code> — Constrain input types to any tensor type. <code>T</code> : <code>tensor(uint8)</code>, <code>tensor(uint16)</code>, <code>tensor(uint32)</code>, <code>tensor(uint64)</code>, <code>tensor(int8)</code>, <code>tensor(int16)</code>, <code>tensor(int32)</code>, <code>tensor(int64)</code>, <code>tensor(float16)</code>, <code>tensor(float)</code>, <code>tensor(double)</code>, <code>tensor(string)</code>, <code>tensor(bool)</code>, <code>tensor(complex64)</code>, <code>tensor(complex128)</code> — Constrain output types to any tensor type.
Test vectors	None published for this operator.
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.36. Constant

This operator produces a constant tensor. Exactly one of the provided attributes, either `value`, `sparse_value`, or `value_*` must be specified.

Domain	<code>ai.onnx</code>
Since version	25
Earlier versions	1, 9, 11, 12, 13, 19, 21, 23, 24
Inputs	None.
Outputs	<code>output</code> : <code>T</code> — Output tensor containing the same value of the provided tensor.
Attributes	<code>sparse_value</code> : <code>sparse_tensor</code> — The value for the elements of the output tensor in sparse format. <code>value</code> : <code>tensor</code> — The value for the elements of the output tensor. <code>value_float</code> : <code>float</code> — The value for the sole element for the scalar, float32, output tensor. <code>value_floats</code> : <code>list of floats</code> — The values for the elements for the 1D, float32, output tensor. <code>value_int</code> : <code>int</code> — The value for the sole element for the scalar, int64, output tensor. <code>value_ints</code> : <code>list of ints</code> — The values for the elements for the 1D, int64, output tensor. <code>value_string</code> : <code>string</code> — The value for the sole element for the scalar, UTF-8 string, output tensor. <code>value_strings</code> : <code>list of strings</code> — The values for the elements for the 1D, UTF-8 string, output tensor.

Type constraints	T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128), tensor(float8e4m3fn), tensor(float8e4m3fnuz), tensor(float8e5m2), tensor(float8e5m2fnuz), tensor(uint4), tensor(int4), tensor(float4e2m1), tensor(float8e8m0), tensor(uint2), tensor(int2) — Constrain input and output types to all tensor types.
Test vectors	test_constant
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.37. ConstantOfShape

Generate a tensor with given value and shape.

Domain	ai.onnx
Since version	25
Earlier versions	9, 20, 21, 23, 24
Inputs	input : T1 — 1D tensor. The shape of the expected output tensor. If empty tensor is given, the output would be a scalar. All values must be >= 0.
Outputs	output : T2 — Output tensor of shape specified by 'input'. If attribute 'value' is specified, the value and datatype of the output tensor is taken from 'value'. If attribute 'value' is not specified, the value in the output defaults to 0, and the datatype defaults to float32.
Attributes	value : tensor — (Optional) The value of the output elements. Should be a one-element tensor. If not specified, it defaults to a tensor of value 0 and datatype float32
Type constraints	T1 : tensor(int64) — Constrain input types. T2 : tensor(float16), tensor(float), tensor(double), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(uint4), tensor(int4), tensor(bool), tensor(bfloat16), tensor(float8e4m3fn), tensor(float8e4m3fnuz), tensor(float8e5m2), tensor(float8e5m2fnuz), tensor(float4e2m1), tensor(float8e8m0), tensor(uint2), tensor(int2) — Constrain output types to be numerics or boolean.
Test vectors	test_constantofshape_float_ones, test_constantofshape_int_shape_zero, test_constantofshape_int_zeros
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.38. Conv

The convolution operator consumes an input tensor and a filter, and computes the output.

Domain	ai.onnx
Since version	22
Earlier versions	1, 11
Inputs (2 — 3)	X (differentiable) : T — Input data tensor from previous layer; has size $(N \times C \times H \times W)$, where N is the batch size, C is the number of channels, and H and W are the height and width. Note that this is for the 2D image. Otherwise the size is $(N \times C \times D1 \times D2 \dots \times Dn)$. Optionally, if dimension denotation is in effect, the operation expects input data tensor to arrive with the dimension denotation of <code>[DATA_BATCH, DATA_CHANNEL, DATA_FEATURE, DATA_FEATURE ...]</code>. W (differentiable) : T — The weight tensor that will be used in the convolutions; has size $(M \times C/\text{group} \times kH \times kW)$, where C is the number of channels, and kH and kW are the height and width of the kernel, and M is the number of feature maps. For more than 2 dimensions, the kernel shape will be $(M \times C/\text{group} \times k1 \times k2 \dots \times kn)$, where $(k1 \times k2 \dots \times kn)$ is the dimension of the kernel. Optionally, if dimension denotation is in effect, the operation expects the weight tensor to arrive with the dimension denotation of <code>[FILTER_OUT_CHANNEL, FILTER_IN_CHANNEL, FILTER_SPATIAL, FILTER_SPATIAL ...]</code>. Assuming zero based indices for the shape array, $X.\text{shape}[1] == (W.\text{shape}[1] * \text{group}) == C$ and $W.\text{shape}[0] \bmod G == 0$. Or in other words <code>FILTER_IN_CHANNEL</code> multiplied by the number of groups should be equal to <code>DATA_CHANNEL</code> and the number of feature maps M should be a multiple of the number of groups G. B (optional, differentiable) : T — Optional 1D bias to be added to the convolution, has size of M.
Outputs	Y (differentiable) : T — Output data tensor that contains the result of the convolution. The output dimensions are functions of the kernel size, stride size, and pad lengths.
Attributes	auto_pad : string (default is NOTSET) — auto_pad must be either NOTSET, SAME_UPPER, SAME_LOWER or VALID. Where default value is NOTSET, which means explicit padding is used. SAME_UPPER or SAME_LOWER mean pad the input so that $\text{output_shape}[i] = \text{ceil}(\text{input_shape}[i] / \text{strides}[i])$ for each axis i. The padding is split between the two sides equally or almost equally (depending on whether it is even or odd). In case the padding is an odd number, the extra padding is added at the end for SAME_UPPER and at the beginning for SAME_LOWER. dilations : list of ints — dilation value along each spatial axis of the filter. If not present, the dilation defaults is 1 along each spatial axis. group : int (default is 1) — number of groups input channels and output channels are divided into. kernel_shape : list of ints — The shape of the convolution kernel. If not present, should be inferred from input W. pads : list of ints — Padding for the beginning and ending along each spatial axis, it can take any value greater than or equal to 0. The value represent the number of pixels added to the beginning and end part of the corresponding axis. pads format should be as follow <code>[x1_begin, x2_begin...x1_end, x2_end,...]</code>, where x_i_begin the number of pixels added at the beginning of axis i and x_i_end, the number of pixels added at the end of axis i. This attribute cannot be used simultaneously with auto_pad

	attribute. If not present, the padding defaults to 0 along start and end of each spatial axis. strides : list of ints — Stride along each spatial axis. If not present, the stride defaults is 1 along each spatial axis.
Type constraints	T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain input and output types to float tensors.
Test vectors	test_basic_conv_with_padding, test_basic_conv_without_padding, test_conv_with_autopad_same, test_conv_with_strides_padding, test_conv_with_strides_no_padding, test_conv_with_strides_and_asymmetric_padding
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.39. ConvInteger

The integer convolution operator consumes an input tensor, its zero-point, a filter, and its zero-point, and computes the output. The production MUST never overflow. The accumulation may overflow if and only if in 32 bits.

Domain	ai.onnx
Since version	10
Inputs (2 — 4)	x : T1 — Input data tensor from previous layer; has size (N x C x H x W), where N is the batch size, C is the number of channels, and H and W are the height and width. Note that this is for the 2D image. Otherwise the size is (N x C x D1 x D2 ... x Dn). Optionally, if dimension denotation is in effect, the operation expects input data tensor to arrive with the dimension denotation of [DATA_BATCH, DATA_CHANNEL, DATA_FEATURE, DATA_FEATURE ...]. w : T2 — The weight tensor that will be used in the convolutions; has size (M x C/group x kH x kW), where C is the number of channels, and kH and kW are the height and width of the kernel, and M is the number of feature maps. For more than 2 dimensions, the kernel shape will be (M x C/group x k1 x k2 x ... x kn), where (k1 x k2 x ... x kn) is the dimension of the kernel. Optionally, if dimension denotation is in effect, the operation expects the weight tensor to arrive with the dimension denotation of [FILTER_OUT_CHANNEL, FILTER_IN_CHANNEL, FILTER_SPATIAL, FILTER_SPATIAL ...]. X.shape[1] == (W.shape[1] * group) == C (assuming zero based indices for the shape array). Or in other words FILTER_IN_CHANNEL should be equal to DATA_CHANNEL. x_zero_point (optional) : T1 — Zero point tensor for input 'x'. It's optional and default value is 0. It's a scalar, which means a per-tensor/layer quantization. w_zero_point (optional) : T2 — Zero point tensor for input 'w'. It's optional and default value is 0. It could be a scalar or a 1-D tensor, which means a per-tensor/layer or per output channel quantization. If it's a 1-D tensor, its number of elements should be equal to the number of output channels (M)

Outputs	<code>y</code> : T3 — Output data tensor that contains the result of the convolution. The output dimensions are functions of the kernel size, stride size, and pad lengths.
Attributes	<code>auto_pad</code> : string (default is NOTSET) — <code>auto_pad</code> must be either NOTSET, SAME_UPPER, SAME_LOWER or VALID. Where default value is NOTSET, which means explicit padding is used. SAME_UPPER or SAME_LOWER mean pad the input so that $\text{output_shape}[i] = \text{ceil}(\text{input_shape}[i] / \text{strides}[i])$ for each axis i. The padding is split between the two sides equally or almost equally (depending on whether it is even or odd). In case the padding is an odd number, the extra padding is added at the end for SAME_UPPER and at the beginning for SAME_LOWER. <code>dilations</code> : list of ints — dilation value along each spatial axis of the filter. If not present, the dilation defaults to 1 along each axis. <code>group</code> : int (default is 1) — number of groups input channels and output channels are divided into. default is 1. <code>kernel_shape</code> : list of ints — The shape of the convolution kernel. If not present, should be inferred from input 'w'. <code>pads</code> : list of ints — Padding for the beginning and ending along each spatial axis, it can take any value greater than or equal to 0. The value represent the number of pixels added to the beginning and end part of the corresponding axis. <code>pads</code> format should be as follow <code>[x1_begin, x2_begin, ..., x1_end, x2_end, ...]</code>, where x_i_begin the number of pixels added at the beginning of axis i and x_i_end, the number of pixels added at the end of axis i. This attribute cannot be used simultaneously with <code>auto_pad</code> attribute. If not present, the padding defaults to 0 along start and end of each spatial axis. <code>strides</code> : list of ints — Stride along each spatial axis. If not present, the stride defaults to 1 along each axis.
Type constraints	T1 : tensor(int8), tensor(uint8) — Constrain input x and its zero point data type to 8-bit integer tensor. T2 : tensor(int8), tensor(uint8) — Constrain input w and its zero point data type to 8-bit integer tensor. T3 : tensor(int32) — Constrain output y data type to 32-bit integer tensor.
Test vectors	<code>test_convinteger_with_padding</code> , <code>test_convinteger_without_padding</code>
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.40. ConvTranspose

The convolution transpose operator consumes an input tensor and a filter, and computes the output.

If the `pads` parameter is provided the shape of the output is calculated via the following equation:

$$\text{output_shape}[i] = \text{stride}[i] * (\text{input_size}[i] - 1) + \text{output_padding}[i] + ((\text{kernel_shape}[i] - 1) * \text{dilations}[i] + 1) - \text{pads}[\text{start}_i] - \text{pads}[\text{end}_i]$$

`output_shape` can also be explicitly specified in which case `pads` values are auto generated using these equations:

$\text{total_padding}[i] = \text{stride}[i] * (\text{input_size}[i] - 1) + \text{output_padding}[i] + ((\text{kernel_shape}[i] - 1) * \text{dilations}[i] + 1) - \text{output_shape}[i]$ If (auto_pads == SAME_UPPER): pads[start_i] = total_padding[i]/2; pads[end_i] = total_padding[i] — (total_padding[i]/2) Else: pads[start_i] = total_padding[i] — (total_padding[i]/2); pads[end_i] = (total_padding[i]/2).

Domain	ai.onnx
Since version	22
Earlier versions	1, 11
Inputs (2 — 3)	X (differentiable) : T — Input data tensor from previous layer; has size (N x C x H x W), where N is the batch size, C is the number of channels, and H and W are the height and width. Note that this is for the 2D image. Otherwise the size is (N x C x D1 x D2 ... x Dn) W (differentiable) : T — The weight tensor that will be used in the convolutions; has size (C x M/group x kH x kW), where C is the number of channels, and kH and kW are the height and width of the kernel, and M is the number of feature maps. For more than 2 dimensions, the weight shape will be (C x M/group x k1 x k2 x ... x kn), where (k1 x k2 x ... x kn) is the dimension of the kernel. The number of channels in the output should be equal to W.shape[1] * group (assuming zero based indices of the shape array) B (optional, differentiable) : T — Optional 1D bias to be added to the convolution, has size of M.
Outputs	Y (differentiable) : T — Output data tensor that contains the result of the convolution. The output dimensions are functions of the kernel size, stride size, pad lengths and group count. The number of channels in the output should be equal to W.shape[1] * group (assuming zero based indices of the shape array)
Attributes	auto_pad : string (default is NOTSET) — auto_pad must be either NOTSET, SAME_UPPER, SAME_LOWER or VALID. Where default value is NOTSET, which means explicit padding is used. SAME_UPPER or SAME_LOWER mean pad the input so that output_shape[i] = input_shape[i] * strides[i] for each axis i. The padding is split between the two sides equally or almost equally (depending on whether it is even or odd). In case the padding is an odd number, the extra padding is added at the end for SAME_UPPER and at the beginning for SAME_LOWER. dilations : list of ints — dilation value along each spatial axis of the filter. If not present, the dilation defaults to 1 along each spatial axis. group : int (default is 1) — number of groups input channels and output channels are divided into. kernel_shape : list of ints — The shape of the convolution kernel. If not present, should be inferred from input W. output_padding : list of ints — Additional elements added to the side with higher coordinate indices in the output. Each padding value in "output_padding" must be less than the corresponding stride/dilation dimension. By default, this attribute is a zero vector. Note that this attribute doesn't directly affect the computed output values. It only controls the selection of the computed values, so changing this attribute only adds or removes output elements. If "output_shape" is explicitly provided, "output_padding" does not contribute additional size to "output_shape" but participates in the computation of the needed padding amount. This is also called adjs or adjustment in some frameworks. output_shape : list of ints — The shape of the output can be explicitly set which will cause pads values to be auto generated. If output_shape is specified pads values are ignored. See doc for details for equations to generate pads. Note that the output_shape attribute value

should not include dimensions for batch size and channels, which are automatically inferred.

pads : list of ints — Padding for the beginning and ending along each spatial axis, it can take any value greater than or equal to 0. The value represent the number of pixels added to the beginning and end part of the corresponding axis. pads format should be as follow [x1_begin, x2_begin...x1_end, x2_end,...], where xi_begin the number of pixels added at the beginning of axis i and xi_end, the number of pixels added at the end of axis i. This attribute cannot be used simultaneously with auto_pad attribute. If not present, the padding defaults to 0 along start and end of each spatial axis.

strides : list of ints — Stride along each spatial axis. If not present, the stride defaults to 1 along each spatial axis.

Type constraints	T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain input and output types to float tensors.
Test vectors	test_convtranspose, test_convtranspose_1d, test_convtranspose_3d, test_convtranspose_output_shape, test_convtranspose_pad, test_convtranspose_kernel_shape, test_convtranspose_autopad_same, test_convtranspose_dilations, test_convtranspose_group_2, test_convtranspose_group_2_image_3, test_convtranspose_pads
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.41. Cos

Calculates the cosine of the given input tensor, element-wise.

Domain	ai.onnx
Since version	22
Earlier versions	7
Inputs	input (differentiable) : T — Input tensor
Outputs	output (differentiable) : T — The cosine of the input tensor computed element-wise
Attributes	None.
Type constraints	T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain input and output types to float tensors.
Test vectors	test_cos_example, test_cos
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.42. Cosh

Calculates the hyperbolic cosine of the given input tensor element-wise.

Domain	ai.onnx
Since version	22
Earlier versions	9
Inputs	input (differentiable) : T—Input tensor
Outputs	output (differentiable) : T—The hyperbolic cosine values of the input tensor computed element-wise
Attributes	None.
Type constraints	T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain input and output types to float tensors.
Test vectors	test_cosh_example, test_cosh
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.43. CumProd

Performs cumulative product of the input elements along the given axis. By default, it will do the product inclusively meaning the first element is copied as is. Through an exclusive attribute, this behavior can change to exclude the first element. It can also perform product in the opposite direction of the axis. For that, set reverse attribute to 1.

Example:

```
input_x = [1, 2, 3]
axis=0
output = [1, 2, 6]
exclusive=1
output = [1, 1, 2]
exclusive=0
reverse=1
output = [6, 6, 3]
exclusive=1
reverse=1
output = [6, 3, 1]
```

Domain	ai.onnx
Since version	26
Inputs	x (differentiable) : T—An input tensor that is to be processed. axis (non-differentiable) : T2—A 0-D tensor. Must be in the range [-rank(x), rank(x)-1]. Negative value means counting dimensions from the back.
Outputs	y (differentiable) : T—Output tensor of the same type as 'x' with cumulative products of the x's elements
Attributes	exclusive : int (default is 0) — If set to 1 will return exclusive product in which the top element is not included. In other terms, if set to 1, the j-th output element would be the product of the first (j-1) elements. Otherwise, it would be the product of the first j elements.

	<code>reverse : int (default is 0)</code> — If set to 1 will perform the products in reverse direction.
Type constraints	<code>T : tensor(uint32), tensor(uint64), tensor(int32), tensor(int64), tensor(float16), tensor(float), tensor(double), tensor(bfloat16)</code> — Constrain input and output types to numeric tensors. <code>T2 : tensor(int32), tensor(int64)</code> — axis tensor can be int32 or int64 only
Test vectors	<code>test_cumprod_1d, test_cumprod_1d_exclusive, test_cumprod_1d_int32_exclusive, test_cumprod_1d_reverse, test_cumprod_1d_reverse_exclusive, test_cumprod_2d_axis_0, test_cumprod_2d_axis_1, test_cumprod_2d_int32, test_cumprod_2d_negative_axis</code>
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.44. CumSum

Performs cumulative sum of the input elements along the given axis. By default, it will do the sum inclusively meaning the first element is copied as is. Through an exclusive attribute, this behavior can change to exclude the first element. It can also perform summation in the opposite direction of the axis. For that, set `reverse` attribute to 1.

Example:

```
input_x = [1, 2, 3]
axis=0
output = [1, 3, 6]
exclusive=1
output = [0, 1, 3]
exclusive=0
reverse=1
output = [6, 5, 3]
exclusive=1
reverse=1
output = [5, 3, 0]
```

Domain	ai.onnx
Since version	14
Earlier versions	11
Inputs	<code>x (differentiable) : T</code> — An input tensor that is to be processed. <code>axis (non-differentiable) : T2</code> — A 0-D tensor. Must be in the range <code>[-rank(x), rank(x)-1]</code> . Negative value means counting dimensions from the back.
Outputs	<code>y (differentiable) : T</code> — Output tensor of the same type as 'x' with cumulative sums of the x's elements
Attributes	<code>exclusive : int (default is 0)</code> — If set to 1 will return exclusive sum in which the top element is not included. In other terms, if set to 1, the j-th output element would be the sum of the first (j-1) elements. Otherwise, it would be the sum of the first j elements.

reverse : int (default is 0) — If set to 1 will perform the sums in reverse direction.

Type constraints T : tensor(uint32), tensor(uint64), tensor(int32), tensor(int64), tensor(float16), tensor(float), tensor(double), tensor(bfloat16) — Constrain input and output types to numeric tensors.
T2 : tensor(int32), tensor(int64) — axis tensor can be int32 or int64 only

Test vectors test_cumsum_1d, test_cumsum_1d_exclusive, test_cumsum_1d_int32_exclusive, test_cumsum_1d_reverse, test_cumsum_1d_reverse_exclusive, test_cumsum_2d_axis_0, test_cumsum_2d_axis_1, test_cumsum_2d_int32, test_cumsum_2d_negative_axis

Not supplied by the upstream source Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.45. DFT

Computes the discrete Fourier Transform (DFT) of the input.

Assuming the input has shape [M, N], where N is the dimension over which the DFT is computed and M denotes the conceptual “all other dimensions,” the DFT $y[m, k]$ of shape [M, N] is defined as

$$y[m, k] = \sum_{n=0}^{N-1} e^{-2\pi j \frac{kn}{N}} x[m, n], \quad (1)$$

and the inverse transform is defined as

$$x[m, n] = \frac{1}{N} \sum_{k=0}^{N-1} e^{2\pi j \frac{kn}{N}} y[m, k], \quad (2)$$

where j is the imaginary unit.

The actual shape of the output is specified in the “output” section.

Reference: <https://docs.scipy.org/doc/scipy/tutorial/fft.html>

Domain ai.onnx

Since version 20

Earlier versions 17

Inputs (1 — 3) input (non-differentiable) : T1 — For real input, the following shape is expected: [signal_dim0][signal_dim1][signal_dim2]...[signal_dimN][1]. For complex input, the following shape is expected: [signal_dim0][signal_dim1][signal_dim2]...[signal_dimN][2]. The final dimension represents the real and imaginary parts of the value in that order.
dft_length (optional, non-differentiable) : T2 — The length of the signal as a scalar. If greater than the axis dimension, the signal will be zero-padded up to dft_length. If less than the axis dimension, only the first dft_length values will be used as the signal. If not provided, the default dft_length = signal_dim_axis, except for the IFFT case

	(onesided=1, inverse=1), in which case the default <code>dft_length</code> is $2 * (\text{signal_dim_axis} - 1)$. <code>axis</code> (optional, non-differentiable) : <code>tensor(int64)</code> — The axis as a scalar on which to perform the DFT. Default is -2 (last signal axis). Negative value means counting dimensions from the back. Accepted range is $[-r, -2] \cup [0, r-2]$ where $r = \text{rank}(\text{input})$. The last dimension is for representing complex numbers and thus is an invalid axis.
Outputs	<code>output</code> : <code>T1</code> — The Fourier Transform of the input vector. For standard DFT (<code>onesided=0</code>), the output shape is: <code>[signal_dim0][signal_dim1][signal_dim2]...[signal_dimN][2]</code> (complex), with <code>signal_dim_axis = dft_length</code>. For RFFT (<code>onesided=1, inverse=0</code>), the output shape is: <code>[signal_dim0][signal_dim1][signal_dim2]...[signal_dimN][2]</code> (one-sided complex), with <code>signal_dim_axis = floor(dft_length/2) + 1</code>. For IRFFT (<code>onesided=1, inverse=1</code>), the output shape is: <code>[signal_dim0][signal_dim1][signal_dim2]...[signal_dimN][1]</code> (real), where <code>signal_dim_axis = dft_length</code>.
Attributes	<code>inverse</code> : <code>int</code> (default is 0) — Whether to perform the inverse discrete Fourier Transform. Default is 0, which corresponds to false. <code>onesided</code> : <code>int</code> (default is 0) — If <code>onesided</code> is 1, only values for <code>k</code> in $[0, 1, 2, \dots, \text{floor}(n_fft/2) + 1]$ are used or returned because the real-to-complex Fourier transform satisfies the conjugate symmetry, i. e., $X[m, k] = X[m, n_fft - k]^*$, where <code>m</code> denotes “all other dimensions” DFT was not applied on. When <code>onesided=1</code> and <code>inverse=0</code> (forward DFT), only real input is supported and a one-sided complex spectrum is returned (RFFT). When <code>onesided=1</code> and <code>inverse=1</code> (inverse DFT), only complex input is supported and a full real signal is returned (IRFFT). Value can be 0 or 1. Default is 0.
Type constraints	<code>T1</code> : <code>tensor(bfloat16)</code>, <code>tensor(float16)</code>, <code>tensor(float)</code>, <code>tensor(double)</code> — Constrain input and output types to float tensors. <code>T2</code> : <code>tensor(int32)</code>, <code>tensor(int64)</code> — Constrain scalar length types to integers.
Test vectors	<code>test_dft</code>, <code>test_dft_axis</code>, <code>test_dft_inverse</code>, <code>test_dft_rfft</code>, <code>test_dft_irfft</code>, <code>test_dft_opset19</code>, <code>test_dft_axis_opset19</code>, <code>test_dft_inverse_opset19</code>, <code>test_dft_rfft_opset19</code>, <code>test_dft_irfft_opset19</code>
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.46. DeformConv

Performs deformable convolution as described in <https://arxiv.org/abs/1703.06211> and <https://arxiv.org/abs/1811.11168>. This operator specification supports the general N-D case. Note that most common use cases have 2D or 3D data.

Domain	<code>ai.onnx</code>
Since version	22
Earlier versions	19
Inputs (3 — 5)	<code>X</code> : <code>T</code> — Input data tensor. For 2D image data, it has shape (N, C, H, W) where <code>N</code> is the batch size, <code>C</code> is the number of input channels, and <code>H</code> and <code>W</code> are the height and width. In general, the shape is $(N, C, D_1, D_2, \dots, D_n)$ for

	n-dimensional data, where D_1 to D_n are the spatial dimension sizes. Most common use cases have $n = 2$ or 3. W : T — Weight tensor that will be used in the convolutions. It has shape $(oC, C/group, kH, kW)$, where oC is the number of output channels and kH and kW are the kernel height and width. For more than 2 dimensions, it has shape $(oC, C/group, k_1, k_2, \dots, k_n)$. $offset$: T — Offset tensor denoting the offset for the sampling locations in the convolution kernel. It has shape $(N, offset_group * kH * kW * 2, oH, oW)$ for 2D data or $(N, offset_group * k_1 * k_2 * \dots * k_n * n, o_1, o_2, \dots, o_n)$ for nD data. Use linear interpolation for fractional offset values. Sampling locations outside of the padded input tensor gives zero. B (optional) : T — Optional 1D bias of length oC to be added to the convolution. Default is a tensor of zeros. $mask$ (optional) : T — The mask tensor to be applied to each position in the convolution kernel. It has shape $(N, offset_group * kH * kW, oH, oW)$ for 2D data or $(N, offset_group * k_1 * k_2 * \dots * k_n * n, o_1, o_2, \dots, o_n)$ for nD data. Default is a tensor of ones.
Outputs	Y : T — Output data tensor that contains the result of convolution. It has shape (N, oC, oH, oW) for 2D data or $(N, oC, o_1, o_2, \dots, o_n)$ for nD data
Attributes	$dilations$: list of ints — Dilation value along each spatial axis of the kernel. Default is 1 along each axis. $group$: int (default is 1) — Number of groups the input and output channels, C and oC, are divided into. C and oC must both be divisible by $group$. Default is 1. $kernel_shape$: list of ints — Shape of the convolution kernel. If not present, it is inferred from the shape of input W. $offset_group$: int (default is 1) — Number of groups of offset. C must be divisible by $offset_group$. Default is 1. $pads$: list of ints — Padding for the beginning and end along each spatial axis. The values represent the number of pixels added to the beginning and end of the corresponding axis and can take any nonnegative value. The format should be as follows: $[x1_begin, x2_begin, \dots, x1_end, x2_end, \dots]$, where xi_begin is the number of pixels added at the beginning of axis i and xi_end is the number of pixels added at the end of axis i. Default is 0 along each axis. $strides$: list of ints — Stride along each spatial axis. Default is 1 along each axis.
Type constraints	T : $tensor(bfloat16), tensor(float16), tensor(float), tensor(double)$ — Constrain input and output types to float tensors.
Test vectors	<code>test_basic_deform_conv_with_padding</code>, <code>test_basic_deform_conv_without_padding</code>, <code>test_deform_conv_with_mask_bias</code>, <code>test_deform_conv_with_multiple_offset_groups</code>
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.47. DepthToSpace

DepthToSpace rearranges (permutes) data from depth into blocks of spatial data. This is the reverse transformation of SpaceToDepth. More specifically, this op outputs a copy of the input tensor where values from the depth dimension are moved in spatial blocks to the height and width dimensions. By default, `mode = DCR`. In the DCR mode, elements along the depth dimension from

the input tensor are rearranged in the following order: depth, column, and then row. The output y is computed from the input x as below:

```
b, c, h, w = x.shape
tmp = np.reshape(x, [b, blocksize, blocksize, c // (blocksize**2), h, w])
tmp = np.transpose(tmp, [0, 3, 4, 1, 5, 2])
y = np.reshape(tmp, [b, c // (blocksize**2), h * blocksize, w * blocksize])
```

In the CRD mode, elements along the depth dimension from the input tensor are rearranged in the following order: column, row, and the depth. The output y is computed from the input x as below:

```
b, c, h, w = x.shape
tmp = np.reshape(x, [b, c // (blocksize ** 2), blocksize, blocksize, h, w])
tmp = np.transpose(tmp, [0, 1, 4, 2, 5, 3])
y = np.reshape(tmp, [b, c // (blocksize ** 2), h * blocksize, w * blocksize])
```

Domain	ai.onnx
Since version	28
Earlier versions	1, 11, 13
Inputs	input (differentiable) : T — Input tensor of [N,C,H,W], where N is the batch axis, C is the channel or depth, H is the height and W is the width.
Outputs	output (differentiable) : T — Output tensor of [N, C/(blocksize * blocksize), H * blocksize, W * blocksize].
Attributes	blocksize : int (required) — Blocks of [blocksize, blocksize] are moved. mode : string (default is DCR) — DCR (default) for depth-column-row order re-arrangement. Use CRD for column-row-depth order.
Type constraints	T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128) — Constrain input and output types to all tensor types.
Test vectors	test_depthtospace_crd_mode_example, test_depthtospace_example
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.48. DequantizeLinear

The linear dequantization operator. It consumes a quantized tensor, a scale, and a zero point to compute the full-precision tensor. The dequantization formula is $y = (x - x_zero_point) * x_scale$. x_scale and x_zero_point must have the same shape, determining the quantization's granularity: a scalar for per-tensor/per-layer quantization, a 1-D tensor for per-axis quantization, or have a rank identical to the input for blocked quantization. See QuantizeLinear for details on quantization granularity.

x_zero_point and x must have the same type. x and y must have the same shape. In the case of dequantizing int32, there's no zero point (zero point is supposed to be 0). zero-point is

usually not used in the case of float8 and 4-bit types quantization, but the dequantization formula remains the same for consistency. The output type is determined by the attribute `output_dtype`. If `output_dtype` is not supplied then the output type is the same as `x_scale`. The output type also determines the precision of the multiplication operation.

Domain	ai.onnx
Since version	28
Earlier versions	10, 13, 19, 21, 23, 24, 25
Inputs (2 — 3)	<code>x</code> : T1 — N-D quantized input tensor to be de-quantized. <code>x_scale</code> : T2 — Scale for input <code>x</code>. For per-tensor/layer dequantization the scale is a scalar, for per per-axis dequantization it is a 1-D Tensor and for blocked dequantization it has the same shape as the input, except for one dimension in which blocking is performed. <code>x_zero_point</code> (optional) : T1 — Zero point for input <code>x</code>. Shape must match <code>x_scale</code>. It's optional. Zero point is 0 when it's not specified.
Outputs	<code>y</code> : T3 — N-D full precision output tensor. It has the same shape as input <code>x</code> . The data type is specified by the <code>output_dtype</code> attribute or, in its absence, the type of <code>x_scale</code> .
Attributes	<code>axis</code> : int (default is 1) — (Optional) The axis of the dequantizing dimension of the input tensor. Used for per-axis and blocked quantization. Negative value means counting dimensions from the back. Accepted range is $[-r, r-1]$ where $r = \text{rank}(\text{input})$. <code>block_size</code> : int (default is 0) — (Optional) The size of the quantization block (number of times every scale is replicated). Used only for blocked quantization. The block size is a positive integer. Given <code>x</code> shape $(D_0, \dots, D_i, \dots, D_n)$, <code>y_scale</code> shape $(S_0, \dots, S_i, \dots, S_n)$ and <code>axis=i</code>, the accepted range is $[\text{ceil}(D_i/S_i), \text{ceil}(D_i/(S_i-1))-1]$ <code>output_dtype</code> : int (default is 0) — (Optional) The output data type. If not supplied, the output data type is inferred from <code>x_scale</code> data type (T2)
Type constraints	T1 : tensor(int8), tensor(uint8), tensor(int16), tensor(uint16), tensor(int32), tensor(float8e4m3fn), tensor(float8e4m3fnuz), tensor(float8e5m2), tensor(float8e5m2fnuz), tensor(uint4), tensor(int4), tensor(float4e2m1), tensor(uint2), tensor(int2), tensor(float6e2m3), tensor(float6e3m2) — The type of the inputs '<code>x</code>', '<code>zero_point</code>' and '<code>x</code>'. T2 : tensor(float), tensor(float16), tensor(bfloat16), tensor(float8e8m0) — The type of the input '<code>x_scale</code>'. T3 : tensor(float), tensor(float16), tensor(bfloat16) — The type of the output '<code>y</code>'.
Test vectors	test_dequantizelinear_axis, test_dequantizelinear_blocked, test_dequantizelinear, test_dequantizelinear_e4m3fn, test_dequantizelinear_e4m3fn_float16, test_dequantizelinear_e4m3fn_zero_point, test_dequantizelinear_e5m2, test_dequantizelinear_float4e2m1, test_dequantizelinear_int16, test_dequantizelinear_int2, test_dequantizelinear_int4, test_dequantizelinear_uint16, test_dequantizelinear_uint2, test_dequantizelinear_uint4
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.49. Det

Det calculates determinant of a square matrix or batches of square matrices. Det takes one input tensor of shape $[*, M, M]$, where $*$ is zero or more batch dimensions, and the inner-most 2 dimensions form square matrices. The output is a tensor of shape $[*]$, containing the determinants of all input submatrices. e.g., When the input is 2-D, the output is a scalar(shape is empty: []).

Domain	ai.onnx
Since version	22
Earlier versions	11
Inputs	X (differentiable) : T — Input tensor
Outputs	Y (differentiable) : T — Output tensor
Attributes	None.
Type constraints	T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain input and output types to floating-point tensors.
Test vectors	test_det_2d, test_det_nd
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.50. Div

Performs element-wise binary division (with Numpy-style broadcasting support).

This operator supports **multidirectional (i.e., Numpy-style) broadcasting**; for more details please check the doc.

For integer inputs, the result is computed using truncating division (rounding toward zero). (Opset 14 change): Extend supported types to include uint8, int8, uint16, and int16.

Domain	ai.onnx
Since version	14
Earlier versions	1, 6, 7, 13
Inputs	A (differentiable) : T — First operand. B (differentiable) : T — Second operand.
Outputs	C (differentiable) : T — Result, has same element type as two inputs
Attributes	None.
Type constraints	T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(float16), tensor(float),

	tensor(double), tensor(bfloat16) — Constrain input and output types to all numeric tensors.
Test vectors	test_div_example, test_div, test_div_int8, test_div_int16, test_div_int32_trunc, test_div_uint8, test_div_uint16, test_div_uint32, test_div_uint64, test_div_bcast
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.51. Dropout

Dropout takes an input floating-point tensor, an optional input ratio (floating-point scalar) and an optional input training_mode (boolean scalar). It produces two tensor outputs, output (floating-point tensor) and mask (optional Tensor<bool>). If training_mode is true then the output Y will be a random dropout; Note that this Dropout scales the masked input data by the following equation, so to convert the trained model into inference mode, the user can simply not pass training_mode input or set it to false.

$$\text{output} = \text{scale} * \text{data} * \text{mask},$$

where

$$\text{scale} = 1. / (1. - \text{ratio}).$$

This operator has **optional** inputs/outputs. See the doc for more details about the representation of optional arguments. An empty string may be used in the place of an actual argument's name to indicate a missing argument. Trailing optional arguments (those not followed by an argument that is present) may also be simply omitted.

Domain	ai.onnx
Since version	22
Earlier versions	1, 6, 7, 10, 12, 13
Inputs (1 — 3)	data (differentiable) : T — The input data as Tensor. ratio (optional, non-differentiable) : T1 — The ratio of random dropout, with value in [0, 1). If set to 0, the output would be a simple copy of the input. If it's non-zero, output will be a random dropout of the scaled input, which is typically the case during training. It is an optional value, if not specified it will default to 0.5. training_mode (optional, non-differentiable) : T2 — If set to true then it indicates dropout is being used for training. It is an optional value hence unless specified explicitly, it is false. If it is false, ratio is ignored and the operation mimics inference mode where nothing will be dropped from the input data and if mask is requested as output it will contain all ones.
Outputs (1 — 2)	output (differentiable) : T — The output. mask (optional, non-differentiable) : T2 — The output mask.
Attributes	seed : int — (Optional) Seed to the random generator, if not specified we will auto generate one.
Type constraints	T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(float8e4m3fn), tensor(float8e4m3fnuz), tensor(float8e5m2),

tensor(float8e5m2fnuz) — Constrain input and output types to float tensors.
 T1 : tensor(bfloat16), tensor(float16),
 tensor(float), tensor(double), tensor(float8e4m3fn),
 tensor(float8e4m3fnuz), tensor(float8e5m2),
 tensor(float8e5m2fnuz) — Constrain input 'ratio' types to float tensors.
 T2 : tensor(bool) — Constrain output 'mask' types to boolean tensors.

Test vectors test_dropout_default, test_dropout_default_mask, test_dropout_default_mask_ratio, test_dropout_default_old, test_dropout_default_ratio, test_dropout_random_old, test_training_dropout, test_training_dropout_default, test_training_dropout_default_mask, test_training_dropout_zero_ratio, test_training_dropout_zero_ratio_mask, test_training_dropout_mask

Not supplied by the upstream source Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.52. DynamicQuantizeLinear

A Function to fuse calculation for Scale, Zero Point and FP32->8Bit conversion of FP32 Input data. Outputs Scale, ZeroPoint and Quantized Input for a given FP32 Input. Scale is calculated as:

`y_scale = (maximum(0, max(x)) - minimum(0, min(x))) / (qmax - qmin)`

- where qmax and qmin are max and min values for quantization range i.e. [0, 255] in case of uint8
- data range is adjusted to include 0.

Zero point is calculated as:

`intermediate_zero_point = qmin - min(x)/y_scale`
`y_zero_point = cast(round(saturate(intermediate_zero_point)))`

- where qmax and qmin are max and min values for quantization range .i.e [0, 255] in case of uint8
- for saturation, it saturates to [0, 255] if it's uint8, or [-127, 127] if it's int8. Right now only uint8 is supported.
- rounding to nearest ties to even.

Data quantization formula is:

`y = saturate (round (x / y_scale) + y_zero_point)`

- for saturation, it saturates to [0, 255] if it's uint8, or [-127, 127] if it's int8. Right now only uint8 is supported.
- rounding to nearest ties to even.

Domain	ai.onnx
Since version	11
Inputs	x : T1 — Input tensor
Outputs	y : T2 — Quantized output tensor
	y_scale : tensor(float) — Output scale. It's a scalar, which means a per-tensor/layer quantization.
	y_zero_point : T2 — Output zero point. It's a scalar, which means a per-tensor/layer quantization.
Attributes	None.
Type constraints	T1 : tensor(float) — Constrain 'x' to float tensor.
	T2 : tensor(uint8) — Constrain 'y_zero_point' and 'y' to 8-bit unsigned integer tensor.

Test vectors	<code>test_dynamicquantizelinear, test_dynamicquantizelinear_max_adjusted, test_dynamicquantizelinear_min_adjusted</code>
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.53. Einsum

An einsum of the form `term1, term2 -> output-term` produces an output tensor using the following equation

```
output[output-term] = reduce-sum( input1[term1] * input2[term2] )
```

where the reduce-sum performs a summation over all the indices occurring in the input terms (`term1`, `term2`) that do not occur in the `output-term`.

The Einsum operator evaluates algebraic tensor operations on a sequence of tensors, using the Einstein summation convention. The equation string contains a comma-separated sequence of lower case letters and/or upper case letters. Each term corresponds to an operand tensor, and the characters within the terms correspond to operands dimensions. Lower case letters and upper case letters are treated as distinct symbols, that is, "a" and "A" refer to different symbols.

This sequence may be followed by "->" to separate the left and right hand side of the equation. If the equation contains "->" followed by the right-hand side, the explicit (not classical) form of the Einstein summation is performed, and the right-hand side indices indicate output tensor dimensions. In other cases, output indices are (implicitly) set to the sequence of indices appearing exactly once in the equation, sorted in increasing order of their ASCII values (so that all upper case letters precede all lower case letters, e.g., "A" < "Z" < "a" < "z").

When a dimension character is repeated in the left-hand side, it represents summation along the dimension.

The equation may contain ellipsis ("...") to enable broadcasting. Ellipsis must indicate a fixed number of dimensions. Specifically, every occurrence of ellipsis in the equation must represent the same number of dimensions. The right-hand side may contain exactly one ellipsis. In implicit mode, the ellipsis dimensions are set to the beginning of the output. The equation string may contain space (U+0020) character.

Domain	<code>ai.onnx</code>
Since version	28
Earlier versions	12
Inputs (1 — unbounded)	Inputs (variadic, differentiable) : T — Operands
Outputs	Output (differentiable) : T — Output tensor
Attributes	<code>equation</code> : string (required) — Einsum expression string.
Type constraints	T : <code>tensor(uint8)</code> , <code>tensor(uint16)</code> , <code>tensor(uint32)</code> , <code>tensor(uint64)</code> , <code>tensor(int8)</code> , <code>tensor(int16)</code> , <code>tensor(int32)</code> , <code>tensor(int64)</code> , <code>tensor(float16)</code> , <code>tensor(float)</code> , <code>tensor(double)</code> , <code>tensor(bfloat16)</code> — Constrain input and output types to all numerical tensor types.
Test vectors	<code>test_einsum_batch_diagonal</code> , <code>test_einsum_batch_matmul</code> , <code>test_einsum_batch_matmul_bfloat16</code> , <code>test_einsum_inner_prod</code> , <code>test_einsum_scalar</code> , <code>test_einsum_sum</code> , <code>test_einsum_sum_bfloat16</code> , <code>test_einsum_transpose</code> , <code>test_einsum_transpose_bfloat16</code>

Not supplied by the upstream source Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.54. Elu

Elu takes one input data (Tensor<T>) and produces one output data (Tensor<T>) where the function $f(x) = \alpha * (\exp(x) - 1.)$ for $x < 0$, $f(x) = x$ for $x \geq 0$, is applied to the tensor elementwise.

Domain	ai.onnx
Since version	22
Earlier versions	1, 6
Inputs	X (differentiable) : T—Input tensor
Outputs	Y (differentiable) : T—Output tensor
Attributes	alpha : float (default is 1.0) — Coefficient of ELU.
Type constraints	T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain input and output types to float tensors.
Test vectors	test_elu_example, test_elu, test_elu_default
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.55. Equal

Returns the tensor resulted from performing the equal logical operation elementwise on the input tensors A and B (with Numpy-style broadcasting support).

This operator supports **multidirectional (i.e., Numpy-style) broadcasting**; for more details please check the doc.

Domain	ai.onnx
Since version	19
Earlier versions	1, 7, 11, 13
Inputs	A (non-differentiable) : T—First input operand for the logical operator. B (non-differentiable) : T—Second input operand for the logical operator.
Outputs	C (non-differentiable) : T1—Result tensor.
Attributes	None.
Type constraints	T : tensor(bool), tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(float16), tensor(float), tensor(double),

	<code>tensor(bfloat16), tensor(string)</code> — Constrain input types to all (non-complex) tensors. <code>T1 : tensor(bool)</code> — Constrain output to boolean tensor.
Test vectors	<code>test_equal, test_equal_int8, test_equal_int16, test_equal_uint8, test_equal_uint16, test_equal_uint32, test_equal_uint64, test_equal_bcast, test_equal_string, test_equal_string_broadcast</code>
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.56. Erf

Computes the error function of the given input tensor element-wise.

Domain	<code>ai.onnx</code>
Since version	13
Earlier versions	9
Inputs	<code>input (differentiable) : T</code> — Input tensor
Outputs	<code>output (differentiable) : T</code> — The error function of the input tensor computed element-wise. It has the same shape and type of the input.
Attributes	None.
Type constraints	<code>T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double)</code> — Constrain input and output types to float tensors.
Test vectors	<code>test_erf</code>
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.57. Exp

Calculates the exponential of the given input tensor, element-wise.

Domain	<code>ai.onnx</code>
Since version	13
Earlier versions	1, 6
Inputs	<code>input (differentiable) : T</code> — Input tensor
Outputs	<code>output (differentiable) : T</code> — The exponential of the input tensor computed element-wise
Attributes	None.

Type constraints	T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain input and output types to float tensors.
Test vectors	test_exp_example, test_exp
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.58. Expand

Broadcast the input tensor following the given shape and the broadcast rule. The broadcast rule is similar to `numpy.array(input) * numpy.ones(shape)`: Dimensions are right alignment; Two corresponding dimensions must have the same value, or one of them is equal to 1. Also, this operator is similar to `numpy.broadcast_to(input, shape)`, but the major difference is `numpy.broadcast_to()` does not allow shape to be smaller than `input.size()`. It is possible that the output.shape is not equal to shape, when some dimensions in shape is equal to 1, or the `shape.ndim < input.shape.ndim`.

Domain	ai.onnx
Since version	13
Earlier versions	8
Inputs	input (differentiable) : T — Input tensor shape (non-differentiable) : tensor(int64) — A 1-D tensor indicates the shape you want to expand to, following the broadcast rule
Outputs	output (differentiable) : T — Output tensor
Attributes	None.
Type constraints	T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128) — Constrain input and output types to all tensors.
Test vectors	test_expand_dim_changed, test_expand_dim_unchanged
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.59. EyeLike

Generate a 2D tensor (matrix) with ones on the diagonal and zeros everywhere else. Only 2D tensors are supported, i.e. input T1 must be of rank 2. The shape of the output tensor is the same as the input tensor. The data type can be specified by the 'dtype' argument. If 'dtype' is not specified, then the type of input tensor is used. By default, the main diagonal is populated with ones, but attribute 'k' can be used to populate upper or lower diagonals. The 'dtype' argument must be one of the data types specified in the 'DataType' enum field in the TensorProto message and be valid as an output type.

ONNX 1-2 (unofficial draft)

Domain	ai.onnx
Since version	22
Earlier versions	9
Inputs	input : T1 — 2D input tensor to copy shape, and optionally, type information from.
Outputs	output : T2 — Output tensor, same shape as input tensor T1.
Attributes	dtype : int — (Optional) The data type for the elements of the output tensor. If not specified, the data type of the input tensor T1 is used. k : int (default is 0) — (Optional) Index of the diagonal to be populated with ones. Default is 0. If T2 is the output, this op sets $T2[i, i+k] = 1$. $k = 0$ populates the main diagonal, $k > 0$ populates an upper diagonal, and $k < 0$ populates a lower diagonal.
Type constraints	T1 : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(bool) — Constrain input types. Strings and complex are not supported. T2 : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(bool) — Constrain output types. Strings and complex are not supported.
Test vectors	test_eyelike_populate_off_main_diagonal, test_eyelike_with_dtype, test_eyelike_without_dtype
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.60. Flatten

Flattens the input tensor into a 2D matrix. If input tensor has shape $(d_0, d_1, \dots, d_n)$ then the output will have shape $(d_0 \times d_1 \dots d_{(axis-1)}, d_{axis} \times d_{(axis+1)} \dots \times d_n)$.

Domain	ai.onnx
Since version	25
Earlier versions	1, 9, 11, 13, 21, 23, 24
Inputs	input (differentiable) : T — A tensor of rank $\geq$ axis.
Outputs	output (differentiable) : T — A 2D tensor with the contents of the input tensor, with input dimensions up to axis flattened to the outer dimension of the output and remaining input dimensions flattened into the inner dimension of the output.
Attributes	axis : int (default is 1) — Indicate up to which input dimensions (exclusive) should be flattened to the outer dimension of the output. The value for axis must be in the range $[-r, r]$, where r is the rank of the input tensor. Negative value means counting dimensions from the back. When

axis = 0, the shape of the output tensor is (1, (d_0 X d_1 ... d_n), where the shape of the input tensor is (d_0, d_1, ... d_n).

Type constraints T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128), tensor(float8e4m3fn), tensor(float8e4m3fnuz), tensor(float8e5m2), tensor(float8e5m2fnuz), tensor(uint4), tensor(int4), tensor(float4e2m1), tensor(float8e8m0), tensor(uint2), tensor(int2) — Constrain input and output to all tensor types up to IRv13.

Test vectors test_flatten_axis, test_flatten_negative_axis, test_flatten_default_axis

Not supplied by the upstream source Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.61. Floor

Floor takes one input data (Tensor<T>) and produces one output data (Tensor<T>) where the floor is, $y = \text{floor}(x)$, is applied to the tensor elementwise. If x is integral, +0, -0, NaN, or infinite, x itself is returned.

Domain	ai.onnx
Since version	13
Earlier versions	1, 6
Inputs	X (non-differentiable) : T — Input tensor
Outputs	Y (non-differentiable) : T — Output tensor
Attributes	None.
Type constraints	T : tensor(float16), tensor(float), tensor(double), tensor(bfloat16) — Constrain input and output types to float tensors.
Test vectors	test_floor_example, test_floor
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.62. GRU

Computes an one-layer GRU. This operator is usually supported via some custom implementation such as CuDNN.

Notations:

- x — input tensor
- z — update gate

- r — reset gate
- h — hidden gate
- t — time step ($t-1$ means previous time step)
- $W[zrh]$ — W parameter weight matrix for update, reset, and hidden gates
- $R[zrh]$ — R recurrence weight matrix for update, reset, and hidden gates
- $Wb[zrh]$ — W bias vectors for update, reset, and hidden gates
- $Rb[zrh]$ — R bias vectors for update, reset, and hidden gates
- $WB[zrh]$ — W parameter weight matrix for backward update, reset, and hidden gates
- $RB[zrh]$ — R recurrence weight matrix for backward update, reset, and hidden gates
- $WBb[zrh]$ — W bias vectors for backward update, reset, and hidden gates
- $RBb[zrh]$ — R bias vectors for backward update, reset, and hidden gates
- H — Hidden state
- $num_directions$ — 2 if $direction == bidirectional$ else 1

Activation functions:

- $Relu(x)$ — $\max(0, x)$
- $Tanh(x)$ — $(1 - e^{-2x}) / (1 + e^{-2x})$
- $Sigmoid(x)$ — $1 / (1 + e^{-x})$

NOTE: Below are optional

- $Affine(x)$ — $\alpha * x + \beta$
- $LeakyRelu(x)$ — x if $x \geq 0$ else $\alpha * x$
- $ThresholdedRelu(x)$ — x if $x \geq \alpha$ else 0
- $ScaledTanh(x)$ — $\alpha * \tanh(\beta * x)$
- $HardSigmoid(x)$ — $\min(\max(\alpha * x + \beta, 0), 1)$
- $Elu(x)$ — x if $x \geq 0$ else $\alpha * (e^x - 1)$

Table 5 — Table from the upstream description of GRU

- $Softsign(x)$ — $x / (1 + |x|)$
- $Softplus(x)$ — $\log(1 + e^x)$

Equations (Default: $f=Sigmoid$, $g=Tanh$):

- $z_t = f(X_t(W_z^T) + H_{t-1}(R_z^T) + W_{bz} + R_{bz})$
- $r_t = f(X_t(W_r^T) + H_{t-1}(R_r^T) + W_{br} + R_{br})$
- $h_t = g(X_t(W_h^T) + (r_t \odot H_{t-1})(R_h^T) + R_{bh} + W_{bh})$ # default, when $linear_before_reset = 0$
- $h_t = g(X_t(W_h^T) + (r_t \odot (H_{t-1}(R_h^T) + R_{bh})) + W_{bh})$ # when $linear_before_reset \neq 0$
- $H_t = (1 - z_t) \odot h_t + z_t \odot H_{t-1}$ This operator has **optional** inputs/outputs. See the doc for more details about the representation of optional arguments. An empty string may be used in the place of an actual argument's name to indicate a missing argument. Trailing optional arguments (those not followed by an argument that is present) may also be simply omitted.

Domain `ai.onnx`

Since version 22

Earlier 1, 3, 7, 14

versions

Inputs (3 — 6) X (differentiable) : T — The input sequences packed (and potentially padded) into one 3-D tensor with the shape of $[seq_length, batch_size, input_size]$.

W (differentiable) : T — The weight tensor for the gates.

Concatenation of $W[zrh]$ and $WB[zrh]$ (if $bidirectional$) along dimension 0. This tensor has shape $[num_directions, 3*hidden_size, input_size]$.

R (differentiable) : T — The recurrence weight tensor.

Concatenation of $R[zrh]$ and $RB[zrh]$ (if $bidirectional$) along dimension 0. This tensor has shape $[num_directions, 3*hidden_size, hidden_size]$.

	B (optional, differentiable) : T — The bias tensor for the gates. Concatenation of [Wb[zrh], Rb[zrh]] and [WBb[zrh], RBb[zrh]] (if bidirectional) along dimension 0. This tensor has shape [num_directions, 6*hidden_size]. Optional: If not specified — assumed to be 0 sequence_lens (optional, non-differentiable) : T1 — Optional tensor specifying lengths of the sequences in a batch. If not specified — assumed all sequences in the batch to have length seq_length. It has shape [batch_size]. initial_h (optional, non-differentiable) : T — Optional initial value of the hidden. If not specified — assumed to be 0. It has shape [num_directions, batch_size, hidden_size].
Outputs (0 — 2)	Y (optional, differentiable) : T — A tensor that concatenates all the intermediate output values of the hidden. It has shape [seq_length, num_directions, batch_size, hidden_size]. Y_h (optional, differentiable) : T — The last output value of the hidden. It has shape [num_directions, batch_size, hidden_size].
Attributes	activation_alpha : list of floats — Optional scaling values used by some activation functions. The values are consumed in the order of activation functions, for example (f, g, h) in LSTM. Default values are the same as of corresponding ONNX operators. For example with LeakyRelu, the default alpha is 0.01. activation_beta : list of floats — Optional scaling values used by some activation functions. The values are consumed in the order of activation functions, for example (f, g, h) in LSTM. Default values are the same as of corresponding ONNX operators. activations : list of strings — A list of 2 (or 4 if bidirectional) activation functions for update, reset, and hidden gates. The activation functions must be one of the activation functions specified above. Optional: See the equations for default if not specified. clip : float — Cell clip threshold. Clipping bounds the elements of a tensor in the range of [-threshold, +threshold] and is applied to the input of activations. No clip if not specified. direction : string (default is forward) — Specify if the RNN is forward, reverse, or bidirectional. Must be one of forward (default), reverse, or bidirectional. hidden_size : int — Number of neurons in the hidden layer layout : int (default is 0) — The shape format of inputs X, initial_h and outputs Y, Y_h. If 0, the following shapes are expected: X.shape = [seq_length, batch_size, input_size], Y.shape = [seq_length, num_directions, batch_size, hidden_size], initial_h.shape = Y_h.shape = [num_directions, batch_size, hidden_size]. If 1, the following shapes are expected: X.shape = [batch_size, seq_length, input_size], Y.shape = [batch_size, seq_length, num_directions, hidden_size], initial_h.shape = Y_h.shape = [batch_size, num_directions, hidden_size]. linear_before_reset : int (default is 0) — When computing the output of the hidden gate, apply the linear transformation before multiplying by the output of the reset gate.
Type constraints	T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain input and output types to float tensors. T1 : tensor(int32) — Constrain seq_lens to integer tensor.
Test vectors	test_gru_batchwise, test_gru_bidirectional, test_gru_defaults, test_gru_with_initial_bias, test_gru_reverse, test_gru_seq_length
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.63. Gather

Given data tensor of rank $r \geq 1$, and indices tensor of rank q , gather entries of the axis dimension of data (by default outer-most one as $\text{axis}=0$) indexed by indices, and concatenates them in an output tensor of rank $q + (r - 1)$.

It is an indexing operation that indexes into the input data along a single (specified) axis. Each entry in indices produces a $r-1$ dimensional slice of the input tensor. The entire operation produces, conceptually, a q -dimensional tensor of $r-1$ dimensional slices, which is arranged into a $q + (r-1)$ -dimensional tensor, with the q dimensions taking the place of the original axis that is being indexed into.

The following few examples illustrate how Gather works for specific shapes of data, indices, and given value of axis:

data shape	indices shape	axis	output shape	output equation
(P, Q)	$()$ (a scalar)	0	(Q)	$\text{output}[q] = \text{data}[\text{indices}, q]$
(P, Q, R)	$()$ (a scalar)	1	(P, R)	$\text{output}[p, r] = \text{data}[p, \text{indices}, r]$
(P, Q)	(R, S)	0	(R, S, Q)	$\text{output}[r, s, q] = \text{data}[\text{indices}[r, s], q]$
(P, Q)	(R, S)	1	(P, R, S)	$\text{output}[p, r, s] = \text{data}[p, \text{indices}[r, s]]$

More generally, if $\text{axis} = 0$, let $k = \text{indices}[i_{\{0\}}, \dots, i_{\{q-1\}}]$ then $\text{output}[i_{\{0\}}, \dots, i_{\{q-1\}}, j_{\{0\}}, \dots, j_{\{r-2\}}] = \text{input}[k, j_{\{0\}}, \dots, j_{\{r-2\}}]$:

```
data = [
    [1.0, 1.2],
    [2.3, 3.4],
    [4.5, 5.7],
]
indices = [
    [0, 1],
    [1, 2],
]
output = [
    [
        [1.0, 1.2],
        [2.3, 3.4],
    ],
    [
        [2.3, 3.4],
        [4.5, 5.7],
    ],
]
```

If $\text{axis} = 1$, let $k = \text{indices}[i_{\{0\}}, \dots, i_{\{q-1\}}]$ then $\text{output}[j_{\{0\}}, i_{\{0\}}, \dots, i_{\{q-1\}}, j_{\{1\}}, \dots, j_{\{r-2\}}] = \text{input}[j_{\{0\}}, k, j_{\{1\}}, \dots, j_{\{r-2\}}]$:

```
data = [
    [1.0, 1.2, 1.9],
    [2.3, 3.4, 3.9],
    [4.5, 5.7, 5.9],
]
indices = [
    [0, 2],
]
axis = 1,
output = [
    [[1.0, 1.9]],
    [[2.3, 3.9]],
    [[4.5, 5.9]],
]
```

]

Domain	ai.onnx
Since version	13
Earlier versions	1, 11
Inputs	data (differentiable) : T — Tensor of rank $r \geq 1$. indices (non-differentiable) : Tind — Tensor of int32/int64 indices, of any rank q. All index values are expected to be within bounds $[-s, s-1]$ along axis of size s. It is an error if any of the index values are out of bounds.
Outputs	output (differentiable) : T — Tensor of rank $q + (r - 1)$.
Attributes	axis : int (default is 0) — Which axis to gather on. Negative value means counting dimensions from the back. Accepted range is $[-r, r-1]$ where $r = \text{rank}(\text{data})$.
Type constraints	T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128) — Constrain input and output types to any tensor type. Tind : tensor(int32), tensor(int64) — Constrain indices to integer types
Test vectors	test_gather_0, test_gather_1, test_gather_2d_indices, test_gather_negative_indices
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.64. GatherElements

GatherElements takes two inputs data and indices of the same rank $r \geq 1$ and an optional attribute axis that identifies an axis of data (by default, the outer-most axis, that is axis 0). It is an indexing operation that produces its output by indexing into the input data tensor at index positions determined by elements of the indices tensor. Its output shape is the same as the shape of indices and consists of one value (gathered from the data) for each element in indices.

For instance, in the 3-D case ($r = 3$), the output produced is determined by the following equations:

```

out[i][j][k] = input[index[i][j][k]][j][k] if axis = 0,
out[i][j][k] = input[i][index[i][j][k]][k] if axis = 1,
out[i][j][k] = input[i][j][index[i][j][k]] if axis = 2,

```

This operator is also the inverse of ScatterElements. It is similar to Torch's gather operation.

Example 1:

```

data = [
    [1, 2],
    [3, 4],
]
indices = [

```

```

    [0, 0],
    [1, 0],
]
axis = 1
output = [
    [1, 1],
    [4, 3],
]

```

Example 2:

```

data = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9],
]
indices = [
    [1, 2, 0],
    [2, 0, 0],
]
axis = 0
output = [
    [4, 8, 3],
    [7, 2, 3],
]

```

Domain	ai.onnx
Since version	13
Earlier versions	11
Inputs	data (differentiable) : T — Tensor of rank $r \geq 1$. indices (non-differentiable) : Tind — Tensor of int32/int64 indices, with the same rank r as the input. All index values are expected to be within bounds $[-s, s-1]$ along axis of size s. It is an error if any of the index values are out of bounds.
Outputs	output (differentiable) : T — Tensor of the same shape as indices.
Attributes	axis : int (default is 0) — Which axis to gather on. Negative value means counting dimensions from the back. Accepted range is $[-r, r-1]$ where $r = \text{rank}(\text{data})$.
Type constraints	T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128) — Constrain input and output types to any tensor type. Tind : tensor(int32), tensor(int64) — Constrain indices to integer types
Test vectors	test_gather_elements_0, test_gather_elements_1, test_gather_elements_negative_indices
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.65. GatherND

Given data tensor of rank $r \geq 1$, indices tensor of rank $q \geq 1$, and batch_dims integer b , this operator gathers slices of data into an output tensor of rank $q + r - \text{indices_shape}[-1] - 1 - b$.

indices is an q -dimensional integer tensor, best thought of as a $(q-1)$ -dimensional tensor of index-tuples into data, where each element defines a slice of data

batch_dims (denoted as b) is an integer indicating the number of batch dimensions, i.e the leading b number of dimensions of data tensor and indices are representing the batches, and the gather starts from the $b+1$ dimension.

Some salient points about the inputs' rank and shape:

- 1) $r \geq 1$ and $q \geq 1$ are to be honored. There is no dependency condition to be met between ranks r and q
- 2) The first b dimensions of the shape of indices tensor and data tensor must be equal.
- 3) $b < \min(q, r)$ is to be honored.
- 4) The $\text{indices_shape}[-1]$ should have a value between 1 (inclusive) and rank $r-b$ (inclusive)
- 5) All values in indices are expected to be within bounds $[-s, s-1]$ along axis of size s (i.e.) $-\text{data_shape}[i] \leq \text{indices}[\dots, i] \leq \text{data_shape}[i] - 1$. It is an error if any of the index values are out of bounds.

The output is computed as follows:

The output tensor is obtained by mapping each index-tuple in the indices tensor to the corresponding slice of the input data.

- 1) If $\text{indices_shape}[-1] > r-b \Rightarrow$ error condition
- 2) If $\text{indices_shape}[-1] == r-b$, since the rank of indices is q , indices can be thought of as N $(q-b-1)$ -dimensional tensors containing 1-D tensors of dimension $r-b$, where N is an integer equals to the product of 1 and all the elements in the batch dimensions of the indices_shape. Let us think of each such $r-b$ ranked tensor as *indices_slice*. Each *scalar value* corresponding to $\text{data}[0:b-1, \text{indices_slice}]$ is filled into the corresponding location of the $(q-b-1)$ -dimensional tensor to form the output tensor (Example 1 below)
- 3) If $\text{indices_shape}[-1] < r-b$, since the rank of indices is q , indices can be thought of as N $(q-b-1)$ -dimensional tensor containing 1-D tensors of dimension $< r-b$. Let us think of each such tensors as *indices_slice*. Each *tensor slice* corresponding to $\text{data}[0:b-1, \text{indices_slice}, :]$ is filled into the corresponding location of the $(q-b-1)$ -dimensional tensor to form the output tensor (Examples 2, 3, 4 and 5 below)

This operator is the inverse of ScatterND.

Example 1

```
batch_dims = 0
data       = [[0,1],[2,3]]   # data_shape      = [2, 2]
indices    = [[0,0],[1,1]]   # indices_shape = [2, 2]
output     = [0,3]           # output_shape  = [2]
```

Example 2

```
batch_dims = 0
data       = [[0,1],[2,3]] # data_shape      = [2, 2]
indices    = [[1],[0]]     # indices_shape = [2, 1]
output     = [[2,3],[0,1]] # output_shape = [2, 2]
```

Example 3

```
batch_dims = 0
data       = [[[0,1],[2,3]],[[4,5],[6,7]]] # data_shape      = [2, 2, 2]
indices    = [[0,1],[1,0]]                 # indices_shape = [2, 2]
output     = [[2,3],[4,5]]                 # output_shape = [2, 2]
```

Example 4

```
batch_dims = 0
data       = [[[0,1],[2,3]],[[4,5],[6,7]]] # data_shape      = [2, 2, 2]
indices    = [[[0,1]],[[1,0]]]             # indices_shape = [2, 1, 2]
output     = [[[2,3]],[[4,5]]]             # output_shape = [2, 1, 2]
```

Example 5

```
batch_dims = 1
data       = [[[0,1],[2,3]],[[4,5],[6,7]]] # data_shape      = [2, 2, 2]
indices    = [[1],[0]]                     # indices_shape = [2, 1]
output     = [[2,3],[4,5]]                 # output_shape = [2, 2]
```

Domain ai.onnx

Since version 13

Earlier versions 11, 12

Inputs data (differentiable) : T — Tensor of rank $r \geq 1$.
 indices (non-differentiable) : tensor(int64) — Tensor of rank $q \geq 1$. All index values are expected to be within bounds $[-s, s-1]$ along axis of size s . It is an error if any of the index values are out of bounds.

Outputs output (differentiable) : T — Tensor of rank $q + r$ — indices_shape[-1] — 1.

Attributes batch_dims : int (default is 0) — The number of batch dimensions. The gather of indexing starts from dimension of data[batch_dims:].

Type constraints T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128) — Constrain input and output types to any tensor type.

Test vectors test_gathernd_example_float32, test_gathernd_example_int32, test_gathernd_example_int32_batch_dim1

Not supplied by the upstream source Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.66. Gelu

Gelu takes one input data (Tensor<T>) and produces one output data (Tensor<T>) where the gaussian error linear units function, $y = 0.5 * x * (1 + \text{erf}(x/\sqrt{2}))$ is applied to the tensor elementwise. If the attribute "approximate" is set to "tanh", the function estimation, $y = 0.5 * x * (1 + \text{Tanh}(\sqrt{2/\pi}) * (x + 0.044715 * x^3))$ is used and applied to the tensor elementwise.

Domain	ai.onnx
Since version	20
Inputs	X (differentiable) : T—Input tensor
Outputs	Y (differentiable) : T—Output tensor
Attributes	approximate : string (default is none) — Gelu approximation algorithm: "tanh", "none"(default). "none": do not use approximation. "tanh": use tanh approximation.
Type constraints	T : tensor(float16), tensor(float), tensor(double), tensor(bfloat16) — Constrain input and output types to float tensors.
Test vectors	test_gelu_default_1, test_gelu_default_2, test_gelu_tanh_1, test_gelu_tanh_2
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.67. Gemm

General Matrix multiplication: https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms#Level_3

- $A' = \text{transpose}(A)$ if transA else A
- $B' = \text{transpose}(B)$ if transB else B

Compute $Y = \alpha * A' * B' + \beta * C$, where input tensor A has shape (M, K) or (K, M), input tensor B has shape (K, N) or (N, K), input tensor C is broadcastable to shape (M, N), and output tensor Y has shape (M, N). A will be transposed before doing the computation if attribute transA is non-zero, same for B and transB. This operator supports **unidirectional broadcasting** (tensor C should be unidirectional broadcastable to tensor $A * B$); for more details please check the doc. This operator has **optional** inputs/outputs. See the doc for more details about the representation of optional arguments. An empty string may be used in the place of an actual argument's name to indicate a missing argument. Trailing optional arguments (those not followed by an argument that is present) may also be simply omitted.

Domain	ai.onnx
Since version	13
Earlier versions	1, 6, 7, 9, 11
Inputs (2 — 3)	A (differentiable) : T—Input tensor A. The shape of A should be (M, K) if transA is 0, or (K, M) if transA is non-zero. B (differentiable) : T—Input tensor B. The shape of B should be (K, N) if transB is 0, or (N, K) if transB is non-zero.

	C (optional, differentiable) : T — Optional input tensor C. If not specified, the computation is done as if C is a scalar 0. The shape of C should be unidirectional broadcastable to (M, N).
Outputs	Y (differentiable) : T — Output tensor of shape (M, N).
Attributes	alpha : float (default is 1.0) — Scalar multiplier for the product of input tensors A * B. beta : float (default is 1.0) — Scalar multiplier for input tensor C. transA : int (default is 0) — Whether A should be transposed transB : int (default is 0) — Whether B should be transposed
Type constraints	T : tensor(float16), tensor(float), tensor(double), tensor(uint32), tensor(uint64), tensor(int32), tensor(int64), tensor(bfloat16) — Constrain input and output types to float/int tensors.
Test vectors	test_gemm_all_attributes, test_gemm_alpha, test_gemm_beta, test_gemm_default_matrix_bias, test_gemm_default_no_bias, test_gemm_default_scalar_bias, test_gemm_default_single_elem_vector_bias, test_gemm_default_vector_bias, test_gemm_default_zero_bias, test_gemm_transposeA, test_gemm_transposeB
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.68. GlobalAveragePool

GlobalAveragePool consumes an input tensor X and applies average pooling across the values in the same channel. This is equivalent to AveragePool with kernel size equal to the spatial dimension of input tensor.

Domain	ai.onnx
Since version	22
Earlier versions	1
Inputs	X (differentiable) : T — Input data tensor from the previous operator; dimensions for image case are (N x C x H x W), where N is the batch size, C is the number of channels, and H and W are the height and the width of the data. For non image case, the dimensions are in the form of (N x C x D1 x D2 ... Dn), where N is the batch size.
Outputs	Y (differentiable) : T — Output data tensor from pooling across the input tensor. The output tensor has the same rank as the input. The first two dimensions of output shape are the same as the input (N x C), while the other dimensions are all 1.
Attributes	None.
Type constraints	T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain input and output types to float tensors.
Test vectors	test_globalaveragepool, test_globalaveragepool_precomputed

Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.
-------------------------------------	--

5.69. GlobalLpPool

GlobalLpPool consumes an input tensor X and applies lp pool pooling across the values in the same channel. This is equivalent to LpPool with kernel size equal to the spatial dimension of input tensor.

Domain	ai.onnx
Since version	22
Earlier versions	1, 2
Inputs	X (differentiable) : T — Input data tensor from the previous operator; dimensions for image case are (N x C x H x W), where N is the batch size, C is the number of channels, and H and W are the height and the width of the data. For non image case, the dimensions are in the form of (N x C x D1 x D2 ... Dn), where N is the batch size.
Outputs	Y (differentiable) : T — Output data tensor from pooling across the input tensor. The output tensor has the same rank as the input. The first two dimensions of output shape are the same as the input (N x C), while the other dimensions are all 1.
Attributes	p : int (default is 2) — p value of the Lp norm used to pool over the input data.
Type constraints	T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain input and output types to float tensors.
Test vectors	None published for this operator.
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.70. GlobalMaxPool

GlobalMaxPool consumes an input tensor X and applies max pooling across the values in the same channel. This is equivalent to MaxPool with kernel size equal to the spatial dimension of input tensor.

Domain	ai.onnx
Since version	22
Earlier versions	1
Inputs	X (differentiable) : T — Input data tensor from the previous operator; dimensions for image case are (N x C x H x W), where N is the batch size, C is the number of channels, and H and W are the height and the width of the data. For non image case, the dimensions are in the form of (N x C x D1 x D2 ... Dn), where N is the batch size.

Outputs	Y (differentiable) : T — Output data tensor from pooling across the input tensor. The output tensor has the same rank as the input. The first two dimensions of output shape are the same as the input (N x C), while the other dimensions are all 1.
Attributes	None.
Type constraints	T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain input and output types to float tensors.
Test vectors	test_globalmaxpool, test_globalmaxpool_precomputed
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.71. Greater

Returns the tensor resulted from performing the greater logical operation elementwise on the input tensors A and B (with Numpy-style broadcasting support).

This operator supports **multidirectional (i.e., Numpy-style) broadcasting**; for more details please check the doc.

Domain	ai.onnx
Since version	13
Earlier versions	1, 7, 9
Inputs	A (non-differentiable) : T — First input operand for the logical operator. B (non-differentiable) : T — Second input operand for the logical operator.
Outputs	C (non-differentiable) : T1 — Result tensor.
Attributes	None.
Type constraints	T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(float16), tensor(float), tensor(double), tensor(bfloat16) — Constrain input types to all numeric tensors. T1 : tensor(bool) — Constrain output to boolean tensor.
Test vectors	test_greater, test_greater_int8, test_greater_int16, test_greater_uint8, test_greater_uint16, test_greater_uint32, test_greater_uint64, test_greater_bcast
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.72. GreaterOrEqual

Returns the tensor resulted from performing the `greater_equal` logical operation elementwise on the input tensors A and B (with Numpy-style broadcasting support).

This operator supports **multidirectional (i.e., Numpy-style) broadcasting**; for more details please check the doc.

Domain	ai.onnx
Since version	16
Earlier versions	12
Inputs	A (non-differentiable) : T — First input operand for the logical operator. B (non-differentiable) : T — Second input operand for the logical operator.
Outputs	C (non-differentiable) : T1 — Result tensor.
Attributes	None.
Type constraints	T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(float16), tensor(float), tensor(double), tensor(bfloat16) — Constrain input types to all numeric tensors. T1 : tensor(bool) — Constrain output to boolean tensor.
Test vectors	test_greater_equal_bcast, test_greater_equal, test_greater_equal_int8, test_greater_equal_int16, test_greater_equal_uint8, test_greater_equal_uint16, test_greater_equal_uint32, test_greater_equal_uint64
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.73. GridSample

Given an input X and a flow-field grid, computes the output Y using X values and pixel locations from the grid. For spatial input X with shape (N, C, H, W), the grid will have shape (N, H_out, W_out, 2), the output Y will have shape (N, C, H_out, W_out). For volumetric input X with shape (N, C, D, H, W), the grid will have shape (N, D_out, H_out, W_out, 3), the output Y will have shape (N, C, D_out, H_out, W_out). More generally, for an input X of rank r+2 with shape (N, C, d1, d2, ..., dr), the grid will have shape (N, D1_out, D2_out, ..., Dr_out, r), the output Y will have shape (N, C, D1_out, D2_out, ..., Dr_out).

The tensor X contains values at centers of square pixels (voxels, etc) locations such as (n, c, d1_in, d2_in, ..., dr_in). The (n, d1_out, d2_out, ..., dr_out, :) values from the tensor grid are the normalized positions for interpolating the values at the (n, c, d1_out, d2_out, ..., dr_out) locations from the output tensor Y using a specified interpolation method (the mode) and a padding mode (for grid positions falling outside the 2-dimensional image).

For example, the values in `grid[n, h_out, w_out, :]` are size-2 vectors specifying normalized positions in the 2-dimensional space of X. They are used to interpolate output values of `Y[n, c, h_out, w_out]`.

The GridSample operator is often used in doing grid generator and sampler in the [Spatial Transformer Networks](#). See also in [torch.nn.functional.grid_sample](#).

Domain	ai.onnx
Since version	22
Earlier versions	16, 20
Inputs	X (differentiable) : T1 — Input tensor of rank r+2 that has shape (N, C, D1, D2, ..., Dr), where N is the batch size, C is the number of channels, D1, D2, ..., Dr are the spatial dimensions. grid (non-differentiable) : T2 — Input offset of shape (N, D1_out, D2_out, ..., Dr_out, r), where D1_out, D2_out, ..., Dr_out are the spatial dimensions of the grid and output, and r is the number of spatial dimensions. Grid specifies the sampling locations normalized by the input spatial dimensions. Therefore, it should have most values in the range of [-1, 1]. If the grid has values outside the range of [-1, 1], the corresponding outputs will be handled as defined by padding_mode. Following computer vision convention, the coordinates in the length-r location vector are listed from the innermost tensor dimension to the outermost, the opposite of regular tensor indexing.
Outputs	Y (differentiable) : T1 — Output tensor of rank r+2 that has shape (N, C, D1_out, D2_out, ..., Dr_out) of the sampled values. For integer input types, intermediate values are computed as floating point and cast to integer at the end.
Attributes	align_corners : int (default is 0) — If align_corners=1, the extrema (-1 and 1) are considered as referring to the center points of the input's corner pixels (voxels, etc.). If align_corners=0, they are instead considered as referring to the corner points of the input's corner pixels (voxels, etc.), making the sampling more resolution agnostic. mode : string (default is linear) — Three interpolation modes: linear (default), nearest and cubic. The "linear" mode includes linear and N-linear interpolation modes depending on the number of spatial dimensions of the input tensor (i.e. linear for 1 spatial dimension, bilinear for 2 spatial dimensions, etc.). The "cubic" mode also includes N-cubic interpolation modes following the same rules. The "nearest" mode rounds to the nearest even index when the sampling point falls halfway between two indices. padding_mode : string (default is zeros) — Support padding modes for outside grid values: zeros(default), border, reflection. zeros: use 0 for out-of-bound grid locations, border: use border values for out-of-bound grid locations, reflection: use values at locations reflected by the border for out-of-bound grid locations. If index 0 represents the margin pixel, the reflected value at index -1 will be the same as the value at index 1. For location far away from the border, it will keep being reflected until becoming in bound. If pixel location $x = -3.5$ reflects by border -1 and becomes $x' = 1.5$, then reflects by border 1 and becomes $x'' = 0.5$.
Type constraints	T1 : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool),

tensor(complex64), tensor(complex128) — Constrain input X and output Y types to all tensor types.

T2 : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain grid types to float tensors.

Test vectors test_gridsample, test_gridsample_bilinear, test_gridsample_aligncorners_true, test_gridsample_nearest, test_gridsample_bicubic, test_gridsample_nearest_align_corners_0_additional_1, test_gridsample_nearest_align_corners_1_additional_1, test_gridsample_bilinear_align_corners_0_additional_1, test_gridsample_bilinear_align_corners_1_additional_1, test_gridsample_bicubic_align_corners_0_additional_1, test_gridsample_bicubic_align_corners_1_additional_1, test_gridsample_zeros_padding, test_gridsample_border_padding, test_gridsample_reflection_padding, test_gridsample_volumetric_nearest_align_corners_0, test_gridsample_volumetric_nearest_align_corners_1, test_gridsample_volumetric_bilinear_align_corners_0, test_gridsample_volumetric_bilinear_align_corners_1

Not
supplied
by the
upstream
source

Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.74. GroupNormalization

A GroupNormalization function. Carries out group normalization as described in the paper <https://arxiv.org/abs/1803.08494>

This operator transforms input according to

$$y = \text{scale} * (x - \text{mean}) / \sqrt{\text{variance} + \text{epsilon}} + \text{bias},$$

where the mean and variance are computed per instance per group of channels, and scale and bias should be specified for each channel. The number of groups num_groups should be divisible by the number of channels so that there are an equal number of channels per group.

The overall computation has two stages: the first stage normalizes the elements to have zero mean and unit variance for each instance in each group, and the second stage scales and shifts the results of the first stage. The floating-point precision used in the first stage is determined by the stash_type attribute. For example, if stash_type is 1, the operator casts all input variables to 32-bit float, performs the computation, and finally casts the normalized results back to the original type of X. The second stage does not depend on stash_type.

When the number of groups is the same as the number of channels, this operator is equivalent to InstanceNormalization. When there is only one group, this operator is equivalent to LayerNormalization.

Domain ai.onnx

Since version 21

Earlier versions 18

Inputs X (differentiable) : T — Input data tensor. Dimensions for image cases are (N x C x H x W), where N is the batch size, C is the number of channels, and H and W are the height and width of the data. Statistics are computed for every group of channels over C, H, and W. For non-

	image cases, the dimensions are in the form of (N x C x D1 x D2 ... Dn). scale (differentiable) : T — Scale tensor of shape (C). bias (differentiable) : T — Bias tensor of shape (C).
Outputs	Y (differentiable) : T — The output tensor of the same shape as X.
Attributes	epsilon : float (default is 1e-05) — The epsilon value to use to avoid division by zero. num_groups : int (required) — The number of groups of channels. It should be a divisor of the number of channels C. stash_type : int (default is 1) — The floating-point precision used in stage one of the computation.
Type constraints	T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain input and output types to float tensors.
Test vectors	test_group_normalization_epsilon, test_group_normalization_example
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.75. HammingWindow

Generates a Hamming window as described in the paper <https://ieeexplore.ieee.org/document/1455106>.

Domain	ai.onnx
Since version	17
Inputs	size (non-differentiable) : T1 — A scalar value indicating the length of the window.
Outputs	output (non-differentiable) : T2 — A Hamming window with length: size. The output has the shape: [size].
Attributes	output_datatype : int (default is 1) — The data type of the output tensor. Strictly must be one of the values from DataType enum in TensorProto whose values correspond to T2. The default value is 1 = FLOAT. periodic : int (default is 1) — If 1, returns a window to be used as periodic function. If 0, return a symmetric window. When 'periodic' is specified, hann computes a window of length size + 1 and returns the first size points. The default value is 1.
Type constraints	T1 : tensor(int32), tensor(int64) — Constrain the input size to int32_t or int64_t. T2 : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(float16), tensor(float), tensor(double), tensor(bfloat16) — Constrain output types to numeric tensors.
Test vectors	test_hammingwindow, test_hammingwindow_symmetric

Not supplied by the upstream source Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.76. HannWindow

Generates a Hann window as described in the paper <https://ieeexplore.ieee.org/document/1455106>.

Domain	ai.onnx
Since version	17
Inputs	size (non-differentiable) : T1 — A scalar value indicating the length of the window.
Outputs	output (non-differentiable) : T2 — A Hann window with length: size. The output has the shape: [size].
Attributes	output_datatype : int (default is 1) — The data type of the output tensor. Strictly must be one of the values from DataType enum in TensorProto whose values correspond to T2. The default value is 1 = FLOAT. periodic : int (default is 1) — If 1, returns a window to be used as periodic function. If 0, return a symmetric window. When 'periodic' is specified, hann computes a window of length size + 1 and returns the first size points. The default value is 1.
Type constraints	T1 : tensor(int32), tensor(int64) — Constrain the input size to int32_t or int64_t. T2 : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(float16), tensor(float), tensor(double), tensor(bfloat16) — Constrain output types to numeric tensors.
Test vectors	test_hannwindow, test_hannwindow_symmetric
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.77. HardSigmoid

HardSigmoid takes one input data (Tensor<T>) and produces one output data (Tensor<T>) where the HardSigmoid function, $y = \max(0, \min(1, \alpha * x + \beta))$, is applied to the tensor elementwise.

Domain	ai.onnx
Since version	22
Earlier versions	1, 6
Inputs	X (differentiable) : T — Input tensor
Outputs	Y (differentiable) : T — Output tensor
Attributes	alpha : float (default is 0.2) — Value of alpha.

	beta : float (default is 0.5) — Value of beta.
Type constraints	T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain input and output types to float tensors.
Test vectors	test_hardsigmoid_example, test_hardsigmoid, test_hardsigmoid_default
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.78. HardSwish

HardSwish takes one input data (Tensor<T>) and produces one output data (Tensor<T>) where the HardSwish function, $y = x * \max(0, \min(1, \alpha * x + \beta)) = x * \text{HardSigmoid}(\alpha, \beta)(x)$, where $\alpha = 1/6$ and $\beta = 0.5$, is applied to the tensor elementwise.

Domain	ai.onnx
Since version	22
Earlier versions	14
Inputs	X (differentiable) : T — Input tensor
Outputs	Y (differentiable) : T — Output tensor
Attributes	None.
Type constraints	T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain input and output types to float tensors.
Test vectors	test_hardswish
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.79. Hardmax

The operator computes the hardmax values for the given input:

$\text{Hardmax}(\text{element in input, axis}) = 1$ if the element is the first maximum value along the specified axis, 0 otherwise

The “axis” attribute indicates the dimension along which Hardmax will be performed. The output tensor has the same shape and contains the Hardmax values of the corresponding input.

Domain	ai.onnx
Since version	13
Earlier versions	1, 11
Inputs	input (differentiable) : T — The input tensor of rank $\geq$ axis.

Outputs	output (differentiable) : T — The output values with the same shape as the input tensor.
Attributes	axis : int (default is -1) — Describes the dimension Hardmax will be performed on. Negative value means counting dimensions from the back. Accepted range is [-r, r-1] where r = rank(input).
Type constraints	T : tensor(float16), tensor(float), tensor(double), tensor(bfloat16) — Constrain input and output types to float tensors.
Test vectors	test_hardmax_example, test_hardmax_one_hot, test_hardmax_axis_0, test_hardmax_axis_1, test_hardmax_axis_2, test_hardmax_negative_axis, test_hardmax_default_axis
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.80. Identity

Identity operator

Domain	ai.onnx
Since version	25
Earlier versions	1, 13, 14, 16, 19, 21, 23, 24
Inputs	input (differentiable) : V — Input tensor
Outputs	output (differentiable) : V — Tensor to copy input into.
Attributes	None.
Type constraints	V : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128), tensor(float8e4m3fn), tensor(float8e4m3fnuz), tensor(float8e5m2), tensor(float8e5m2fnuz), tensor(uint4), tensor(int4), tensor(float4e2m1), tensor(float8e8m0), tensor(uint2), tensor(int2), seq(tensor(uint8)), seq(tensor(uint16)), seq(tensor(uint32)), seq(tensor(uint64)), seq(tensor(int8)), seq(tensor(int16)), seq(tensor(int32)), seq(tensor(int64)), seq(tensor(float16)), seq(tensor(float)), seq(tensor(double)), seq(tensor(string)), seq(tensor(bool)), seq(tensor(complex64)), seq(tensor(complex128)), optional(seq(tensor(uint8))), optional(seq(tensor(uint16))), optional(seq(tensor(uint32))), optional(seq(tensor(uint64))), optional(seq(tensor(int8))), optional(seq(tensor(int16))), optional(seq(tensor(int32))), optional(seq(tensor(int64))), optional(seq(tensor(float16))), optional(seq(tensor(float))), optional(seq(tensor(double))), optional(seq(tensor(string))), optional(seq(tensor(bool))), optional(seq(tensor(complex64))), optional(seq(tensor(complex128))), optional(tensor(uint8)), optional(tensor(uint16)), optional(tensor(uint32)), optional(tensor(uint64)), optional(tensor(int8)),

optional(tensor(int16)), optional(tensor(int32)),
 optional(tensor(int64)), optional(tensor(float16)),
 optional(tensor(float)), optional(tensor(double)),
 optional(tensor(string)), optional(tensor(bool)),
 optional(tensor(complex64)), optional(tensor(complex128)) —
 Constrain input and output types to all tensor, sequence, and optional types.

Test vectors
 Not supplied by the upstream source

test_identity, test_identity_opt, test_identity_sequence

Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.81. If

If conditional

Domain	ai.onnx
Since version	25
Earlier versions	1, 11, 13, 16, 19, 21, 23, 24
Inputs	cond : B — Condition for the if. The tensor must contain a single element.
Outputs (1 — unbounded)	outputs (variadic, heterogeneous) : V — Values that are live-out to the enclosing scope. The return values in the then_branch and else_branch must be of the same data type. The then_branch and else_branch may produce tensors with the same element type and different shapes. If corresponding outputs from the then-branch and the else-branch have static shapes S1 and S2, then the shape of the corresponding output variable of the if-node (if present) must be compatible with both S1 and S2 as it represents the union of both possible shapes. For example, if in a model file, the first output of then_branch is typed float tensor with shape [2] and the first output of else_branch is another float tensor with shape [3], If's first output should have (a) no shape set, or (b) a shape of rank 1 with neither dim_value nor dim_param set, or © a shape of rank 1 with a unique dim_param. In contrast, the first output cannot have the shape [2] since [2] and [3] are not compatible.
Attributes	else_branch : graph (required) — Graph to run if condition is false. Has N outputs: values you wish to be live-out to the enclosing scope. The number of outputs must match the number of outputs in the then_branch. then_branch : graph (required) — Graph to run if condition is true. Has N outputs: values you wish to be live-out to the enclosing scope. The number of outputs must match the number of outputs in the else_branch.
Type constraints	V : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128), tensor(float8e4m3fn), tensor(float8e4m3fnuz), tensor(float8e5m2), tensor(float8e5m2fnuz), tensor(uint4), tensor(int4), tensor(float4e2m1), tensor(float8e8m0), tensor(uint2), tensor(int2), seq(tensor(uint8)), seq(tensor(uint16)), seq(tensor(uint32)), seq(tensor(uint64)),

```

seq(tensor(int8)), seq(tensor(int16)), seq(tensor(int32)),
seq(tensor(int64)), seq(tensor(bfloat16)),
seq(tensor(float16)), seq(tensor(float)),
seq(tensor(double)), seq(tensor(string)), seq(tensor(bool)),
seq(tensor(complex64)), seq(tensor(complex128)),
seq(tensor(float8e4m3fn)), seq(tensor(float8e4m3fnuz)),
seq(tensor(float8e5m2)), seq(tensor(float8e5m2fnuz)),
seq(tensor(uint4)), seq(tensor(int4)),
seq(tensor(float4e2m1)), seq(tensor(float8e8m0)),
seq(tensor(uint2)), seq(tensor(int2)),
optional(seq(tensor(uint8))), optional(seq(tensor(uint16))),
optional(seq(tensor(uint32))), optional(seq(tensor(uint64))),
optional(seq(tensor(int8))), optional(seq(tensor(int16))),
optional(seq(tensor(int32))), optional(seq(tensor(int64))),
optional(seq(tensor(bfloat16))),
optional(seq(tensor(float16))),
optional(seq(tensor(float))), optional(seq(tensor(double))),
optional(seq(tensor(string))), optional(seq(tensor(bool))),
optional(seq(tensor(complex64))),
optional(seq(tensor(complex128))), optional(tensor(uint8)),
optional(tensor(uint16)), optional(tensor(uint32)),
optional(tensor(uint64)), optional(tensor(int8)),
optional(tensor(int16)), optional(tensor(int32)),
optional(tensor(int64)), optional(tensor(bfloat16)),
optional(tensor(float16)), optional(tensor(float)),
optional(tensor(double)), optional(tensor(string)),
optional(tensor(bool)), optional(tensor(complex64)),
optional(tensor(complex128)), optional(tensor(float8e4m3fn)),
optional(tensor(float8e4m3fnuz)),
optional(tensor(float8e5m2)),
optional(tensor(float8e5m2fnuz)), optional(tensor(uint4)),
optional(tensor(int4)), optional(tensor(float4e2m1)),
optional(tensor(float8e8m0)), optional(tensor(uint2)),
optional(tensor(int2)) — All Tensor, Sequence(Tensor),
Optional(Tensor), and Optional(Sequence(Tensor)) types up to IRv13.
B : tensor(bool) — Only bool

```

Test vectors test_if, test_if_opt, test_if_seq

Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.
---	--

5.82. ImageDecoder

Loads and decodes an image from a file. If it can't decode for any reason (e.g. corrupted encoded stream, invalid format, it will return an empty matrix). The following image formats are supported:

- BMP
- JPEG (note: Lossless JPEG support is optional)
- JPEG2000
- TIFF
- PNG
- WebP
- Portable image format (PBM, PGM, PPM, PXM, PNM) Decoded images follow a channel-last layout: (Height, Width, Channels). **JPEG chroma upsampling method:** When upsampling the chroma components by a factor of 2, the pixels are linearly interpolated so that the centers of

the output pixels are 1/4 and 3/4 of the way between input pixel centers. When rounding, 0.5 is rounded down and up at alternative pixels locations to prevent bias towards larger values (ordered dither pattern). Considering adjacent input pixels A, B, and C, B is upsampled to pixels B0 and B1 so that

$$B0 = \text{round_half_down}((1/4) * A + (3/4) * B)$$

$$B1 = \text{round_half_up}((3/4) * B + (1/4) * C)$$

This method, is the default chroma upsampling method in the well-established libjpeg-turbo library, also referred as “smooth” or “fancy” upsampling.

Domain	ai.onnx
Since version	20
Inputs	encoded_stream (non-differentiable) : T1 — Encoded stream
Outputs	image (non-differentiable) : T2 — Decoded image
Attributes	pixel_format : string (default is RGB) — Pixel format. Can be one of “RGB”, “BGR”, or “Grayscale”.
Type constraints	T1 : tensor(uint8) — Constrain input types to 8-bit unsigned integer tensor. T2 : tensor(uint8) — Constrain output types to 8-bit unsigned integer tensor.
Test vectors	test_image_decoder_decode_bmp_rgb, test_image_decoder_decode_jpeg2k_rgb, test_image_decoder_decode_jpeg_bgr, test_image_decoder_decode_jpeg_grayscale, test_image_decoder_decode_jpeg_rgb, test_image_decoder_decode_png_rgb, test_image_decoder_decode_pnm_rgb, test_image_decoder_decode_tiff_rgb, test_image_decoder_decode_webp_rgb
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.83. InstanceNormalization

Carries out instance normalization as described in the paper <https://arxiv.org/abs/1607.08022>.

$y = \text{scale} * (x - \text{mean}) / \sqrt{\text{variance} + \text{epsilon}} + B$, where mean and variance are computed per instance per channel.

Domain	ai.onnx
Since version	22
Earlier versions	1, 6
Inputs	input (differentiable) : T — Input data tensor from the previous operator; dimensions for image case are (N x C x H x W), where N is the batch size, C is the number of channels, and H and W are the height and the width of the data. For non image case, the dimensions are in the form of (N x C x D1 x D2 ... Dn), where N is the batch size. scale (differentiable) : T — The input 1-dimensional scale tensor of size C. B (differentiable) : T — The input 1-dimensional bias tensor of size C.
Outputs	output (differentiable) : T — The output tensor of the same shape as input.
Attributes	epsilon : float (default is 1e-05) — The epsilon value to use to avoid division by zero.

Type constraints	T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain input and output types to float tensors.
Test vectors	test_instancenorm_example, test_instancenorm_epsilon
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.84. IsInf

Map infinity to true and other values to false.

Domain	ai.onnx
Since version	20
Earlier versions	10
Inputs	X (non-differentiable) : T1 — input
Outputs	Y (non-differentiable) : T2 — output
Attributes	detect_negative : int (default is 1) — (Optional) Whether map negative infinity to true. Default to 1 so that negative infinity induces true. Set this attribute to 0 if negative infinity should be mapped to false. detect_positive : int (default is 1) — (Optional) Whether map positive infinity to true. Default to 1 so that positive infinity induces true. Set this attribute to 0 if positive infinity should be mapped to false.
Type constraints	T1 : tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(float8e4m3fn), tensor(float8e4m3fnuz), tensor(float8e5m2), tensor(float8e5m2fnuz) — Constrain input types to float tensors. T2 : tensor(bool) — Constrain output types to boolean tensors.
Test vectors	test_isinf, test_isinf_float16, test_isinf_negative, test_isinf_positive
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.85. IsNaN

Returns which elements of the input are NaN.

Domain	ai.onnx
Since version	20
Earlier versions	9, 13
Inputs	X (non-differentiable) : T1 — input

Outputs	Y (non-differentiable) : T2—output
Attributes	None.
Type constraints	T1 : tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(float8e4m3fn), tensor(float8e4m3fnuz), tensor(float8e5m2), tensor(float8e5m2fnuz) — Constrain input types to float tensors. T2 : tensor(bool) — Constrain output types to boolean tensors.
Test vectors	test_isnan_float16, test_isnan
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.86. LRN

Local Response Normalization proposed in the [AlexNet paper](#). It normalizes over local input regions. The local region is defined across the channels. For an element $X[n, c, d1, \dots, dk]$ in a tensor of shape $(N \times C \times D1 \times D2, \dots, Dk)$, its region is $\{X[n, i, d1, \dots, dk] \mid \max(0, c - \text{floor}((\text{size} - 1) / 2)) \leq i \leq \min(C - 1, c + \text{ceil}((\text{size} - 1) / 2))\}$.

$\text{square_sum}[n, c, d1, \dots, dk] = \text{sum}(X[n, i, d1, \dots, dk] ^ 2)$, where $\max(0, c - \text{floor}((\text{size} - 1) / 2)) \leq i \leq \min(C - 1, c + \text{ceil}((\text{size} - 1) / 2))$.

$Y[n, c, d1, \dots, dk] = X[n, c, d1, \dots, dk] / (\text{bias} + \text{alpha} / \text{size} * \text{square_sum}[n, c, d1, \dots, dk]) ^ \text{beta}$

Domain	ai.onnx
Since version	13
Earlier versions	1
Inputs	X (differentiable) : T—Input data tensor from the previous operator; dimensions for image case are $(N \times C \times H \times W)$, where N is the batch size, C is the number of channels, and H and W are the height and the width of the data. For non image case, the dimensions are in the form of $(N \times C \times D1 \times D2 \dots Dn)$, where N is the batch size. Optionally, if dimension denotation is in effect, the operation expects the input data tensor to arrive with the dimension denotation of [DATA_BATCH, DATA_CHANNEL, DATA_FEATURE, DATA_FEATURE ...].
Outputs	Y (differentiable) : T—Output tensor, which has the shape and type as input tensor
Attributes	alpha : float (default is 0.0001) — Scaling parameter. beta : float (default is 0.75) — The exponent. bias : float (default is 1.0) size : int (required) — The number of channels to sum over
Type constraints	T : tensor(float16), tensor(float), tensor(double), tensor(bfloat16) — Constrain input and output types to float tensors.
Test vectors	test_lrn_default, test_lrn

Not supplied by the upstream source Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.87. LSTM

Computes an one-layer LSTM. This operator is usually supported via some custom implementation such as CuDNN.

Notations:

- X — input tensor
- i — input gate
- o — output gate
- f — forget gate
- c — cell gate
- t — time step ($t-1$ means previous time step)
- $W[iofc]$ — W parameter weight matrix for input, output, forget, and cell gates
- $R[iofc]$ — R recurrence weight matrix for input, output, forget, and cell gates
- $Wb[iofc]$ — W bias vectors for input, output, forget, and cell gates
- $Rb[iofc]$ — R bias vectors for input, output, forget, and cell gates
- $P[iof]$ — P peephole weight vector for input, output, and forget gates
- $WB[iofc]$ — W parameter weight matrix for backward input, output, forget, and cell gates
- $RB[iofc]$ — R recurrence weight matrix for backward input, output, forget, and cell gates
- $WBb[iofc]$ — W bias vectors for backward input, output, forget, and cell gates
- $RBb[iofc]$ — R bias vectors for backward input, output, forget, and cell gates
- $PB[iof]$ — P peephole weight vector for backward input, output, and forget gates
- H — Hidden state
- $num_directions$ — 2 if direction == bidirectional else 1

Activation functions:

- $Relu(x)$ — $\max(0, x)$
- $Tanh(x)$ — $(1 - e^{-2x}) / (1 + e^{-2x})$
- $Sigmoid(x)$ — $1 / (1 + e^{-x})$

NOTE Below are optional

- $Affine(x)$ — $\alpha * x + \beta$
- $LeakyRelu(x)$ — x if $x \geq 0$ else $\alpha * x$
- $ThresholdedRelu(x)$ — x if $x \geq \alpha$ else 0
- $ScaledTanh(x)$ — $\alpha Tanh(\beta x)$
- $HardSigmoid(x)$ — $\min(\max(\alpha * x + \beta, 0), 1)$
- $Elu(x)$ — x if $x \geq 0$ else $\alpha * (e^x - 1)$

Table 6 — Table from the upstream description of LSTM

- $Softsign(x)$ — $x / (1 + |x|)$
- $Softplus(x)$ — $\log(1 + e^x)$

Equations (Default: $f=Sigmoid$, $g=Tanh$, $h=Tanh$):

- $it = f(X_t(W_i^T) + H_{t-1}(R_i^T) + P_i(.) C_{t-1} + W_{bi} + R_{bi})$
- $ft = f(X_t(W_f^T) + H_{t-1}(R_f^T) + P_f(.) C_{t-1} + W_{bf} + R_{bf})$
- $ct = g(X_t(W_c^T) + H_{t-1}(R_c^T) + W_{bc} + R_{bc})$
- $C_t = ft(.) C_{t-1} + it(.) ct$
- $ot = f(X_t(W_o^T) + H_{t-1}(R_o^T) + P_o(.) C_t + W_{bo} + R_{bo})$

- $H_t = \text{ot}(\cdot) h(C_t)$ This operator has **optional** inputs/outputs. See the doc for more details about the representation of optional arguments. An empty string may be used in the place of an actual argument's name to indicate a missing argument. Trailing optional arguments (those not followed by an argument that is present) may also be simply omitted.

Domain `ai.onnx`

Since version `22`

Earlier `1, 7, 14`

versions

Inputs (3 — 8) `X (differentiable) : T` — The input sequences packed (and potentially padded) into one 3-D tensor with the shape of `[seq_length, batch_size, input_size]`.

`W (differentiable) : T` — The weight tensor for the gates.

Concatenation of `W[iofc]` and `WB[iofc]` (if bidirectional) along dimension 0. The tensor has shape `[num_directions, 4*hidden_size, input_size]`.

`R (differentiable) : T` — The recurrence weight tensor.

Concatenation of `R[iofc]` and `RB[iofc]` (if bidirectional) along dimension 0. This tensor has shape `[num_directions, 4*hidden_size, hidden_size]`.

`B (optional, differentiable) : T` — The bias tensor for input gate. Concatenation of `[Wb[iofc], Rb[iofc], and [WBb[iofc], RBb[iofc]]` (if bidirectional) along dimension 0. This tensor has shape `[num_directions, 8*hidden_size]`. Optional: If not specified — assumed to be 0.

`sequence_lens (optional, non-differentiable) : T1` — Optional tensor specifying lengths of the sequences in a batch. If not specified — assumed all sequences in the batch to have length `seq_length`. It has shape `[batch_size]`.

`initial_h (optional, non-differentiable) : T` — Optional initial value of the hidden. If not specified — assumed to be 0. It has shape `[num_directions, batch_size, hidden_size]`.

`initial_c (optional, non-differentiable) : T` — Optional initial value of the cell. If not specified — assumed to be 0. It has shape `[num_directions, batch_size, hidden_size]`.

`P (optional, differentiable) : T` — The weight tensor for peepholes. Concatenation of `P[iof]` and `PB[iof]` (if bidirectional) along dimension 0. It has shape `[num_directions, 3*hidden_size]`. Optional: If not specified — assumed to be 0.

Outputs (0 — 3) `Y (optional, differentiable) : T` — A tensor that concatenates all the intermediate output values of the hidden. It has shape `[seq_length, num_directions, batch_size, hidden_size]`.

`Y_h (optional, differentiable) : T` — The last output value of the hidden. It has shape `[num_directions, batch_size, hidden_size]`.

`Y_c (optional, differentiable) : T` — The last output value of the cell. It has shape `[num_directions, batch_size, hidden_size]`.

Attributes `activation_alpha` : list of floats — Optional scaling values used by some activation functions. The values are consumed in the order of activation functions, for example (f, g, h) in LSTM. Default values are the same as of corresponding ONNX operators. For example with LeakyRelu, the default alpha is 0.01.

`activation_beta` : list of floats — Optional scaling values used by some activation functions. The values are consumed in the order of activation functions, for example (f, g, h) in LSTM. Default values are the same as of corresponding ONNX operators.

`activations` : list of strings — A list of 3 (or 6 if bidirectional) activation functions for input, output, forget, cell, and hidden. The activation functions must be one of the activation functions specified above. Optional: See the equations for default if not specified.

	<code>clip</code> : float — Cell clip threshold. Clipping bounds the elements of a tensor in the range of <code>[-threshold, +threshold]</code> and is applied to the input of activations. No clip if not specified. <code>direction</code> : string (default is <code>forward</code>) — Specify if the RNN is forward, reverse, or bidirectional. Must be one of <code>forward</code> (default), <code>reverse</code> , or <code>bidirectional</code> . <code>hidden_size</code> : int — Number of neurons in the hidden layer <code>input_forget</code> : int (default is 0) — Couple the input and forget gates if 1. <code>layout</code> : int (default is 0) — The shape format of inputs <code>X</code> , initial <code>h</code> , initial <code>c</code> and outputs <code>Y</code> , <code>Y_h</code> , <code>Y_c</code> . If 0, the following shapes are expected: <code>X.shape = [seq_length, batch_size, input_size]</code> , <code>Y.shape = [seq_length, num_directions, batch_size, hidden_size]</code> , <code>initial_h.shape = Y_h.shape = initial_c.shape = Y_c.shape = [num_directions, batch_size, hidden_size]</code> . If 1, the following shapes are expected: <code>X.shape = [batch_size, seq_length, input_size]</code> , <code>Y.shape = [batch_size, seq_length, num_directions, hidden_size]</code> , <code>initial_h.shape = Y_h.shape = initial_c.shape = Y_c.shape = [batch_size, num_directions, hidden_size]</code> .
Type	<code>T</code> : tensor(bfloat16), tensor(float16), tensor(float),
constraints	tensor(double) — Constrain input and output types to float tensors. <code>T1</code> : tensor(int32) — Constrain <code>seq_lens</code> to integer tensor.
Test vectors	<code>test_lstm_batchwise</code> , <code>test_lstm_bidirectional</code> , <code>test_lstm_defaults</code> , <code>test_lstm_with_initial_bias</code> , <code>test_lstm_with_peepholes</code> , <code>test_lstm_reverse</code>
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.88. LayerNormalization

This is layer normalization defined in ONNX as function. The overall computation can be split into two stages. The first stage is standardization, which makes the normalized elements have zero mean and unit variances. The computation required by standardization can be described by the following equations. $\text{Mean} = \text{ReduceMean}(\text{axes}=\text{normalized_axes})(X)$ $D = \text{Sub}(X, \text{Mean})$ $DD = \text{Mul}(D, D)$ $\text{Var} = \text{ReduceMean}(\text{axes}=\text{normalized_axes})(DD)$ $\text{VarEps} = \text{Add}(\text{Var}, \text{epsilon})$ $\text{StdDev} = \text{Sqrt}(\text{VarEps})$ $\text{InvStdDev} = \text{Reciprocal}(\text{StdDev})$ $\text{Normalized} = \text{Mul}(D, \text{InvStdDev})$ where `normalized_axes` is `[axis, ..., rank of X - 1]`. The variables `Var` and `StdDev` stand for variance and standard deviation, respectively. The second output is `Mean` and the last one is `InvStdDev`. Depending on `stash_type` attribute, the actual computation must happen in different floating-point precision. For example, if `stash_type` is 1, this operator casts all input variables to 32-bit float, perform the computation, and finally cast `Normalized` back to the original type of `X`. The second stage then scales and shifts the outcome of the first stage using $\text{NormalizedScaled} = \text{Mul}(\text{Normalized}, \text{Scale})$ $Y = \text{Add}(\text{NormalizedScaled}, B)$ The second stage doesn't depends on `stash_type`. All equations are in [this syntax](#). The same variable (i.e., input, output, and attribute) uses the same name in the equations above and this operator's definition. Let `d[i]` indicate the *i*-th dimension of `X`. If `X`'s shape is `[d[0], ..., d[axis-1], d[axis], ..., d[rank-1]]`, the shape of `Mean` and `InvStdDev` is `[d[0], ..., d[axis-1], 1, ..., 1]`. `Y` and `X` have the same shape. This operator supports unidirectional broadcasting (tensors `Scale` and `B` should be unidirectional broadcastable to tensor `X`); for more details please check the doc.

Domain	<code>ai.onnx</code>
Since version	17
Inputs (2 — 3)	<code>X</code> : T — Tensor to be normalized. <code>Scale</code> : T — Scale tensor. <code>B (optional)</code> : T — Bias tensor.

Outputs (1 — 3)	$Y : T$ — Normalized tensor. Mean (optional) : U — Saved mean used during training to speed up gradient computation InvStdDev (optional) : U — Saved inverse standard deviation used during training to speed up gradient computation.
Attributes	axis : int (default is -1) — The first normalization dimension. If rank(X) is r, axis' allowed range is $[-r, r)$. Negative value means counting dimensions from the back. epsilon : float (default is $1e-05$) — The epsilon value to use to avoid division by zero. stash_type : int (default is 1) — Type of Mean and InvStdDev. This also specifies stage one's computation precision.
Type constraints	$T : \text{tensor}(\text{float16}), \text{tensor}(\text{float}), \text{tensor}(\text{double}), \text{tensor}(\text{bfloat16})$ — Constrain input types and output Y type to float tensors. $U : \text{tensor}(\text{float}), \text{tensor}(\text{bfloat16})$ — Type of Mean and InvStdDev tensors.
Test vectors	test_layer_normalization_default_axis
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.89. LeakyRelu

LeakyRelu takes input data ($\text{Tensor}<T>$) and an argument alpha, and produces one output data ($\text{Tensor}<T>$) where the function $f(x) = \alpha * x$ for $x < 0$, $f(x) = x$ for $x \geq 0$, is applied to the data tensor elementwise.

Domain	ai.onnx
Since version	16
Earlier versions	1, 6
Inputs	X (differentiable) : T — Input tensor
Outputs	Y (differentiable) : T — Output tensor
Attributes	alpha : float (default is 0.01) — Coefficient of leakage.
Type constraints	$T : \text{tensor}(\text{bfloat16}), \text{tensor}(\text{float16}), \text{tensor}(\text{float}), \text{tensor}(\text{double})$ — Constrain input and output types to float tensors.
Test vectors	test_leakyrelu_example, test_leakyrelu, test_leakyrelu_default
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.90. Less

Returns the tensor resulted from performing the less logical operation elementwise on the input tensors A and B (with Numpy-style broadcasting support).

This operator supports **multidirectional (i.e., Numpy-style) broadcasting**; for more details please check the doc.

Domain	ai.onnx
Since version	13
Earlier versions	1, 7, 9
Inputs	A (non-differentiable) : T — First input operand for the logical operator. B (non-differentiable) : T — Second input operand for the logical operator.
Outputs	C (non-differentiable) : T1 — Result tensor.
Attributes	None.
Type constraints	T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(float16), tensor(float), tensor(double), tensor(bfloat16) — Constrain input types to all numeric tensors. T1 : tensor(bool) — Constrain output to boolean tensor.
Test vectors	test_less, test_less_int8, test_less_int16, test_less_uint8, test_less_uint16, test_less_uint32, test_less_uint64, test_less_bcast
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.91. LessOrEqual

Returns the tensor resulted from performing the less_equal logical operation elementwise on the input tensors A and B (with Numpy-style broadcasting support).

This operator supports **multidirectional (i.e., Numpy-style) broadcasting**; for more details please check the doc.

Domain	ai.onnx
Since version	16
Earlier versions	12
Inputs	A (non-differentiable) : T — First input operand for the logical operator. B (non-differentiable) : T — Second input operand for the logical operator.

Outputs	C (non-differentiable) : T1 — Result tensor.
Attributes	None.
Type constraints	T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(float16), tensor(float), tensor(double), tensor(bfloat16) — Constrain input types to all numeric tensors. T1 : tensor(bool) — Constrain output to boolean tensor.
Test vectors	test_less_equal_bcast, test_less_equal, test_less_equal_int8, test_less_equal_int16, test_less_equal_uint8, test_less_equal_uint16, test_less_equal_uint32, test_less_equal_uint64
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.92. LinearAttention

Unified linear attention operator for autoregressive decoding ($T=1$) and prefill ($T>1$).

The query, key, value, and (where applicable) decay/beta inputs use 3D packed format $[B, T, H*D]$, where heads are flattened into the last dimension; `q_num_heads` and `kv_num_heads` are always required and are used to unpack to 4D internally for computation. The optional `past_state` and `present_state` are 4D with shape (B, H_{kv}, d_k, d_v) .

Group-query attention (GQA) is supported: `q_num_heads` must be a positive multiple of `kv_num_heads`. When `q_num_heads == kv_num_heads` this reduces to multi-headed linear attention; when `q_num_heads > kv_num_heads` each KV head (and its recurrent state) is shared by `q_num_heads / kv_num_heads` query heads (multi-query attention is the special case `kv_num_heads == 1`).

The `update_rule` attribute selects the recurrence type:

- "linear": $S_t = S_{t-1} + k_t \otimes v_t$; $o_t = \text{scale} * q_t^T S_t$
- "gated": $S_t = \exp(g_t) * S_{t-1} + k_t \otimes v_t$; $o_t = \text{scale} * q_t^T S_t$
- "delta": $S_t = S_{t-1} + \beta_t * k_t \otimes (v_t - S_{t-1}^T k_t)$; $o_t = \text{scale} * q_t^T S_t$
- "gated_delta": $S_t = \exp(g_t) * S_{t-1} + \beta_t * k_t \otimes (v_t - \exp(g_t) * S_{t-1}^T k_t)$; $o_t = \text{scale} * q_t^T S_t$

where g_t is the decay (in log-space), β_t is the update rate, and $\otimes$ denotes outer product.

Semantics: Equivalent to running the recurrent update sequentially for each token, but may be implemented using chunk-parallel algorithms for GPU efficiency.

Domain `ai.onnx`

Since version 27

Inputs (3 — 6)

- `query` (differentiable) : T — Query vectors with 3D packed shape $(B, T, H_q * d_k)$. Heads are packed into the last dimension.
- `key` (differentiable) : T — Key vectors with 3D packed shape $(B, T, H_{kv} * d_k)$. Should be L2-normalized for delta/gated_delta modes.
- `value` (differentiable) : T — Value vectors with 3D packed shape $(B, T, H_{kv} * d_v)$.

	<code>past_state</code> (optional, non-differentiable) : S — Recurrent state from previous step with shape (B, H_kv, d_k, d_v). Always 4D. If not provided, defaults to zeros. <code>decay</code> (optional, differentiable) : T — Exponential decay gate in log-space. 3D packed shape: (B, T, H_kv * d_k) for per-key-dimension decay (GLA/RWKV-6), or (B, T, H_kv) for per-head scalar decay (DeltaNet/RetNet). Required for 'gated' and 'gated_delta' modes. <code>beta</code> (optional, differentiable) : T — Update rate (sigmoid output). 3D packed shape: (B, T, H_kv) or (B, T, 1). Required for 'delta' and 'gated_delta' modes.
Outputs	<code>output</code> (differentiable) : T — Attention output with 3D packed shape (B, T, H_q * d_v). <code>present_state</code> (non-differentiable) : S — Updated recurrent state with shape (B, H_kv, d_k, d_v). Always 4D.
Attributes	<code>chunk_size</code> : int (default is 64) — Chunk size for the chunk-parallel WY decomposition during prefill (T>1). Tuning hint; does not affect output correctness. <code>kv_num_heads</code> : int (required) — Number of key/value heads. Always required. <code>q_num_heads</code> : int (required) — Number of query heads. Always required. <code>scale</code> : float (default is 0.0) — Output scaling factor. When 0.0 (default), derives $d_k = \text{query.shape}[-1] / q_num_heads$ and uses $1/\sqrt{d_k}$. Set explicitly to override. <code>update_rule</code> : string (default is gated_delta) — The update rule for the linear attention recurrence. One of: 'linear', 'gated', 'delta', 'gated_delta'. Default is 'gated_delta'.
Type constraints	T : tensor(float16), tensor(bfloat16), tensor(float) — Constrain activation input and output types to float16, bfloat16, or float32 tensors. S : tensor(float16), tensor(bfloat16), tensor(float) — Constrain state types to float16, bfloat16, or float32 tensors. Should be float32 or the same as T for numerical stability on long sequences.
Test vectors	<code>test_linear_attention_decode_step</code>, <code>test_linear_attention_delta</code>, <code>test_linear_attention_explicit_scale</code>, <code>test_linear_attention_fp16</code>, <code>test_linear_attention_gated</code>, <code>test_linear_attention_gated_delta</code>, <code>test_linear_attention_gated_delta_beta_scalar</code>, <code>test_linear_attention_gated_delta_gqa</code>, <code>test_linear_attention_gated_delta_mqa</code>, <code>test_linear_attention_gated_per_head_decay</code>, <code>test_linear_attention_linear</code>, <code>test_linear_attention_linear_t1_no_past</code>, <code>test_linear_attention_no_past_explicit_zeros</code>, <code>test_linear_attention_prefill_with_past</code>
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.93. Log

Calculates the natural log of the given input tensor, element-wise.

Domain `ai.onnx`

ONNX 1-2 (unofficial draft)

Since version	13
Earlier versions	1, 6
Inputs	input (differentiable) : T—Input tensor
Outputs	output (differentiable) : T—The natural log of the input tensor computed element-wise
Attributes	None.
Type constraints	T : tensor(float16), tensor(float), tensor(double), tensor(bfloat16) — Constrain input and output types to float tensors.
Test vectors	test_log_example, test_log
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.94. LogSoftmax

The operator computes the log of softmax values for the given input:

$\text{LogSoftmax}(\text{input}, \text{axis}) = \text{Log}(\text{Softmax}(\text{input}, \text{axis}=\text{axis}))$

The “axis” attribute indicates the dimension along which LogSoftmax will be performed. The output tensor has the same shape and contains the LogSoftmax values of the corresponding input.

Domain	ai.onnx
Since version	13
Earlier versions	1, 11
Inputs	input (differentiable) : T—The input tensor of rank $\geq$ axis.
Outputs	output (differentiable) : T—The output values with the same shape as the input tensor.
Attributes	axis : int (default is -1) — Describes the dimension LogSoftmax will be performed on. Negative value means counting dimensions from the back. Accepted range is $[-r, r-1]$ where $r = \text{rank}(\text{input})$.
Type constraints	T : tensor(float16), tensor(float), tensor(double), tensor(bfloat16) — Constrain input and output types to float tensors.
Test vectors	test_logsoftmax_example_1, test_logsoftmax_large_number, test_logsoftmax_axis_0, test_logsoftmax_axis_1, test_logsoftmax_axis_2, test_logsoftmax_negative_axis, test_logsoftmax_default_axis
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.95. Loop

Generic Looping construct. This loop has multiple termination conditions:

1) Trip count. Iteration count specified at runtime. Set by specifying the input M. Optional. Set to empty string to omit. Note that a static trip count (specified at graph construction time) can be specified by passing in a constant node for input M. 2) Loop termination condition. This is an input to the op that determines whether to run the first iteration and also a loop-carried dependency for the body graph. The body graph must yield a value for the condition variable, whether this input is provided or not.

This table summarizes the operating modes of this operator with equivalent C-style code:

Operator inputs defined as (max_trip_count, condition_var).

- input ("", ""): for (int i=0; ; ++i) { cond = ... // Note this value is ignored, but is required in the body }
- input ("", cond) // Note this is analogous to a while loop bool cond = ...; for (int i=0; cond; ++i) { cond = ...; }
- input ("", 1) // Note this is analogous to a do-while loop bool cond = true for (int i=0; cond; ++i) { cond = ...; }
- input (trip_count, "") // Note this is analogous to a for loop int trip_count = ... for (int i=0; i < trip_count; ++i) { cond = ...; // ignored }
- input (trip_count, cond) int trip_count = ...; bool cond = ...; for (int i=0; i < trip_count && cond; ++i) { cond = ...; }

Sample usage — cond as well as trip count

```
graph predict-net {
  %a = Constant[value = <Scalar Tensor [3]>]()
  %b = Constant[value = <Scalar Tensor [6]>]()
  %keepgoing = Constant[value = <Scalar Tensor [1]>]()
  %max_trip_count = Constant[value = <Scalar Tensor [10]>]()
  %keepgoing_out, %b_out, %user_defined_vals = Loopbody = <graph body-net>
  return
}

graph body-net (
  %i[INT32, scalar]           // iteration number
  %keepgoing_in[B00L, scalar] // incoming loop-termination-condition; not used
  %b_in[INT32, scalar]        // incoming value of loop-carried-dependency b
) {
  %my_local = Add(%a, %b_in)
  %b_out = Sub(%a, %b_in) // outgoing value of loop-carried-dependency b
  %keepgoing_out = Greater(%my_local, %b_out) // outgoing loop-termination-
condition
  %user_defined_val = Add(%b_in, %b_in) // scan-output value to be accumulated
  return %keepgoing_out, %b_out, %user_defined_val
}
```

Sample equivalent C code

```
{
  /* User-defined code (enclosing scope) */
  int a = 3, b = 6;
  bool keepgoing = true; // Analogous to input cond
  /* End user-defined code */

  /* Implicitly-defined code */
```

ONNX 1-2 (unofficial draft)

```
const int max_trip_count = 10; // Analogous to input M
int user_defined_vals[]; // Imagine this is resizable
/* End implicitly-defined code */
/* initialize loop-carried variables and scan-output variables */
bool keepgoing_out = keepgoing
int b_out = b

for (int i=0; i < max_trip_count && keepgoing_out; ++i) {
    /* Implicitly-defined code: bind actual parameter values
       to formal parameter variables of loop-body */
    bool keepgoing_in = keepgoing_out;
    bool b_in = b_out;

    /* User-defined code (loop body) */
    int my_local = a + b_in; // Reading value "a" from the enclosing scope is
fine
    b_out = a - b_in;
    keepgoing_out = my_local > b_out;
    user_defined_val = b_in + b_in; // b_in and b_out are different variables
    /* End user-defined code */

    /* Implicitly defined-code */
    user_defined_vals[i] = user_defined_val // accumulate scan-output values
}
// int t = my_local; // Can't do this. my_local is not accessible here.

// The values below are bound to the output variables of the loop and
therefore accessible
// b_out; user_defined_vals; keepgoing_out;
}
```

There are several things of note in this code snippet:

1) Values from the enclosing scope (i.e. variable “a” here) are in scope and can be referenced in the inputs of the loop. 2) Any values computed in the loop body that needs to be used in a subsequent iteration or after the loop are modeled using a pair of variables in the loop-body, consisting of an input variable (eg., b_in) and an output variable (eg., b_out). These are referred to as loop-carried dependences. The loop operation node supplies the input value of the input variable for the first iteration, and returns the output value of the output variable produced by the final iteration. 3) Scan_output variables are used to implicitly concatenate values computed across all the iterations. In the above example, the value of user_defined_val computed over all iterations are concatenated and returned as the value of user_defined_vals after the loop. 4) Values created in the body cannot be accessed in the enclosing scope, except using the mechanism described above.

Note that the semantics of this op support “diagonal” or “wavefront” execution. (See Step 3 here for an example: <https://devblogs.nvidia.com/optimizing-recurrent-neural-networks-cudnn-5/>). Frontends should emit multi-layer RNNs as a series of While operators (with time being the inner looping dimension), with each successive layer consuming the scan_outputs from the previous layer, possibly going through several point-wise operators (e.g. dropout, residual connections, linear layer).

The input/output of subgraph (produced by loop node) matching is based on order instead of name. The implementation will figure out the names based on this order.

Domain	ai.onnx
Since version	25
Earlier versions	1, 11, 13, 16, 19, 21, 23, 24

Inputs (2 — unbounded)	M (optional) : I — A maximum trip-count for the loop specified at runtime. Optional. Pass empty string to skip. cond (optional) : B — A boolean termination condition. Optional. Pass empty string to skip. v_initial (variadic, heterogeneous) : V — The initial values of any loop-carried dependencies (values that change across loop iterations)
Outputs (1 — unbounded)	v_final_and_scan_outputs (variadic, heterogeneous) : V — Final N loop carried dependency values then K scan_outputs. Scan outputs must be Tensors.
Attributes	body : graph (required) — The graph run each iteration. It has 2+N inputs: (iteration_num, condition, loop carried dependencies...). It has 1+N+K outputs: (condition, loop carried dependencies..., scan_outputs...). Each scan_output is created by concatenating the value of the specified output value at the end of each iteration of the loop. It is an error if the dimensions or data type of these scan_outputs change across loop iterations.
Type constraints	V : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128), tensor(float8e4m3fn), tensor(float8e4m3fnuz), tensor(float8e5m2), tensor(float8e5m2fnuz), tensor(uint4), tensor(int4), tensor(float4e2m1), tensor(float8e8m0), tensor(uint2), tensor(int2), seq(tensor(uint8)), seq(tensor(uint16)), seq(tensor(uint32)), seq(tensor(uint64)), seq(tensor(int8)), seq(tensor(int16)), seq(tensor(int32)), seq(tensor(int64)), seq(tensor(bfloat16)), seq(tensor(float16)), seq(tensor(float)), seq(tensor(double)), seq(tensor(string)), seq(tensor(bool)), seq(tensor(complex64)), seq(tensor(complex128)), seq(tensor(float8e4m3fn)), seq(tensor(float8e4m3fnuz)), seq(tensor(float8e5m2)), seq(tensor(float8e5m2fnuz)), seq(tensor(uint4)), seq(tensor(int4)), seq(tensor(float4e2m1)), seq(tensor(float8e8m0)), seq(tensor(uint2)), seq(tensor(int2)), optional(seq(tensor(uint8))), optional(seq(tensor(uint16))), optional(seq(tensor(uint32))), optional(seq(tensor(uint64))), optional(seq(tensor(int8))), optional(seq(tensor(int16))), optional(seq(tensor(int32))), optional(seq(tensor(int64))), optional(seq(tensor(bfloat16))), optional(seq(tensor(float16))), optional(seq(tensor(float))), optional(seq(tensor(double))), optional(seq(tensor(string))), optional(seq(tensor(bool))), optional(seq(tensor(complex64))), optional(seq(tensor(complex128))), optional(tensor(uint8)), optional(tensor(uint16)), optional(tensor(uint32)), optional(tensor(uint64)), optional(tensor(int8)), optional(tensor(int16)), optional(tensor(int32)), optional(tensor(int64)), optional(tensor(bfloat16)), optional(tensor(float16)), optional(tensor(float)), optional(tensor(double)), optional(tensor(string)), optional(tensor(bool)), optional(tensor(complex64)), optional(tensor(complex128)), optional(tensor(float8e4m3fn)), optional(tensor(float8e4m3fnuz)), optional(tensor(float8e5m2)), optional(tensor(float8e5m2fnuz)), optional(tensor(uint4)),

optional(tensor(int4)), optional(tensor(float4e2m1)),
 optional(tensor(float8e8m0)), optional(tensor(uint2)),
 optional(tensor(int2)) — All Tensor, Sequence(Tensor),
 Optional(Tensor), and Optional(Sequence(Tensor)) types up to IRv13.
 I : tensor(int64) — tensor of int64, which should be a scalar.
 B : tensor(bool) — tensor of bool, which should be a scalar.

Test vectors test_loop11, test_loop13_seq, test_loop16_seq_none

Not supplied by the upstream source Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.96. LpNormalization

Given a matrix, apply Lp-normalization along the provided axis. The output is computed as: $\text{output} = \text{input} / \text{Lp_norm}(\text{input}, \text{axis})$. When the Lp norm is zero (i.e., all elements along the axis are zero), the output is defined to be zero to avoid division by zero.

Domain	ai.onnx
Since version	22
Earlier versions	1
Inputs	input (differentiable) : T — Input matrix
Outputs	output (differentiable) : T — Matrix after normalization
Attributes	axis : int (default is -1) — The axis on which to apply normalization, -1 mean last axis. p : int (default is 2) — The order of the normalization, only 1 or 2 are supported.
Type constraints	T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain input and output types to float tensors.
Test vectors	test_lpnormalization_default, test_l1normalization_axis_0, test_l1normalization_axis_1, test_l1normalization_axis_last, test_l2normalization_axis_0, test_l2normalization_axis_1
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.97. LpPool

LpPool consumes an input tensor X and applies Lp pooling across the tensor according to kernel sizes, stride sizes, and pad lengths. Lp pooling consisting of computing the Lp norm on all values of a subset of the input tensor according to the kernel size and downsampling the data into the output tensor Y for further processing. The output spatial shape will be following:

```
output_spatial_shape[i] = floor((input_spatial_shape[i] + pad_shape[i] -
{kernelSpatialShape}) / strides_spatial_shape[i] + 1)
```

or

```
output_spatial_shape[i] = ceil((input_spatial_shape[i] + pad_shape[i] -
{kernelSpatialShape}) / strides_spatial_shape[i] + 1)
```

if `ceil_mode` is enabled `pad_shape[i]` is the sum of pads along axis `i`.

`auto_pad` is a DEPRECATED attribute. If you are using them currently, the output spatial shape will be following:

```
VALID: output_spatial_shape[i] = ceil((input_spatial_shape[i] -
{kernelSpatialShape} + 1) / strides_spatial_shape[i])
SAME_UPPER or SAME_LOWER: output_spatial_shape[i] = ceil(input_spatial_shape[i]
/ strides_spatial_shape[i])
```

And `pad_shape` will be following if `SAME_UPPER` or `SAME_LOWER`:

```
pad_shape[i] = (output_spatial_shape[i] - 1) * strides_spatial_shape[i] +
{kernelSpatialShape} - input_spatial_shape[i]
```

Domain ai.onnx

Since version 22

Earlier versions 1, 2, 11, 18

Inputs `X (differentiable) : T` — Input data tensor from the previous operator; dimensions for image case are $(N \times C \times H \times W)$, where N is the batch size, C is the number of channels, and H and W are the height and the width of the data. For non image case, the dimensions are in the form of $(N \times C \times D1 \times D2 \dots Dn)$, where N is the batch size.

Outputs `Y (differentiable) : T` — Output data tensor from Lp pooling across the input tensor. Dimensions will vary based on various kernel, stride, and pad sizes.

Attributes `auto_pad : string` (default is NOTSET) — `auto_pad` must be either NOTSET, SAME_UPPER, SAME_LOWER or VALID. Where default value is NOTSET, which means explicit padding is used. SAME_UPPER or SAME_LOWER mean pad the input so that `output_shape[i] = ceil(input_shape[i] / strides[i])` for each axis `i`. The padding is split between the two sides equally or almost equally (depending on whether it is even or odd). In case the padding is an odd number, the extra padding is added at the end for SAME_UPPER and at the beginning for SAME_LOWER.
 `ceil_mode : int` (default is 0) — Whether to use ceil or floor (default) to compute the output shape.
 `dilations : list of ints` — dilation value along each spatial axis of the filter. If not present, the dilation defaults is 1 along each spatial axis.
 `kernel_shape : list of ints` (required) — The size of the kernel along each axis.
 `p : int` (default is 2) — p value of the Lp norm used to pool over the input data.
 `pads : list of ints` — Padding for the beginning and ending along each spatial axis, it can take any value greater than or equal to 0. The value represent the number of pixels added to the beginning and end part of the corresponding axis. pads format should be as follow `[x1_begin, x2_begin...x1_end, x2_end,...]`, where `xi_begin` the number of pixels added at the beginning of axis `i` and `xi_end`, the number of pixels added at the end of axis `i`. This attribute cannot be used simultaneously with `auto_pad`

	attribute. If not present, the padding defaults to 0 along start and end of each spatial axis. strides : list of ints — Stride along each spatial axis. If not present, the stride defaults to 1 along each spatial axis.
Type constraints	T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain input and output types to float tensors.
Test vectors	test_lppool_1d_default, test_lppool_2d_default, test_lppool_2d_dilations, test_lppool_2d_pads, test_lppool_2d_same_lower, test_lppool_2d_same_upper, test_lppool_2d_strides, test_lppool_3d_default
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.98. MatMul

Matrix product that behaves like [numpy.matmul](#).

Domain	ai.onnx
Since version	13
Earlier versions	1, 9
Inputs	A (differentiable) : T — N-dimensional matrix A B (differentiable) : T — N-dimensional matrix B
Outputs	Y (differentiable) : T — Matrix multiply results from A * B
Attributes	None.
Type constraints	T : tensor(float16), tensor(float), tensor(double), tensor(uint32), tensor(uint64), tensor(int32), tensor(int64), tensor(bfloat16) — Constrain input and output types to float/int tensors.
Test vectors	test_matmul_2d, test_matmul_3d, test_matmul_4d, test_matmul_bcast, test_matmul_1d_3d, test_matmul_4d_1d, test_matmul_1d_1d
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.99. MatMulInteger

Matrix product that behaves like [numpy.matmul](#). The production MUST never overflow. The accumulation may overflow if and only if in 32 bits.

Domain	ai.onnx
Since version	10
Inputs (2 — 4)	A (non-differentiable) : T1 — N-dimensional matrix A B (non-differentiable) : T2 — N-dimensional matrix B

`a_zero_point` (optional, non-differentiable) : T1 — Zero point tensor for input 'A'. It's optional and default value is 0. It could be a scalar or N-D tensor. Scalar refers to per tensor quantization whereas N-D refers to per row quantization. If the input is 2D of shape [M, K] then zero point tensor may be an M element vector [zp_1, zp_2, ..., zp_M]. If the input is N-D tensor with shape [D1, D2, M, K] then zero point tensor may have shape [D1, D2, M, 1].

`b_zero_point` (optional, non-differentiable) : T2 — Zero point tensor for input 'B'. It's optional and default value is 0. It could be a scalar or a N-D tensor, Scalar refers to per tensor quantization whereas N-D refers to per col quantization. If the input is 2D of shape [K, N] then zero point tensor may be an N element vector [zp_1, zp_2, ..., zp_N]. If the input is N-D tensor with shape [D1, D2, K, N] then zero point tensor may have shape [D1, D2, 1, N].

Outputs	Y (non-differentiable) : T3 — Matrix multiply results from $A * B$
Attributes	None.
Type constraints	T1 : tensor(int8), tensor(uint8) — Constrain input A data type to 8-bit integer tensor. T2 : tensor(int8), tensor(uint8) — Constrain input B data type to 8-bit integer tensor. T3 : tensor(int32) — Constrain output Y data type as 32-bit integer tensor.
Test vectors	test_matmulinteger
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.100. Max

Element-wise max of each of the input tensors (with Numpy-style broadcasting support). All inputs and outputs must have the same data type. This operator supports **multidirectional (i.e., Numpy-style) broadcasting**; for more details please check the doc.

Domain	ai.onnx
Since version	13
Earlier versions	1, 6, 8, 12
Inputs (1 — unbounded)	data_0 (variadic, differentiable) : T — List of tensors for max.
Outputs	max (differentiable) : T — Output tensor.
Attributes	None.
Type constraints	T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(float16), tensor(float), tensor(double), tensor(bfloat16) — Constrain input and output types to numeric tensors.
Test vectors	test_max_example, test_max_one_input, test_max_two_inputs

Not supplied by the upstream source Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.101. MaxPool

MaxPool consumes an input tensor X and applies max pooling across the tensor according to kernel sizes, stride sizes, and pad lengths. max pooling consisting of computing the max on all values of a subset of the input tensor according to the kernel size and downsampling the data into the output tensor Y for further processing. The output spatial shape is calculated differently depending on whether explicit padding is used, where pads is employed, or auto padding is used, where auto_pad is utilized. With explicit padding (<https://pytorch.org/docs/stable/generated/torch.nn.MaxPool2d.html?highlight=maxpool#torch.nn.MaxPool2d>):

```
output_spatial_shape[i] = floor((input_spatial_shape[i] + pad_shape[i] -
dilation[i] * (kernel_shape[i] - 1) - 1) / strides_spatial_shape[i] + 1)
```

or

```
output_spatial_shape[i] = ceil((input_spatial_shape[i] + pad_shape[i] -
dilation[i] * (kernel_shape[i] - 1) - 1) / strides_spatial_shape[i] + 1)
```

if ceil_mode is enabled. pad_shape[i] is the sum of pads along axis i. Sliding windows that would start in the right padded region are ignored.

auto_pad is a DEPRECATED attribute. If you are using them currently, the output spatial shape will be following when ceil_mode is enabled:

```
VALID: output_spatial_shape[i] = ceil((input_spatial_shape[i] - ((kernel_
spatial_shape[i] - 1) * dilations[i] + 1) + 1) / strides_spatial_shape[i])
SAME_UPPER or SAME_LOWER: output_spatial_shape[i] = ceil(input_spatial_shape[i]
/ strides_spatial_shape[i])
```

or when ceil_mode is disabled (https://www.tensorflow.org/api_docs/python/tf/keras/layers/AveragePooling2D):

```
VALID: output_spatial_shape[i] = floor((input_spatial_shape[i] - ((kernel_
spatial_shape[i] - 1) * dilations[i] + 1)) / strides_spatial_shape[i]) + 1
SAME_UPPER or SAME_LOWER: output_spatial_shape[i] = floor((input_spatial_
shape[i] - 1) / strides_spatial_shape[i]) + 1
```

And pad shape will be following if SAME_UPPER or SAME_LOWER:

```
pad_shape[i] = (output_spatial_shape[i] - 1) * strides_spatial_shape[i] +
((kernel_spatial_shape[i] - 1) * dilations[i] + 1) - input_spatial_shape[i]
```

The output of each pooling window is maximum number of elements exclude pad.

Domain	ai.onnx
Since version	22
Earlier versions	1, 8, 10, 11, 12
Inputs	X (differentiable) : T—Input data tensor from the previous operator; dimensions for image case are (N x C x H x W), where N is the batch size, C is the number of channels, and H and W are the height and the width of the data. For non image case, the dimensions are in the form of (N x C x D1 x D2 ... Dn), where N is the batch size. Optionally, if dimension denotation is in effect, the operation expects the input data

tensor to arrive with the dimension denotation of [DATA_BATCH, DATA_CHANNEL, DATA_FEATURE, DATA_FEATURE ...].

Outputs (1 — 2)	Y (differentiable) : T — Output data tensor from average or max pooling across the input tensor. Dimensions will vary based on various kernel, stride, and pad sizes. Floor value of the dimension is used Indices (optional, non-differentiable) : I — Indices tensor from max pooling across the input tensor. The dimensions of indices are the same as output tensor. The values in indices of are the indices of the selected values during pooling. The indices are computed as flatten 1-D tensor, and the indices do not consider padding. So the values in indices are in [0, N x C x D1 x ... x Dn).
Attributes	auto_pad : string (default is NOTSET) — auto_pad must be either NOTSET, SAME_UPPER, SAME_LOWER or VALID. Where default value is NOTSET, which means explicit padding is used. SAME_UPPER or SAME_LOWER mean pad the input so that output_shape[i] = ceil(input_shape[i] / strides[i]) for each axis i. The padding is split between the two sides equally or almost equally (depending on whether it is even or odd). In case the padding is an odd number, the extra padding is added at the end for SAME_UPPER and at the beginning for SAME_LOWER. ceil_mode : int (default is 0) — Whether to use ceil or floor (default) to compute the output shape. dilations : list of ints — Dilation value along each spatial axis of filter. If not present, the dilation defaults to 1 along each spatial axis. kernel_shape : list of ints (required) — The size of the kernel along each axis. pads : list of ints — Padding for the beginning and ending along each spatial axis, it can take any value greater than or equal to 0. The value represent the number of pixels added to the beginning and end part of the corresponding axis. pads format should be as follow [x1_begin, x2_begin...x1_end, x2_end,...], where xi_begin the number of pixels added at the beginning of axis i and xi_end, the number of pixels added at the end of axis i. This attribute cannot be used simultaneously with auto_pad attribute. If not present, the padding defaults to 0 along start and end of each spatial axis. storage_order : int (default is 0) — The storage order of the tensor. 0 is row major, and 1 is column major. This attribute is used only to convert an n-tuple index value into a single integer value for producing the second output. strides : list of ints — Stride along each spatial axis. If not present, the stride defaults to 1 along each spatial axis.
Type constraints	T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(int8), tensor(uint8) — Constrain input and output types to float and 8 bit tensors. I : tensor(int64) — Constrain index tensor to int64
Test vectors	test_maxpool_1d_default, test_maxpool_2d_ceil, test_maxpool_2d_ceil_output_size_reduce_by_one, test_maxpool_2d_default, test_maxpool_2d_dilations, test_maxpool_2d_pads, test_maxpool_2d_precomputed_pads, test_maxpool_2d_precomputed_same_upper, test_maxpool_2d_precomputed_strides, test_maxpool_2d_same_lower, test_maxpool_2d_same_upper, test_maxpool_2d_strides, test_maxpool_2d_uint8, test_maxpool_3d_default, test_maxpool_3d_dilations, test_maxpool_3d_dilations_use_ref_impl, test_maxpool_3d_dilations_use_ref_impl_large, test_maxpool_with_argmax_2d_precomputed_pads, test_maxpool_with_argmax_2d_precomputed_strides

Not supplied by the upstream source Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.102. MaxRoiPool

ROI max pool consumes an input tensor X and region of interests (RoIs) to apply max pooling across each RoI, to produce output 4-D tensor of shape $(\text{num_rois}, \text{channels}, \text{pooled_shape}[0], \text{pooled_shape}[1])$.

Domain	ai.onnx
Since version	22
Earlier versions	1
Inputs	X (differentiable) : T — Input data tensor from the previous operator; dimensions for image case are $(N \times C \times H \times W)$, where N is the batch size, C is the number of channels, and H and W are the height and the width of the data. rois (non-differentiable) : T — RoIs (Regions of Interest) to pool over. Should be a 2-D tensor of shape $(\text{num_rois}, 5)$ given as $[[\text{batch_id}, x1, y1, x2, y2], \dots]$.
Outputs	Y (differentiable) : T — RoI pooled output 4-D tensor of shape $(\text{num_rois}, \text{channels}, \text{pooled_shape}[0], \text{pooled_shape}[1])$.
Attributes	pooled_shape : list of ints (required) — ROI pool output shape (height, width). spatial_scale : float (default is 1.0) — Multiplicative spatial scale factor to translate ROI coordinates from their input scale to the scale used when pooling.
Type constraints	T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain input and output types to float tensors.
Test vectors	None published for this operator.
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.103. MaxUnpool

MaxUnpool essentially computes the partial inverse of the MaxPool op. The input information to this op is typically the output information from a MaxPool op. The first input tensor X is the tensor that needs to be unpooled, which is typically the pooled tensor (first output) from MaxPool. The second input tensor, I , contains the indices to the (locally maximal) elements corresponding to the elements in the first input tensor X . Input tensor I is typically the second output of the MaxPool op. The third (optional) input is a tensor that specifies the output size of the unpooling operation.

MaxUnpool is intended to do 'partial' inverse of the MaxPool op. 'Partial' because all the non-maximal values from the original input to MaxPool are set to zero in the output of the MaxUnpool op. Pooling the result of an unpooling operation should give back the original input to the unpooling op.

MaxUnpool can produce the same output size for several input sizes, which makes unpooling op ambiguous. The third input argument, `output_size`, is meant to disambiguate the op and produce output tensor of known/predictable size.

In addition to the inputs, MaxUnpool takes three attributes, namely `kernel_shape`, `strides`, and `pads`, which define the exact unpooling op. The attributes typically have the same values as the corresponding pooling op that the unpooling op is trying to invert.

Domain	ai.onnx
Since version	22
Earlier versions	9, 11
Inputs (2 — 3)	<code>X</code> (differentiable) : <code>T1</code> — Input data tensor that has to be unpooled. This tensor is typically the first output of the MaxPool op. Dimensions for image case are $(N \times C \times H \times W)$, where N is the batch size, C is the number of channels, and H and W are the height and the width of the data. For non-image case, the dimensions are in the form of $(N \times C \times D1 \times D2 \dots Dn)$, where N is the batch size. Optionally, if dimension denotation is in effect, the operation expects the input data tensor to arrive with the dimension denotation of <code>[DATA_BATCH, DATA_CHANNEL, DATA_FEATURE, DATA_FEATURE ...]</code>. <code>I</code> (non-differentiable) : <code>T2</code> — Input data tensor containing the indices corresponding to elements in the first input tensor <code>X</code>. This tensor is typically the second output of the MaxPool op. Dimensions must be the same as input tensor <code>X</code>. The indices are linear, i.e. computed considering the tensor as flattened 1-D tensor, assuming row-major storage. Also, the linear indices should not consider padding. So the values in indices are in the range $[0, N \times C \times D1 \times \dots \times Dn)$. <code>output_shape</code> (optional, non-differentiable) : <code>T2</code> — The shape of the output can be explicitly set which will cause pads values to be auto generated. If 'output_shape' is specified, 'pads' values are ignored.
Outputs	<code>output</code> (differentiable) : <code>T1</code> — Output data tensor that contains the result of the unpooling.
Attributes	<code>kernel_shape</code> : list of ints (required) — The size of the kernel along each axis. <code>pads</code> : list of ints — Padding for the beginning and ending along each spatial axis, it can take any value greater than or equal to 0. The value represent the number of pixels added to the beginning and end part of the corresponding axis. pads format should be as follow <code>[x1_begin, x2_begin...x1_end, x2_end,...]</code>, where <code>xi_begin</code> the number of pixels added at the beginning of axis <code>i</code> and <code>xi_end</code>, the number of pixels added at the end of axis <code>i</code>. This attribute cannot be used simultaneously with <code>auto_pad</code> attribute. If not present, the padding defaults to 0 along start and end of each spatial axis. <code>strides</code> : list of ints — Stride along each spatial axis. If not present, the stride defaults to 1 along each spatial axis.
Type constraints	<code>T1</code> : <code>tensor(bfloat16)</code>, <code>tensor(float16)</code>, <code>tensor(float)</code>, <code>tensor(double)</code> — Constrain input and output types to float tensors. <code>T2</code> : <code>tensor(int64)</code> — Constrain index tensor to int64
Test vectors	<code>test_maxunpool_export_with_output_shape</code> , <code>test_maxunpool_export_without_output_shape</code>

Not supplied by the upstream source Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.104. Mean

Element-wise mean of each of the input tensors (with Numpy-style broadcasting support). All inputs and outputs must have the same data type. This operator supports **multidirectional (i.e., Numpy-style) broadcasting**; for more details please check the doc.

Domain	ai.onnx
Since version	13
Earlier versions	1, 6, 8
Inputs (1 — unbounded)	data_0 (variadic, differentiable) : T — List of tensors for mean.
Outputs	mean (differentiable) : T — Output tensor.
Attributes	None.
Type constraints	T : tensor(float16), tensor(float), tensor(double), tensor(bfloat16) — Constrain input and output types to float tensors.
Test vectors	test_mean_example, test_mean_one_input, test_mean_two_inputs
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.105. MeanVarianceNormalization

A MeanVarianceNormalization Function: Perform mean variance normalization on the input tensor X using formula: $(X - EX) / \sqrt{E(X - EX)^2}$

Domain	ai.onnx
Since version	13
Earlier versions	9
Inputs	X (differentiable) : T — Input tensor
Outputs	Y (differentiable) : T — Output tensor
Attributes	axes : list of ints (default is ['0', '2', '3']) — A list of integers, along which to reduce. The default is to calculate along axes [0,2,3] for calculating mean and variance along each channel. Two variables with the same C-coordinate are associated with the same mean and variance.

Type constraints	T : tensor(float16), tensor(float), tensor(double), tensor(bfloat16) — Constrain input and output types to all numeric tensors.
Test vectors	test_mvn
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.106. MelWeightMatrix

Generate a MelWeightMatrix that can be used to re-weight a Tensor containing a linearly sampled frequency spectra (from DFT or STFT) into num_mel_bins frequency information based on the [lower_edge_hertz, upper_edge_hertz] range on the mel scale. This function defines the mel scale in terms of a frequency in hertz according to the following formula:

$$\text{mel}(f) = 2595 * \log_{10}(1 + f/700)$$

In the returned matrix, all the triangles (filterbanks) have a peak value of 1.0.

The returned MelWeightMatrix can be used to right-multiply a spectrogram S of shape [frames, num_spectrogram_bins] of linear scale spectrum values (e.g. STFT magnitudes) to generate a “mel spectrogram” M of shape [frames, num_mel_bins].

Domain	ai.onnx
Since version	17
Inputs	num_mel_bins (non-differentiable) : T1 — The number of bands in the mel spectrum. dft_length (non-differentiable) : T1 — The size of the original DFT. The size of the original DFT is used to infer the size of the onesided DFT, which is understood to be floor(dft_length/2) + 1, i.e. the spectrogram only contains the nonredundant DFT bins. sample_rate (non-differentiable) : T1 — Samples per second of the input signal used to create the spectrogram. Used to figure out the frequencies corresponding to each spectrogram bin, which dictates how they are mapped into the mel scale. lower_edge_hertz (non-differentiable) : T2 — Lower bound on the frequencies to be included in the mel spectrum. This corresponds to the lower edge of the lowest triangular band. upper_edge_hertz (non-differentiable) : T2 — The desired top edge of the highest frequency band.
Outputs	output (non-differentiable) : T3 — The Mel Weight Matrix. The output has the shape: [floor(dft_length/2) + 1][num_mel_bins].
Attributes	output_datatype : int (default is 1) — The data type of the output tensor. Strictly must be one of the values from DataType enum in TensorProto whose values correspond to T3. The default value is 1 = FLOAT.
Type constraints	T1 : tensor(int32), tensor(int64) — Constrain to integer tensors. T2 : tensor(float), tensor(float16), tensor(double), tensor(bfloat16) — Constrain to float tensors T3 : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(float16),

tensor(float), tensor(double), tensor(bfloat16) — Constrain to any numerical types.

Test vectors test_melweightmatrix

Not supplied by the upstream source Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.107. Min

Element-wise min of each of the input tensors (with Numpy-style broadcasting support). All inputs and outputs must have the same data type. This operator supports **multidirectional (i.e., Numpy-style) broadcasting**; for more details please check the doc.

Domain ai.onnx

Since version 13

Earlier versions 1, 6, 8, 12

Inputs (1 — unbounded) data_0 (variadic, differentiable) : T — List of tensors for min.
Outputs min (differentiable) : T — Output tensor.

Attributes None.

Type constraints T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(float16), tensor(float), tensor(double), tensor(bfloat16) — Constrain input and output types to numeric tensors.

Test vectors test_min_example, test_min_one_input, test_min_two_inputs

Not supplied by the upstream source Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.108. Mish

Mish: A Self Regularized Non-Monotonic Neural Activation Function.

Perform the linear unit element-wise on the input tensor X using formula:

$\text{mish}(x) = x * \tanh(\text{softplus}(x)) = x * \tanh(\ln(1 + e^{\{x\}}))$

Domain ai.onnx

Since version 22

Earlier versions 18

Inputs X (differentiable) : T — Input tensor

Outputs Y (differentiable) : T — Output tensor

Attributes None.

Type constraints	$T : \text{tensor}(\text{bfloat16}), \text{tensor}(\text{float16}), \text{tensor}(\text{float}), \text{tensor}(\text{double})$ — Constrain input X and output types to float tensors.
Test vectors	test_mish
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.109. Mod

Performs an element-wise binary modulo operation. The `fmod` attribute determines how the quotient is rounded. Its value must be 0 (default) or 1.

If `fmod` is 0, the output is calculated as $A - \text{floor}(A / B) * B$. The result has the same sign as B. For floating-point inputs, the following special cases apply:

- If x is ± 0 and y is nonzero, ± 0 with the sign of y is returned.
- If x is $\pm \infty$ and y is not NaN, NaN is returned.
- If y is ± 0 and x is not NaN, NaN is returned.
- If y is $\pm \infty$ and x is finite and nonzero, x is returned when x and y have the same sign; otherwise, y is returned.
- If either argument is NaN, NaN is returned.

If `fmod` is 1, the output is calculated as $A - \text{trunc}(A / B) * B$. The result has the same sign as A, except that either signed zero may be returned when A is -0 and B is positive. For floating-point inputs, the following special cases apply:

- If x is -0 and y is greater than zero, either $+0$ or -0 may be returned.
- If x is $\pm \infty$ and y is not NaN, NaN is returned.
- If y is ± 0 and x is not NaN, NaN should be returned.
- If y is $\pm \infty$ and x is finite, x is returned.
- If either argument is NaN, NaN is returned.

This operator supports **multidirectional (i.e., NumPy-style) broadcasting**; for more details please check the doc.

Domain	ai.onnx
Since version	28
Earlier versions	10, 13
Inputs	A (differentiable) : T — Dividend tensor B (non-differentiable) : T — Divisor tensor
Outputs	C (differentiable) : T — Remainder tensor
Attributes	<code>fmod</code> : int (default is 0) — Whether the operator should use floor (0) or truncation (1) to calculate the quotient.
Type constraints	$T : \text{tensor}(\text{uint8}), \text{tensor}(\text{uint16}), \text{tensor}(\text{uint32}), \text{tensor}(\text{uint64}), \text{tensor}(\text{int8}), \text{tensor}(\text{int16}), \text{tensor}(\text{int32}), \text{tensor}(\text{int64}), \text{tensor}(\text{float16}), \text{tensor}(\text{float}), \text{tensor}(\text{double}), \text{tensor}(\text{bfloat16})$ — Constrain input and output types to numeric tensors.

Test vectors	test_mod_broadcast, test_mod_int64_fmod, test_mod_mixed_sign_float16, test_mod_float16_mixed_sign_fmod_0, test_mod_mixed_sign_float32, test_mod_float32_mixed_sign_fmod_0, test_mod_mixed_sign_float64, test_mod_float64_mixed_sign_fmod_0, test_mod_mixed_sign_int16, test_mod_mixed_sign_int32, test_mod_mixed_sign_int64, test_mod_mixed_sign_int8, test_mod_uint16, test_mod_uint32, test_mod_uint64, test_mod_uint8
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.110. Mul

Performs element-wise binary multiplication (with Numpy-style broadcasting support).

This operator supports **multidirectional (i.e., Numpy-style) broadcasting**; for more details please check the doc.

(Opset 14 change): Extend supported types to include uint8, int8, uint16, and int16.

Domain	ai.onnx
Since version	14
Earlier versions	1, 6, 7, 13
Inputs	A (differentiable) : T — First operand. B (differentiable) : T — Second operand.
Outputs	C (differentiable) : T — Result, has same element type as two inputs
Attributes	None.
Type constraints	T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(float16), tensor(float), tensor(double), tensor(bfloat16) — Constrain input and output types to all numeric tensors.
Test vectors	test_mul_example, test_mul, test_mul_int8, test_mul_int16, test_mul_uint8, test_mul_uint16, test_mul_uint32, test_mul_uint64, test_mul_bcast
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.111. Multinomial

Generate a tensor of samples from a multinomial distribution according to the probabilities of each of the possible outcomes.

Domain	ai.onnx
Since version	22

Earlier versions	7
Inputs	input : T1 — Input tensor with shape [batch_size, class_size], where class_size is the number of all possible outcomes. Each value along the axis zero represents the unnormalized log-probability of each corresponding outcome in a batch.
Outputs	output : T2 — Output tensor with shape [batch_size, sample_size], where sample_size is the number of times to sample. Each value along the axis zero represents the outcome of the corresponding sample in a batch.
Attributes	dtype : int (default is 6) — (Optional) The data type for the elements of the output tensor, if not specified, we will use int32. sample_size : int (default is 1) — Number of times to sample. seed : float — (Optional) Seed to the random generator, if not specified we will auto generate one.
Type constraints	T1 : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain input types to float tensors. T2 : tensor(int32), tensor(int64) — Constrain output types to integral tensors.
Test vectors	None published for this operator.
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.112. Neg

Neg takes one input data (Tensor<T>) and produces one output data (Tensor<T>) where each element flipped sign, $y = -x$, is applied to the tensor elementwise.

Domain	ai.onnx
Since version	13
Earlier versions	1, 6
Inputs	X (differentiable) : T — Input tensor
Outputs	Y (differentiable) : T — Output tensor
Attributes	None.
Type constraints	T : tensor(float), tensor(int32), tensor(int8), tensor(int16), tensor(int64), tensor(float16), tensor(double), tensor(bfloat16) — Constrain input and output types to signed numeric tensors.
Test vectors	test_neg_example, test_neg
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.113. NegativeLogLikelihoodLoss

A NegativeLogLikelihoodLoss operator computes (weighted) negative log likelihood loss. Its “input” tensor has the shape of (N, C, d1, d2, ..., dk) where $k \geq 0$. The “input” tensor contains log-probabilities for input[n, :, d1, d2, ..., dk] being in a class of [0, C). The operator’s “target” input tensor has the shape of (N, d1, d2, ..., dk). It encodes class labels (one of C classes) or it may contain a special value (indicated by an attribute ignore_index) for N x d1 x d2 x ... x dk samples. The loss value for input[n, :, d1, d2, ..., dk] being classified as class $c = \text{target}[n][d_1][d_2] \dots [d_k]$ is computed as:

$$\text{loss}[n][d_1][d_2] \dots [d_k] = -\text{input}[n][c][d_1][d_2] \dots [d_k].$$

When an optional “weight” is provided, the sample loss is calculated as:

$$\text{loss}[n][d_1][d_2] \dots [d_k] = -\text{input}[n][c][d_1][d_2] \dots [d_k] * \text{weight}[c].$$

loss is zero for the case when target-value equals ignore_index.

$$\text{loss}[n][d_1][d_2] \dots [d_k] = 0, \text{ when } \text{target}[n][d_1][d_2] \dots [d_k] = \text{ignore_index}$$

If “reduction” attribute is set to “none”, the operator’s output will be the above loss with shape (N, d1, d2, ..., dk). If “reduction” attribute is set to “mean” (the default attribute value), the output loss is (weight) averaged:

mean(loss), if “weight” is not provided,

or if weight is provided,

$$\text{sum}(\text{loss}) / \text{sum}(\text{weight}[\text{target}[n][d_1][d_2] \dots [d_k]]), \text{ for all samples.}$$

If “reduction” attribute is set to “sum”, the output is a scalar: sum(loss).

See also <https://pytorch.org/docs/stable/nn.html#torch.nn.NLLLoss>.

Example 1:

```
// negative log likelihood loss, "none" reduction
N, C, d1 = 2, 3, 2
input = [[1.0, 2.0], [2.0, 2.0], [3.0, 2.0]],
        [[0.0, 1.0], [2.0, 2.0], [1.0, 2]]
target = [[2, 1], [0, 2]]

loss = np.zeros((N, d1))
for n in range(N):
    for d1 in range(d1):
        c = target[n][d1]
        loss[n][d1] = -input[n][c][d1]

// print(loss)
// [[-3. -2.]
//  [-0. -2.]
```

Example 2:

```
// weighted negative log likelihood loss, sum reduction
N, C, d1 = 2, 3, 2
input = [[1.0, 2.0], [2.0, 2.0], [3.0, 2.0]],
        [[0.0, 1.0], [2.0, 2.0], [1.0, 2]]
target = [[2, 1], [0, 2]]
weight = [0.2, 0.3, 0.1]
loss = np.zeros((N, d1))
for n in range(N):
```

```

    for d_1 in range(d1):
        c = target[n][d_1]
        loss[n][d_1] = -input[n][c][d_1] * weight[c]

loss = np.sum(loss)
// print(loss)
// -1.1

```

Example 3:

```

// weighted negative log likelihood loss, mean reduction
N, C, d1 = 2, 3, 2
input = [[[1.0, 2.0], [2.0, 2.0], [3.0, 2.0]],
         [[0.0, 1.0], [2.0, 2.0], [1.0, 2]]]
target = [[2, 1], [0, 2]]
weight = [0.2, 0.3, 0.1]
loss = np.zeros((N, d1))
weight_total = 0
for n in range(N):
    for d_1 in range(d1):
        c = target[n][d_1]
        loss[n][d_1] = -input[n][c][d_1] * weight[c]
        weight_total = weight_total + weight[c]

loss = np.sum(loss) / weight_total
// print(loss)
// -1.57

```

Domain	ai.onnx
Since version	22
Earlier versions	12, 13
Inputs (2 — 3)	input (differentiable) : T—Input tensor of shape (N, C) or (N, C, d1, d2, ..., dk). target (non-differentiable) : Tind—Target tensor of shape (N) or (N, d1, d2, ..., dk). Target element value shall be in range of [0, C). If ignore_index is specified, it may have a value outside [0, C) and the target values should either be in the range [0, C) or have the value ignore_index. weight (optional, non-differentiable) : T—Optional rescaling weight tensor. If given, it has to be a tensor of size C. Otherwise, it is treated as if having all ones.
Outputs	loss (differentiable) : T—The negative log likelihood loss
Attributes	ignore_index : int—Specifies a target value that is ignored and does not contribute to the input gradient. It's an optional value. reduction : string (default is mean) —Type of reduction to apply to loss: none, sum, mean (default). 'none': the output is the loss for each sample. 'sum': the output will be summed. 'mean': the sum of the output will be divided by the sum of applied weights.
Type constraints	T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) —Constrain input, weight, and output types to floating-point tensors. Tind : tensor(int32), tensor(int64) —Constrain target to integer types
Test vectors	test_nllloss_NC, test_nllloss_NCd1, test_nllloss_NCd1_ii, test_nllloss_NCd1_mean_weight_negative_ii, test_nllloss_NCd1_

weight, test_nllloss_NCd1_weight_ii, test_nllloss_NCd1d2, test_nllloss_NCd1d2_no_weight_reduction_mean_ii, test_nllloss_NCd1d2_reduction_mean, test_nllloss_NCd1d2_reduction_sum, test_nllloss_NCd1d2_with_weight, test_nllloss_NCd1d2_with_weight_reduction_mean, test_nllloss_NCd1d2_with_weight_reduction_sum, test_nllloss_NCd1d2_with_weight_reduction_sum_ii, test_nllloss_NCd1d2d3_none_no_weight_negative_ii, test_nllloss_NCd1d2d3_sum_weight_high_ii, test_nllloss_NCd1d2d3d4d5_mean_weight, test_nllloss_NCd1d2d3d4d5_none_no_weight

Not supplied by the upstream source Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.114. NonMaxSuppression

Filter out boxes that have high intersection-over-union (IOU) overlap with previously selected boxes. Bounding boxes with score less than `score_threshold` are removed. Bounding box format is indicated by attribute `center_point_box`. Boxes are suppressed if their IOU with a previously selected box is strictly greater than `iou_threshold` (i.e., boxes with IOU exactly equal to the threshold are kept). Note that this algorithm is agnostic to where the origin is in the coordinate system and more generally is invariant to orthogonal transformations and translations of the coordinate system; thus translating or reflections of the coordinate system result in the same boxes being selected by the algorithm. The `selected_indices` output is a set of integers indexing into the input collection of bounding boxes representing the selected boxes. The bounding box coordinates corresponding to the selected indices can then be obtained using the `Gather` or `GatherND` operation.

Domain `ai.onnx`

Since version 11

Earlier versions 10

Inputs (2 — 5) `boxes` : `tensor(float)` — An input tensor with shape `[num_batches, spatial_dimension, 4]`. The single box data format is indicated by `center_point_box`.
`scores` : `tensor(float)` — An input tensor with shape `[num_batches, num_classes, spatial_dimension]`
`max_output_boxes_per_class` (optional) : `tensor(int64)` — Integer representing the maximum number of boxes to be selected per batch per class. It is a scalar. Default to 0, which means no output.
`iou_threshold` (optional) : `tensor(float)` — Float representing the threshold for deciding whether boxes overlap too much with respect to IOU. Boxes with IoU strictly greater than this threshold are suppressed. It is scalar. Value range `[0, 1]`. Default to 0.
`score_threshold` (optional) : `tensor(float)` — Float representing the threshold for deciding when to remove boxes based on score. It is a scalar.

Outputs `selected_indices` : `tensor(int64)` — selected indices from the boxes tensor. `[num_selected_indices, 3]`, the selected index format is `[batch_index, class_index, box_index]`.

Attributes `center_point_box` : `int` (default is 0) — Integer indicate the format of the box data. The default is 0. 0 — the box data is supplied as `[y1, x1, y2, x2]` where `(y1, x1)` and `(y2, x2)` are the coordinates of any diagonal pair of box corners and the coordinates can be provided as

normalized (i.e., lying in the interval [0, 1]) or absolute. Mostly used for TF models. 1 — the box data is supplied as [x_center, y_center, width, height]. Mostly used for Pytorch models.

Type constraints	None.
Test vectors	test_nonmaxsuppression_center_point_box_format, test_nonmaxsuppression_flipped_coordinates, test_nonmaxsuppression_identical_boxes, test_nonmaxsuppression_iou_threshold_boundary, test_nonmaxsuppression_limit_output_size, test_nonmaxsuppression_single_box, test_nonmaxsuppression_suppress_by_IOU, test_nonmaxsuppression_suppress_by_IOU_and_scores, test_nonmaxsuppression_two_batches, test_nonmaxsuppression_two_classes
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.115. NonZero

Returns the indices of the elements that are non-zero (in row-major order — by dimension). NonZero behaves similar to `numpy.nonzero`: <https://docs.scipy.org/doc/numpy/reference/generated/numpy.nonzero.html>, but for scalar input, NonZero produces output shape (0, N) instead of (1, N), which is different from Numpy's behavior.

Domain	ai.onnx
Since version	13
Earlier versions	9
Inputs	X (non-differentiable) : T — input
Outputs	Y (non-differentiable) : tensor(int64) — output
Attributes	None.
Type constraints	T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128) — Constrain to all tensor types.
Test vectors	test_nonzero_example
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.116. Not

Returns the negation of the input tensor element-wise.

Domain	ai.onnx
--------	---------

Since version	1
Inputs	X (non-differentiable) : T—Input tensor
Outputs	Y (non-differentiable) : T—Output tensor
Attributes	None.
Type constraints	T : tensor(bool) — Constrain input/output to boolean tensors.
Test vectors	test_not_2d, test_not_3d, test_not_4d
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.117. OneHot

Produces a one-hot tensor based on inputs. The locations represented by the index values in the 'indices' input tensor will have 'on_value' and the other locations will have 'off_value' in the output tensor, where 'on_value' and 'off_value' are specified as part of required input argument 'values', which is a two-element tensor of format [off_value, on_value]. The rank of the output tensor will be one greater than the rank of the input tensor. The additional dimension is for one-hot representation. The additional dimension will be inserted at the position specified by 'axis'. If 'axis' is not specified then the additional dimension will be inserted as the innermost dimension, i.e. axis=-1. The size of the additional dimension is specified by required scalar input 'depth'. The type of the output tensor is the same as the type of the 'values' input. Any entries in the 'indices' input tensor with values outside the range [-depth, depth-1] will result in one-hot representation with all 'off_value' values in the output tensor.

```
when axis = 0:
output[input[i, j, k], i, j, k] = 1 for all i, j, k and 0 otherwise.
```

```
when axis = -1:
output[i, j, k, input[i, j, k]] = 1 for all i, j, k and 0 otherwise.
```

Domain	ai.onnx
Since version	28
Earlier versions	9, 11
Inputs	indices (non-differentiable) : T1—Input tensor containing indices. Any entries in the 'indices' input tensor with values outside the range [-depth, depth-1] will result in one-hot representation with all 'off_value' values in the output tensor. In case 'indices' is of non-integer type, the values will be casted to int64 before use. depth (non-differentiable) : T2—Scalar or Rank 1 tensor containing exactly one element, specifying the number of classes in one-hot tensor. This is also the size of the one-hot dimension (specified by 'axis' attribute) added on in the output tensor. The values in the 'indices' input tensor are expected to be in the range [-depth, depth-1]. In case 'depth' is of non-integer type, it will be casted to int64 before use. values (non-differentiable) : T3—Rank 1 tensor containing exactly two elements, in the format [off_value, on_value], where 'on_value' is the value used for filling locations specified in 'indices' input tensor, and 'off_value' is the value used for filling locations other than those specified in 'indices' input tensor.

Outputs	output (non-differentiable) : T3—Tensor of rank one greater than input tensor 'indices', i.e. $\text{rank}(\text{output}) = \text{rank}(\text{indices}) + 1$. The data type for the elements of the output tensor is the same as the type of input 'values' is used.
Attributes	axis : int (default is -1) —(Optional) Axis along which one-hot representation is added. Default: axis=-1. axis=-1 means that the additional dimension will be inserted as the innermost/last dimension in the output tensor. Negative value means counting dimensions from the back. Accepted range is $[-r-1, r]$ where $r = \text{rank}(\text{indices})$.
Type constraints	T1 : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(float16), tensor(float), tensor(double) — Constrain input to only numeric types. T2 : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(float16), tensor(float), tensor(double) — Constrain input to only numeric types. T3 : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128) — Constrain to any tensor type.
Test vectors	test_onehot_with_axis, test_onehot_with_bfloat16_values, test_onehot_with_negative_axis, test_onehot_negative_indices, test_onehot_out_of_range_indices, test_onehot_without_axis
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.118. Optional

Constructs an optional-type value containing either an empty optional of a certain type specified by the attribute, or a non-empty value containing the input element.

Domain	ai.onnx
Since version	28
Earlier versions	15
Inputs (0 — 1)	input (optional) : V—The input element.
Outputs	output : 0—The optional output enclosing the input element.
Attributes	type : type_proto—Type of the element in the optional output
Type constraints	V : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128), tensor(float8e4m3fn), tensor(float8e4m3fnuz), tensor(float8e5m2), tensor(float8e5m2fnuz), tensor(uint4),

tensor(int4), tensor(float4e2m1), tensor(float8e8m0),
 tensor(uint2), tensor(int2), tensor(float6e2m3),
 tensor(float6e3m2), seq(tensor(uint8)), seq(tensor(uint16)),
 seq(tensor(uint32)), seq(tensor(uint64)), seq(tensor(int8)),
 seq(tensor(int16)), seq(tensor(int32)), seq(tensor(int64)),
 seq(tensor(bfloat16)), seq(tensor(float16)),
 seq(tensor(float)), seq(tensor(double)), seq(tensor(string)),
 seq(tensor(bool)), seq(tensor(complex64)),
 seq(tensor(complex128)), seq(tensor(float8e4m3fn)),
 seq(tensor(float8e4m3fnuz)), seq(tensor(float8e5m2)),
 seq(tensor(float8e5m2fnuz)), seq(tensor(uint4)),
 seq(tensor(int4)), seq(tensor(float4e2m1)),
 seq(tensor(float8e8m0)), seq(tensor(uint2)),
 seq(tensor(int2)), seq(tensor(float6e2m3)),
 seq(tensor(float6e3m2)) — Constrain input type to all tensor and
 sequence types.

0 : optional(tensor(uint8)), optional(tensor(uint16)),
 optional(tensor(uint32)), optional(tensor(uint64)),
 optional(tensor(int8)), optional(tensor(int16)),
 optional(tensor(int32)), optional(tensor(int64)),
 optional(tensor(bfloat16)), optional(tensor(float16)),
 optional(tensor(float)), optional(tensor(double)),
 optional(tensor(string)), optional(tensor(bool)),
 optional(tensor(complex64)), optional(tensor(complex128)),
 optional(tensor(float8e4m3fn)),
 optional(tensor(float8e4m3fnuz)),
 optional(tensor(float8e5m2)),
 optional(tensor(float8e5m2fnuz)), optional(tensor(uint4)),
 optional(tensor(int4)), optional(tensor(float4e2m1)),
 optional(tensor(float8e8m0)), optional(tensor(uint2)),
 optional(tensor(int2)), optional(tensor(float6e2m3)),
 optional(tensor(float6e3m2)), optional(seq(tensor(uint8))),
 optional(seq(tensor(uint16))), optional(seq(tensor(uint32))),
 optional(seq(tensor(uint64))), optional(seq(tensor(int8))),
 optional(seq(tensor(int16))), optional(seq(tensor(int32))),
 optional(seq(tensor(int64))), optional(seq(tensor(bfloat16))),
 optional(seq(tensor(float16))), optional(seq(tensor(float))),
 optional(seq(tensor(double))), optional(seq(tensor(string))),
 optional(seq(tensor(bool))), optional(seq(tensor(complex64))),
 optional(seq(tensor(complex128))),
 optional(seq(tensor(float8e4m3fn))),
 optional(seq(tensor(float8e4m3fnuz))),
 optional(seq(tensor(float8e5m2))),
 optional(seq(tensor(float8e5m2fnuz))),
 optional(seq(tensor(uint4))), optional(seq(tensor(int4))),
 optional(seq(tensor(float4e2m1))),
 optional(seq(tensor(float8e8m0))),
 optional(seq(tensor(uint2))), optional(seq(tensor(int2))),
 optional(seq(tensor(float6e2m3))),
 optional(seq(tensor(float6e3m2))) — Constrain output type to all
 optional tensor or optional sequence types.

Test
 vectors
 Not
 supplied
 by the
 upstream
 source

None published for this operator.

Shape inference, determinism, errors. See the editorial note at the head of
 this clause.

5.119. OptionalGetElement

If the input is a tensor or sequence type, it returns the input. If the input is an optional type, it outputs the element in the input. It is an error if the input is an empty optional-type (i.e. does not have an element) and the behavior is undefined in this case.

Domain	ai.onnx
Since version	28
Earlier versions	15, 18
Inputs	input : 0 — The optional input.
Outputs	output : V — Output element in the optional input.
Attributes	None.
Type constraints	0 : optional(tensor(uint8)), optional(tensor(uint16)), optional(tensor(uint32)), optional(tensor(uint64)), optional(tensor(int8)), optional(tensor(int16)), optional(tensor(int32)), optional(tensor(int64)), optional(tensor(bfloat16)), optional(tensor(float16)), optional(tensor(float)), optional(tensor(double)), optional(tensor(string)), optional(tensor(bool)), optional(tensor(complex64)), optional(tensor(complex128)), optional(tensor(float8e4m3fn)), optional(tensor(float8e4m3fnuz)), optional(tensor(float8e5m2)), optional(tensor(float8e5m2fnuz)), optional(tensor(uint4)), optional(tensor(int4)), optional(tensor(float4e2m1)), optional(tensor(float8e8m0)), optional(tensor(uint2)), optional(tensor(int2)), optional(tensor(float6e2m3)), optional(tensor(float6e3m2)), optional(seq(tensor(uint8))), optional(seq(tensor(uint16))), optional(seq(tensor(uint32))), optional(seq(tensor(uint64))), optional(seq(tensor(int8))), optional(seq(tensor(int16))), optional(seq(tensor(int32))), optional(seq(tensor(int64))), optional(seq(tensor(bfloat16))), optional(seq(tensor(float16))), optional(seq(tensor(float))), optional(seq(tensor(double))), optional(seq(tensor(string))), optional(seq(tensor(bool))), optional(seq(tensor(complex64))), optional(seq(tensor(complex128))), optional(seq(tensor(float8e4m3fn))), optional(seq(tensor(float8e4m3fnuz))), optional(seq(tensor(float8e5m2))), optional(seq(tensor(float8e5m2fnuz))), optional(seq(tensor(uint4))), optional(seq(tensor(int4))), optional(seq(tensor(float4e2m1))), optional(seq(tensor(float8e8m0))), optional(seq(tensor(uint2))), optional(seq(tensor(int2))), optional(seq(tensor(float6e2m3))), optional(seq(tensor(float6e3m2))), tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128), tensor(float8e4m3fn), tensor(float8e4m3fnuz),

tensor(float8e5m2), tensor(float8e5m2fnuz), tensor(uint4),
 tensor(int4), tensor(float4e2m1), tensor(float8e8m0),
 tensor(uint2), tensor(int2), tensor(float6e2m3),
 tensor(float6e3m2), seq(tensor(uint8)), seq(tensor(uint16)),
 seq(tensor(uint32)), seq(tensor(uint64)), seq(tensor(int8)),
 seq(tensor(int16)), seq(tensor(int32)), seq(tensor(int64)),
 seq(tensor(bfloat16)), seq(tensor(float16)),
 seq(tensor(float)), seq(tensor(double)), seq(tensor(string)),
 seq(tensor(bool)), seq(tensor(complex64)),
 seq(tensor(complex128)), seq(tensor(float8e4m3fn)),
 seq(tensor(float8e4m3fnuz)), seq(tensor(float8e5m2)),
 seq(tensor(float8e5m2fnuz)), seq(tensor(uint4)),
 seq(tensor(int4)), seq(tensor(float4e2m1)),
 seq(tensor(float8e8m0)), seq(tensor(uint2)),
 seq(tensor(int2)), seq(tensor(float6e2m3)),
 seq(tensor(float6e3m2)) — Constrain input type to optional, tensor and
 sequence types.

V : tensor(uint8), tensor(uint16), tensor(uint32),
 tensor(uint64), tensor(int8), tensor(int16), tensor(int32),
 tensor(int64), tensor(bfloat16), tensor(float16),
 tensor(float), tensor(double), tensor(string),
 tensor(bool), tensor(complex64), tensor(complex128),
 tensor(float8e4m3fn), tensor(float8e4m3fnuz),
 tensor(float8e5m2), tensor(float8e5m2fnuz), tensor(uint4),
 tensor(int4), tensor(float4e2m1), tensor(float8e8m0),
 tensor(uint2), tensor(int2), tensor(float6e2m3),
 tensor(float6e3m2), seq(tensor(uint8)), seq(tensor(uint16)),
 seq(tensor(uint32)), seq(tensor(uint64)), seq(tensor(int8)),
 seq(tensor(int16)), seq(tensor(int32)), seq(tensor(int64)),
 seq(tensor(bfloat16)), seq(tensor(float16)),
 seq(tensor(float)), seq(tensor(double)), seq(tensor(string)),
 seq(tensor(bool)), seq(tensor(complex64)),
 seq(tensor(complex128)), seq(tensor(float8e4m3fn)),
 seq(tensor(float8e4m3fnuz)), seq(tensor(float8e5m2)),
 seq(tensor(float8e5m2fnuz)), seq(tensor(uint4)),
 seq(tensor(int4)), seq(tensor(float4e2m1)),
 seq(tensor(float8e8m0)), seq(tensor(uint2)),
 seq(tensor(int2)), seq(tensor(float6e2m3)),
 seq(tensor(float6e3m2)) — Constrain output type to all tensor or
 sequence types.

Test vectors	None published for this operator.
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.120. OptionalHasElement

Returns true if (1) the input is an optional-type and contains an element, or, (2) the input is a tensor or sequence type. If the input is not provided or is an empty optional-type, this op returns false.

Domain	ai.onnx
Since version	28

Earlier versions	15, 18
Inputs (0 — 1)	input (optional) : 0 — The optional input.
Outputs	output : B — A scalar boolean tensor. If true, it indicates that optional-type input contains an element. Otherwise, it is empty.
Attributes	None.
Type constraints	0 : optional(tensor(uint8)), optional(tensor(uint16)), optional(tensor(uint32)), optional(tensor(uint64)), optional(tensor(int8)), optional(tensor(int16)), optional(tensor(int32)), optional(tensor(int64)), optional(tensor(bfloat16)), optional(tensor(float16)), optional(tensor(float)), optional(tensor(double)), optional(tensor(string)), optional(tensor(bool)), optional(tensor(complex64)), optional(tensor(complex128)), optional(tensor(float8e4m3fn)), optional(tensor(float8e4m3fnuz)), optional(tensor(float8e5m2)), optional(tensor(float8e5m2fnuz)), optional(tensor(uint4)), optional(tensor(int4)), optional(tensor(float4e2m1)), optional(tensor(float8e8m0)), optional(tensor(uint2)), optional(tensor(int2)), optional(tensor(float6e2m3)), optional(tensor(float6e3m2)), optional(seq(tensor(uint8))), optional(seq(tensor(uint16))), optional(seq(tensor(uint32))), optional(seq(tensor(uint64))), optional(seq(tensor(int8))), optional(seq(tensor(int16))), optional(seq(tensor(int32))), optional(seq(tensor(int64))), optional(seq(tensor(bfloat16))), optional(seq(tensor(float16))), optional(seq(tensor(float))), optional(seq(tensor(double))), optional(seq(tensor(string))), optional(seq(tensor(bool))), optional(seq(tensor(complex64))), optional(seq(tensor(complex128))), optional(seq(tensor(float8e4m3fn))), optional(seq(tensor(float8e4m3fnuz))), optional(seq(tensor(float8e5m2))), optional(seq(tensor(float8e5m2fnuz))), optional(seq(tensor(uint4))), optional(seq(tensor(int4))), optional(seq(tensor(float4e2m1))), optional(seq(tensor(float8e8m0))), optional(seq(tensor(uint2))), optional(seq(tensor(int2))), optional(seq(tensor(float6e2m3))), optional(seq(tensor(float6e3m2))), tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128), tensor(float8e4m3fn), tensor(float8e4m3fnuz), tensor(float8e5m2), tensor(float8e5m2fnuz), tensor(uint4), tensor(int4), tensor(float4e2m1), tensor(float8e8m0), tensor(uint2), tensor(int2), tensor(float6e2m3), tensor(float6e3m2), seq(tensor(uint8)), seq(tensor(uint16)), seq(tensor(uint32)), seq(tensor(uint64)), seq(tensor(int8)), seq(tensor(int16)), seq(tensor(int32)), seq(tensor(int64)), seq(tensor(bfloat16)), seq(tensor(float16)), seq(tensor(float)), seq(tensor(double)), seq(tensor(string)), seq(tensor(bool)), seq(tensor(complex64)), seq(tensor(complex128)), seq(tensor(float8e4m3fn)), seq(tensor(float8e4m3fnuz)), seq(tensor(float8e5m2)),

seq(tensor(float8e5m2fnuz)), seq(tensor(uint4)),
 seq(tensor(int4)), seq(tensor(float4e2m1)),
 seq(tensor(float8e8m0)), seq(tensor(uint2)),
 seq(tensor(int2)), seq(tensor(float6e2m3)),
 seq(tensor(float6e3m2)) — Constrain input type to optional, tensor and
 sequence types.
 B : tensor(bool) — Constrain output to a boolean tensor.

Test vectors test_optional_get_element_optional_sequence, test_optional_get_
 element_sequence, test_optional_get_element_optional_tensor,
 test_optional_get_element_tensor

Not supplied by the upstream source Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.121. Or

Returns the tensor resulted from performing the or logical operation elementwise on the input tensors A and B (with Numpy-style broadcasting support).

This operator supports **multidirectional (i.e., Numpy-style) broadcasting**; for more details please check the doc.

Domain	ai.onnx
Since version	7
Earlier versions	1
Inputs	A (non-differentiable) : T — First input operand for the logical operator. B (non-differentiable) : T — Second input operand for the logical operator.
Outputs	C (non-differentiable) : T1 — Result tensor.
Attributes	None.
Type constraints	T : tensor(bool) — Constrain input to boolean tensor. T1 : tensor(bool) — Constrain output to boolean tensor.
Test vectors	test_or2d, test_or3d, test_or4d, test_or_bcast3v1d, test_or_bcast3v2d, test_or_bcast4v2d, test_or_bcast4v3d, test_or_bcast4v4d
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.122. PRelu

PRelu takes input data (Tensor<T>) and slope tensor as input, and produces one output data (Tensor<T>) where the function $f(x) = \text{slope} * x$ for $x < 0$, $f(x) = x$ for $x \geq 0$., is applied to the data tensor elementwise. This operator supports **unidirectional broadcasting**

(tensor slope should be unidirectional broadcastable to input tensor X); for more details please check the doc.

Domain	ai.onnx
Since version	16
Earlier versions	1, 6, 7, 9
Inputs	X (differentiable) : T — Input tensor slope (differentiable) : T — Slope tensor. The shape of slope can be smaller than first input X; if so, its shape must be unidirectional broadcastable to X
Outputs	Y (differentiable) : T — Output tensor (same size as X)
Attributes	None.
Type constraints	T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(uint32), tensor(uint64), tensor(int32), tensor(int64) — Constrain input and output types to float/int tensors.
Test vectors	test_prelu_example, test_prelu_broadcast
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.123. Pad

Given a tensor containing the data to be padded (data), a tensor containing the number of start and end pad values for axis (pads), (optionally) a mode, and (optionally) constant_value, a padded tensor (output) is generated.

The four supported modes are (similar to corresponding modes supported by `numpy.pad`):

- 1) `constant`(default) — pads with a given constant value as specified by `constant_value` (which defaults to 0, empty string, or False)
- 2) `reflect` — pads with the reflection of the vector mirrored on the first and last values of the vector along each axis
- 3) `edge` — pads with the edge values of array
- 4) `wrap` — wrap-around padding as if the data tensor forms a torus

Example 1 (constant mode):

Insert 0 pads to the beginning of the second dimension.

```
data = [
    [1.0, 1.2],
    [2.3, 3.4],
    [4.5, 5.7],
]

pads = [0, 2, 0, 0]
```

ONNX 1-2 (unofficial draft)

```
mode = 'constant'
```

```
constant_value = 0.0
```

```
output = [  
    [0.0, 0.0, 1.0, 1.2],  
    [0.0, 0.0, 2.3, 3.4],  
    [0.0, 0.0, 4.5, 5.7],  
]
```

Example 2 (reflect mode):

```
data = [  
    [1.0, 1.2],  
    [2.3, 3.4],  
    [4.5, 5.7],  
]
```

```
pads = [0, 2, 0, 0]
```

```
mode = 'reflect'
```

```
output = [  
    [1.0, 1.2, 1.0, 1.2],  
    [2.3, 3.4, 2.3, 3.4],  
    [4.5, 5.7, 4.5, 5.7],  
]
```

Example 3 (edge mode):

```
data = [  
    [1.0, 1.2],  
    [2.3, 3.4],  
    [4.5, 5.7],  
]
```

```
pads = [0, 2, 0, 0]
```

```
mode = 'edge'
```

```
output = [  
    [1.0, 1.0, 1.0, 1.2],  
    [2.3, 2.3, 2.3, 3.4],  
    [4.5, 4.5, 4.5, 5.7],  
]
```

Example 4 (wrap mode):

```
data = [  
    [1.0, 1.2],  
    [2.3, 3.4],  
    [4.5, 5.7],  
]
```

```
pads = [2, 1, 1, 1]
```

```
mode = 'wrap'
```

```
output = [  
    [3.4, 2.3, 3.4, 2.3],  
    [5.7, 4.5, 5.7, 4.5],  
    [1.2, 1.0, 1.2, 1.0],  
]
```

<pre> [3.4, 2.3, 3.4, 2.3], [5.7, 4.5, 5.7, 4.5], [1.2, 1.0, 1.2, 1.0],] </pre>	
Domain	ai.onnx
Since version	25
Earlier versions	1, 2, 11, 13, 18, 19, 21, 23, 24
Inputs (2 — 4)	data (differentiable) : T — Input tensor. pads (non-differentiable) : tensor(int64) — Tensor of integers indicating the number of padding elements to add or remove (if negative) at the beginning and end of each axis. For 2D input tensor, it is the number of pixels. pads should be a 1D tensor of shape $[2 * \text{num_axes}]$ where num_axes refers to the number of elements in the axes input or the input rank if axes are not provided explicitly. pads format should be: $[\text{x1_begin}, \text{x2_begin}, \dots, \text{x1_end}, \text{x2_end}, \dots]$, where x1_begin is the number of pad values added at the beginning of axis axes[i] and x1_end, the number of pad values added at the end of axis axes[i]. constant_value (optional, non-differentiable) : T — (Optional) A scalar value to be used if the mode chosen is constant (by default it is 0, empty string or False). axes (optional, non-differentiable) : Tind — 1-D tensor of axes that pads apply to. Negative value means counting dimensions from the back. Accepted range is $[-r, r-1]$ where $r = \text{rank}(\text{data})$. Behavior is undefined if an axis is repeated. If not provided, all axes are assumed $([0, 1, \dots, \text{input_rank}-1])$.
Outputs	output (differentiable) : T — Tensor after padding.
Attributes	mode : string (default is constant) — Supported modes: constant(default), reflect, edge, wrap
Type constraints	T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128), tensor(float8e4m3fn), tensor(float8e4m3fnuz), tensor(float8e5m2), tensor(float8e5m2fnuz), tensor(uint4), tensor(int4), tensor(float4e2m1), tensor(float8e8m0), tensor(uint2), tensor(int2) — Constrain input and output types to all tensor types up to IRv13. Tind : tensor(int32), tensor(int64) — Constrain indices to integer types
Test vectors	test_constant_pad, test_constant_pad_axes, test_constant_pad_negative_axes
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.124. Pow

Pow takes input data (Tensor<T>) and exponent Tensor, and produces one output data (Tensor<T>) where the function $f(x) = x^{\text{exponent}}$, is applied to the data tensor elementwise.

This operator supports **multidirectional (i.e., Numpy-style) broadcasting**; for more details please check the doc.

Domain	ai.onnx
Since version	15
Earlier versions	1, 7, 12, 13
Inputs	X (differentiable) : T — First operand, base of the exponent. Y (differentiable) : T1 — Second operand, power of the exponent.
Outputs	Z (differentiable) : T — Output tensor
Attributes	None.
Type constraints	T : tensor(int32), tensor(int64), tensor(float16), tensor(float), tensor(double), tensor(bfloat16) — Constrain input X and output types to float/int tensors. T1 : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(float16), tensor(float), tensor(double), tensor(bfloat16) — Constrain input Y types to float/int tensors.
Test vectors	test_pow_example, test_pow, test_pow_bcast_scalar, test_pow_bcast_array, test_pow_types_float32_int64, test_pow_types_int64_float32, test_pow_types_float32_int32, test_pow_types_int32_float32, test_pow_types_float32_uint64, test_pow_types_float32_uint32, test_pow_types_int64_int64, test_pow_types_int32_int32
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.125. QLinearConv

The convolution operator consumes a quantized input tensor, its scale and zero point, a quantized filter, its scale and zero point, and output's scale and zero point, and computes the quantized output. Each scale and zero-point pair must have same shape. It means they must be either scalars (per tensor) or 1-D tensors (per output channel). Each input or output and its related zero point must have same type. When bias is present it must be quantized using scale = input scale * weight scale and zero point as 0.

Domain	ai.onnx
Since version	10
Inputs (8 — 9)	x : T1 — Input data tensor from previous layer; has size (N x C x H x W), where N is the batch size, C is the number of channels, and H and W are the height and width. Note that this is for the 2D image. Otherwise the size is (N x C x D1 x D2 ... x Dn). Optionally, if dimension denotation is in effect, the operation expects input data tensor to arrive with the dimension denotation of [DATA_BATCH, DATA_CHANNEL, DATA_FEATURE, DATA_FEATURE ...]. x_scale : tensor(float) — Scale tensor for input 'x'. It's a scalar, which means a per-tensor/layer quantization.

`x_zero_point` : T1 — Zero point tensor for input 'x'. It's a scalar, which means a per-tensor/layer quantization.

`w` : T2 — The weight tensor that will be used in the convolutions; has size (M x C/group x kH x kW), where C is the number of channels, and kH and kW are the height and width of the kernel, and M is the number of feature maps. For more than 2 dimensions, the kernel shape will be (M x C/group x k1 x k2 x ... x kn), where (k1 x k2 x ... x kn) is the dimension of the kernel. Optionally, if dimension denotation is in effect, the operation expects the weight tensor to arrive with the dimension denotation of [FILTER_OUT_CHANNEL, FILTER_IN_CHANNEL, FILTER_SPATIAL, FILTER_SPATIAL ...]. $X.shape[1] == (W.shape[1] * group) == C$ (assuming zero based indices for the shape array). Or in other words FILTER_IN_CHANNEL should be equal to DATA_CHANNEL.

`w_scale` : `tensor(float)` — Scale tensor for input 'w'. It could be a scalar or a 1-D tensor, which means a per-tensor/layer or per output channel quantization. If it's a 1-D tensor, its number of elements should be equal to the number of output channels (M).

`w_zero_point` : T2 — Zero point tensor for input 'w'. It could be a scalar or a 1-D tensor, which means a per-tensor/layer or per output channel quantization. If it's a 1-D tensor, its number of elements should be equal to the number of output channels (M).

`y_scale` : `tensor(float)` — Scale tensor for output 'y'. It's a scalar, which means a per-tensor/layer quantization.

`y_zero_point` : T3 — Zero point tensor for output 'y'. It's a scalar, which means a per-tensor/layer quantization.

`B (optional)` : T4 — Optional 1D bias to be added to the convolution, has size of M. Bias must be quantized using $scale = x_scale * w_scale$ and $zero_point = 0$

Outputs `y` : T3 — Output data tensor that contains the result of the convolution. The output dimensions are functions of the kernel size, stride size, and pad lengths.

Attributes `auto_pad` : `string` (default is NOTSET) — `auto_pad` must be either NOTSET, SAME_UPPER, SAME_LOWER or VALID. Where default value is NOTSET, which means explicit padding is used. SAME_UPPER or SAME_LOWER mean pad the input so that $output_shape[i] = \lceil input_shape[i] / strides[i] \rceil$ for each axis *i*. The padding is split between the two sides equally or almost equally (depending on whether it is even or odd). In case the padding is an odd number, the extra padding is added at the end for SAME_UPPER and at the beginning for SAME_LOWER.

`dilations` : `list of ints` — dilation value along each spatial axis of the filter. If not present, the dilation defaults to 1 along each spatial axis.

`group` : `int` (default is 1) — number of groups input channels and output channels are divided into. default is 1.

`kernel_shape` : `list of ints` — The shape of the convolution kernel. If not present, should be inferred from input 'w'.

`pads` : `list of ints` — Padding for the beginning and ending along each spatial axis, it can take any value greater than or equal to 0. The value represent the number of pixels added to the beginning and end part of the corresponding axis. pads format should be as follow [x1_begin, x2_begin...x1_end, x2_end,...], where xi_begin the number of pixels added at the beginning of axis *i* and xi_end, the number of pixels added at the end of axis *i*. This attribute cannot be used simultaneously with `auto_pad` attribute. If not present, the padding defaultsto 0 along start and end of each spatial axis.

`strides` : `list of ints` — Stride along each spatial axis. If not present, the stride defaults to 1 along each spatial axis.

Type	T1 : tensor(int8), tensor(uint8) — Constrain input type to 8-bit integer tensor.
constraints	T2 : tensor(int8), tensor(uint8) — Constrain filter type to 8-bit integer tensor.
	T3 : tensor(int8), tensor(uint8) — Constrain output type to 8-bit integer tensor.
	T4 : tensor(int32) — Constrain bias type to 32-bit integer tensor.

Test vectors	test_qlinearconv
--------------	------------------

Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.
-------------------------------------	--

5.126. QLinearMatMul

Matrix product that behaves like [numpy.matmul](#). It consumes two quantized input tensors, their scales and zero points, scale and zero point of output, and computes the quantized output. The quantization formula is $y = \text{satuate}((x / y_scale) + y_zero_point)$. For (x / y_scale) , it is rounding to nearest ties to even. Refer to <https://en.wikipedia.org/wiki/Rounding> for details. Scale and zero point must have same shape. They must be either scalar (per tensor) or N-D tensor (per row for 'a' and per column for 'b'). Scalar refers to per tensor quantization whereas N-D refers to per row or per column quantization. If the input is 2D of shape [M, K] then zero point and scale tensor may be an M element vector [v_1, v_2, ..., v_M] for per row quantization and K element vector of shape [v_1, v_2, ..., v_K] for per column quantization. If the input is N-D tensor with shape [D1, D2, M, K] then zero point and scale tensor may have shape [D1, D2, M, 1] for per row quantization and shape [D1, D2, 1, K] for per column quantization. Production must never overflow, and accumulation may overflow if and only if in 32 bits.

Domain	ai.onnx
Since version	21
Earlier versions	10
Inputs	a (non-differentiable) : T1 — N-dimensional quantized matrix a a_scale (non-differentiable) : TS — scale of quantized input a a_zero_point (non-differentiable) : T1 — zero point of quantized input a b (non-differentiable) : T2 — N-dimensional quantized matrix b b_scale (non-differentiable) : TS — scale of quantized input b b_zero_point (non-differentiable) : T2 — zero point of quantized input b y_scale (non-differentiable) : TS — scale of quantized output y y_zero_point (non-differentiable) : T3 — zero point of quantized output y
Outputs	y (non-differentiable) : T3 — Quantized matrix multiply results from $a * b$
Attributes	None.
Type constraints	TS : tensor(float), tensor(float16), tensor(bfloat16) — Constrain scales.

T1 : tensor(int8), tensor(uint8), tensor(float8e4m3fn),
 tensor(float8e4m3fnuz), tensor(float8e5m2),
 tensor(float8e5m2fnuz) — The type of input a and its zeropoint.
 T2 : tensor(int8), tensor(uint8), tensor(float8e4m3fn),
 tensor(float8e4m3fnuz), tensor(float8e5m2),
 tensor(float8e5m2fnuz) — The type of input b and its zeropoint.
 T3 : tensor(int8), tensor(uint8), tensor(float8e4m3fn),
 tensor(float8e4m3fnuz), tensor(float8e5m2),
 tensor(float8e5m2fnuz) — The type of the output and its zeropoint.

Test vectors None published for this operator.

Not supplied by
 the upstream
 source Shape inference, determinism, errors. See the editorial note at the
 head of this clause.

5.127. QuantizeLinear

The linear quantization operator consumes a high-precision tensor, a scale, and a zero point to compute the low-precision/quantized tensor. The scale factor and zero point must have the same shape, determining the quantization granularity. The quantization formula is $y = \text{satuate}((x / y_scale) + y_zero_point)$.

Saturation is done according to:

- uint16: [0, 65535]
- int16: [-32768, 32767]
- uint8: [0, 255]
- int8: [-128, 127]
- uint4: [0, 15]
- int4: [-8, 7]
- uint2: [0, 3]
- int2: [-2, 1]

For (x / y_scale) , it rounds to the nearest even. Refer to <https://en.wikipedia.org/wiki/Rounding> for details.

y_zero_point and y must have the same type. y_zero_point is usually not used for quantization to float8 and 4bit types, but the quantization formula remains the same for consistency, and the type of the attribute y_zero_point still determines the quantization type. x and y_scale are allowed to have different types. The type of y_scale determines the precision of the division operation between x and y_scale , unless the precision attribute is specified.

There are three supported quantization granularities, determined by the shape of y_scale . In all cases, y_zero_point must have the same shape as y_scale .

- Per-tensor (per-layer) quantization: y_scale is a scalar.
- Per-axis quantization: The scale must be a 1-D tensor, with the length of the quantization axis. For an input shape $(D_0, \dots, D_i, \dots, D_n)$ and $\text{axis}=i$, y_scale is a 1-D tensor of length D_i .
- Blocked quantization: The scale's shape is identical to the input's shape, except for one dimension, in which blocking is performed. Given x shape $(D_0, \dots, D_i, \dots, D_n)$, $\text{axis}=i$, and block size B : y_scale shape is $(D_0, \dots, \text{ceil}(D_i/B), \dots, D_n)$.

Domain ai.onnx

Since version 28

Earlier versions 10, 13, 19, 21, 23, 24, 25

versions

Inputs (2 — 3) x : T1 — N-D full precision Input tensor to be quantized.

	<code>y_scale</code> : T2 — Scale for doing quantization to get y. For per-tensor/layer quantization the scale is a scalar, for per-axis quantization it is a 1-D Tensor and for blocked quantization it has the same shape as the input, except for one dimension in which blocking is performed. <code>y_zero_point</code> (optional) : T3 — Zero point for doing quantization to get y. Shape must match <code>y_scale</code>. Default is uint8 with zero point of 0 if it's not specified.
Outputs	<code>y</code> : T3 — N-D quantized output tensor. It has same shape as input x.
Attributes	<code>axis</code> : int (default is 1) — (Optional) The axis of the dequantizing dimension of the input tensor. Used only for per-axis and blocked quantization. Negative value means counting dimensions from the back. Accepted range is $[-r, r-1]$ where $r = \text{rank}(\text{input})$. When the rank of the input is 1, per-tensor quantization is applied, rendering the axis unnecessary in this scenario. <code>block_size</code> : int (default is 0) — (Optional) The size of the quantization block (number of times every scale is replicated). Used only for blocked quantization. The block size is a positive integer. Given x shape $(D_0, \dots, D_i, \dots, D_n)$, <code>y_scale</code> shape $(S_0, \dots, S_i, \dots, S_n)$ and <code>axis=i</code>, the accepted range is $[\text{ceil}(D_i/S_i), \text{ceil}(D_i/(S_i-1))-1]$ <code>output_dtype</code> : int (default is 0) — (Optional) The output data type. If not supplied, the output data type is inferred from <code>y_zero_point</code> data type (T3). If neither <code>output_dtype</code> nor <code>y_zero_point</code> are supplied, output data type is uint8. If both <code>output_dtype</code> and <code>y_zero_point</code> are specified, <code>output_dtype</code> must be T3. <code>precision</code> : int (default is 0) — (Optional) The precision of the division operation between x and <code>y_scale</code>. If not provided, it will be the same as the type of <code>y_scale</code>. <code>saturate</code> : int (default is 1) — The parameter defines how the conversion behaves if an input value is out of range of the destination type. It only applies for float 8 quantization (float8e4m3fn, float8e4m3fnuz, float8e5m2, float8e5m2fnuz). It is true by default. All cases are fully described in two tables inserted in the operator description. It has no effect for float4e2m1, float6e2m3, or float6e3m2, since those types have no non-saturating (infinity-representable) encoding to fall back to.
Type constraints	T1 : tensor(float), tensor(float16), tensor(bfloat16), tensor(int32) — The type of the input 'x'. T2 : tensor(float), tensor(float16), tensor(bfloat16), tensor(int32), tensor(float8e8m0) — The type of the input 'y_scale'. T3 : tensor(int8), tensor(uint8), tensor(int16), tensor(uint16), tensor(float8e4m3fn), tensor(float8e4m3fnuz), tensor(float8e5m2), tensor(float8e5m2fnuz), tensor(uint4), tensor(int4), tensor(float4e2m1), tensor(uint2), tensor(int2), tensor(float6e2m3), tensor(float6e3m2) — The type of the input <code>y_zero_point</code> and the output y.
Test vectors	test_quantizelinear_axis, test_quantizelinear_blocked_asymmetric, test_quantizelinear_blocked_symmetric, test_quantizelinear_e4m3fn, test_quantizelinear_e5m2, test_quantizelinear_float4e2m1, test_quantizelinear_int16, test_quantizelinear_int2, test_quantizelinear_int4, test_quantizelinear, test_quantizelinear_uint16, test_quantizelinear_uint2, test_quantizelinear_uint4
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.128. RMSNormalization

This is RMS normalization defined in ONNX as function as described in the paper <https://arxiv.org/pdf/1910.07467>. The overall computation can be split into two stages. The root mean squared norm is taken over the last D dimensions, where D is the dimension of normalized_shape. For example, if normalized_shape is (3, 5) (a 2-dimensional shape), the rms norm is computed over the last 2 dimensions of the input. The computation required by standardization can be described by the following equations. $XSquared = Mul(X, X)$ $XSquaredMean = ReduceMean<axes=normalized_axes>(XSquared)$ $MeanSquareEpsilon = Add(XSquaredMean, epsilon)$ $RMS = Sqrt(MeanSquareEpsilon)$ $Normalized = Div(X, RMS)$ where normalized_axes is [axis, ..., rank of X - 1]. The variables RMS stand for root mean square, Depending on stash_type attribute, the actual computation must happen in different floating-point precision. For example, if stash_type is 1, this operator casts all input variables to 32-bit float, perform the computation, and finally cast Normalized back to the original type of X. The second stage then scales the outcome of the first stage using: $Y = Mul(Normalized, Scale)$ Let $d[i]$ indicate the i-th dimension of X. If X's shape is $[d[0], \dots, d[axis-1], d[axis], \dots, d[rank-1]]$, the shape of RMS is $[d[0], \dots, d[axis-1], 1, \dots, 1]$. Y and X have the same shape. This operator supports unidirectional broadcasting (Scale should be unidirectional broadcastable to tensor X); for more details please check the doc.

Domain	ai.onnx
Since version	23
Inputs	X : T — The input tensor to be normalized. In general, the shape is (D1, D2, ..., Dn) for n-dimensional data, where the root mean squared norm is taken over the last D dimensions, D is determined by the axis attribute. scale : V — Scale tensor. Scale tensor shape should be broadcastable to the normalized shape.
Outputs	Y : V — Output data tensor. Same shape as X
Attributes	axis : int (default is -1) — The first normalization dimension. If rank(X) is r, axis' allowed range is [-r, r). Negative value means counting dimensions from the back. epsilon : float (default is 1e-05) — The epsilon value to use to avoid division by zero. stash_type : int (default is 1) — The floating-point precision used in stage one of the computation.
Type constraints	T : tensor(float16), tensor(float), tensor(double), tensor(bfloat16) — Constrain input X type to float tensors. V : tensor(float16), tensor(float), tensor(double), tensor(bfloat16) — Constrain output Y and scale type to float tensors.
Test vectors	test_rms_normalization_default_axis
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.129. RNN

Computes an one-layer simple RNN. This operator is usually supported via some custom implementation such as CuDNN.

Notations:

- X — input tensor
- i — input gate
- t — time step ($t-1$ means previous time step)
- W_i — W parameter weight matrix for input gate
- R_i — R recurrence weight matrix for input gate
- W_{bi} — W parameter bias vector for input gate
- R_{bi} — R parameter bias vector for input gate
- W_{Bi} — W parameter weight matrix for backward input gate
- R_{Bi} — R recurrence weight matrix for backward input gate
- W_{Bbi} — W parameter bias vectors for backward input gate
- R_{Bbi} — R parameter bias vectors for backward input gate
- H — Hidden state
- num_directions — 2 if $\text{direction} == \text{bidirectional}$ else 1

Activation functions:

- $\text{Relu}(x)$ — $\max(0, x)$
- $\text{Tanh}(x)$ — $(1 - e^{-2x}) / (1 + e^{-2x})$
- $\text{Sigmoid}(x)$ — $1 / (1 + e^{-x})$

NOTE Below are optional

- $\text{Affine}(x)$ — $\alpha * x + \beta$
- $\text{LeakyRelu}(x)$ — x if $x \geq 0$ else $\alpha * x$
- $\text{ThresholdedRelu}(x)$ — x if $x \geq \alpha$ else 0
- $\text{ScaledTanh}(x)$ — $\alpha \text{Tanh}(\beta x)$
- $\text{HardSigmoid}(x)$ — $\min(\max(\alpha * x + \beta, 0), 1)$
- $\text{Elu}(x)$ — x if $x \geq 0$ else $\alpha * (e^x - 1)$

Table 7 — Table from the upstream description of RNN

- $\text{Softsign}(x)$ — $x / (1 + |x|)$
- $\text{Softplus}(x)$ — $\log(1 + e^x)$

Equations (Default: $f = \text{Tanh}$):

- $H_t = f(X_t(W_i^T) + H_{t-1}(R_i^T) + W_{bi} + R_{bi})$ This operator has **optional** inputs/outputs. See the doc for more details about the representation of optional arguments. An empty string may be used in the place of an actual argument's name to indicate a missing argument. Trailing optional arguments (those not followed by an argument that is present) may also be simply omitted.

Domain `ai.onnx`

Since version 22

Earlier 1, 7, 14

versions

Inputs (3 — 6) X (differentiable) : T — The input sequences packed (and potentially padded) into one 3-D tensor with the shape of $[\text{seq_length}, \text{batch_size}, \text{input_size}]$.

W (differentiable) : T — The weight tensor for input gate.

Concatenation of W_i and W_{Bi} (if bidirectional). The tensor has shape $[\text{num_directions}, \text{hidden_size}, \text{input_size}]$.

R (differentiable) : T — The recurrence weight tensor.

Concatenation of R_i and R_{Bi} (if bidirectional). The tensor has shape $[\text{num_directions}, \text{hidden_size}, \text{hidden_size}]$.

B (optional, differentiable) : T — The bias tensor for input gate.

Concatenation of $[W_{bi}, R_{bi}]$ and $[W_{Bbi}, R_{Bbi}]$ (if bidirectional). The tensor has shape $[\text{num_directions}, 2 * \text{hidden_size}]$. Optional: If not specified — assumed to be 0.

	sequence_lens (optional, non-differentiable) : T1 — Optional tensor specifying lengths of the sequences in a batch. If not specified — assumed all sequences in the batch to have length seq_length. It has shape [batch_size].
	initial_h (optional, non-differentiable) : T — Optional initial value of the hidden. If not specified — assumed to be 0. It has shape [num_directions, batch_size, hidden_size].
Outputs (0 — 2)	Y (optional, differentiable) : T — A tensor that concatenates all the intermediate output values of the hidden. It has shape [seq_length, num_directions, batch_size, hidden_size]. Y_h (optional, differentiable) : T — The last output value of the hidden. It has shape [num_directions, batch_size, hidden_size].
Attributes	activation_alpha : list of floats — Optional scaling values used by some activation functions. The values are consumed in the order of activation functions, for example (f, g, h) in LSTM. Default values are the same as of corresponding ONNX operators. For example with LeakyRelu, the default alpha is 0.01. activation_beta : list of floats — Optional scaling values used by some activation functions. The values are consumed in the order of activation functions, for example (f, g, h) in LSTM. Default values are the same as of corresponding ONNX operators. activations : list of strings (default is ['Tanh', 'Tanh']) — One (or two if bidirectional) activation function for input gate. The activation function must be one of the activation functions specified above. Optional: Default Tanh if not specified. clip : float — Cell clip threshold. Clipping bounds the elements of a tensor in the range of [-threshold, +threshold] and is applied to the input of activations. No clip if not specified. direction : string (default is forward) — Specify if the RNN is forward, reverse, or bidirectional. Must be one of forward (default), reverse, or bidirectional. hidden_size : int — Number of neurons in the hidden layer layout : int (default is 0) — The shape format of inputs X, initial_h and outputs Y, Y_h. If 0, the following shapes are expected: X.shape = [seq_length, batch_size, input_size], Y.shape = [seq_length, num_directions, batch_size, hidden_size], initial_h.shape = Y_h.shape = [num_directions, batch_size, hidden_size]. If 1, the following shapes are expected: X.shape = [batch_size, seq_length, input_size], Y.shape = [batch_size, seq_length, num_directions, hidden_size], initial_h.shape = Y_h.shape = [batch_size, num_directions, hidden_size].
Type constraints	T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain input and output types to float tensors. T1 : tensor(int32) — Constrain seq_lens to integer tensor.
Test vectors	test_simple_rnn_batchwise, test_simple_rnn_bidirectional, test_simple_rnn_defaults, test_simple_rnn_with_initial_bias, test_simple_rnn_reverse, test_rnn_seq_length
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.130. RandomNormal

Generate a tensor with random values drawn from a normal distribution. The shape of the tensor is specified by the shape argument and the parameter of the normal distribution specified by mean and scale.

The data type is specified by the 'dtype' argument. The 'dtype' argument must be one of the data types specified in the 'DataType' enum field in the TensorProto message.

Domain	ai.onnx
Since version	22
Earlier versions	1
Inputs	None.
Outputs	output : T — Output tensor of random values drawn from normal distribution
Attributes	dtype : int (default is 1) — The data type for the elements of the output tensor. Default is TensorProto::FLOAT. mean : float (default is 0.0) — The mean of the normal distribution. scale : float (default is 1.0) — The standard deviation of the normal distribution. seed : float — (Optional) Seed to the random generator, if not specified we will auto generate one. shape : list of ints (required) — The shape of the output tensor.
Type constraints	T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain output types to float tensors.
Test vectors	None published for this operator.
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.131. RandomNormalLike

Generate a tensor with random values drawn from a normal distribution. The shape of the output tensor is copied from the shape of the input tensor, and the parameters of the normal distribution are specified by mean and scale.

The data type is specified by the 'dtype' argument, or copied from the input tensor if not provided. The 'dtype' argument must be one of the data types specified in the 'DataType' enum field in the TensorProto message, and be valid as an output type.

Domain	ai.onnx
Since version	22
Earlier versions	1
Inputs	input : T1 — Input tensor to copy shape and optionally type information from.
Outputs	output : T2 — Output tensor of random values drawn from normal distribution
Attributes	dtype : int — (Optional) The data type for the elements of the output tensor, if not specified, we will use the data type of the input tensor.

	mean : float (default is 0.0) — The mean of the normal distribution. scale : float (default is 1.0) — The standard deviation of the normal distribution. seed : float — (Optional) Seed to the random generator, if not specified we will auto generate one.
Type constraints	T1 : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128) — Constrain to any tensor type. If the dtype attribute is not provided this must be a valid output type. T2 : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain output types to float tensors.
Test vectors	None published for this operator.
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.132. RandomUniform

Generate a tensor with random values drawn from a uniform distribution. The shape of the tensor is specified by the shape argument and the range by low and high.

The data type is specified by the 'dtype' argument. The 'dtype' argument must be one of the data types specified in the 'DataType' enum field in the TensorProto message.

Domain	ai.onnx
Since version	22
Earlier versions	1
Inputs	None.
Outputs	output : T — Output tensor of random values drawn from uniform distribution
Attributes	dtype : int (default is 1) — The data type for the elements of the output tensor. If not specified, default is TensorProto: :FLOAT. high : float (default is 1.0) — Upper boundary of the output values. low : float (default is 0.0) — Lower boundary of the output values. seed : float — (Optional) Seed to the random generator, if not specified we will auto generate one. shape : list of ints (required) — The shape of the output tensor.
Type constraints	T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain output types to float tensors.

Test vectors	None published for this operator.
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.133. RandomUniformLike

Generate a tensor with random values drawn from a uniform distribution. The shape of the output tensor is copied from the shape of the input tensor, and the parameters of the uniform distribution are specified by low and high.

The data type is specified by the 'dtype' argument, or copied from the input tensor if not provided. The 'dtype' argument must be one of the data types specified in the 'DataType' enum field in the TensorProto message and be valid as an output type.

Domain	ai.onnx
Since version	22
Earlier versions	1
Inputs	input : T1 — Input tensor to copy shape and optionally type information from.
Outputs	output : T2 — Output tensor of random values drawn from uniform distribution
Attributes	dtype : int — (Optional) The data type for the elements of the output tensor, if not specified, we will use the data type of the input tensor. high : float (default is 1.0) — Upper boundary of the output values. low : float (default is 0.0) — Lower boundary of the output values. seed : float — (Optional) Seed to the random generator, if not specified we will auto generate one.
Type constraints	T1 : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128) — Constrain to any tensor type. If the dtype attribute is not provided this must be a valid output type. T2 : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain output types to float tensors.
Test vectors	None published for this operator.
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.134. Range

Generate a tensor containing a sequence of numbers that begin at start and extends by increments of delta up to limit (exclusive).

The number of elements in the output of range is computed as below:

```
number_of_elements = max( ceil( (limit - start) / delta ) , 0 )
```

The pseudocode determining the contents of the output is shown below:

```
for(int i=0; i<number_of_elements; ++i) {
    output[i] = start + (i * delta);
}
```

Example 1:

Inputs: start = 3, limit = 9, delta = 3

Output: [3, 6]

Example 2:

Inputs: start = 10, limit = 4, delta = -2

Output: [10, 8, 6]

For float16 and bfloat16 inputs, the `stash_type` attribute controls the precision used for intermediate accumulation. Setting `stash_type` to 1 (float) causes `start`, `limit`, and `delta` to be cast to 32-bit float before the loop, with the output cast back to the original type. This avoids precision loss for large ranges where successive additions in float16 or bfloat16 would otherwise be inexact (e.g. $x + 1 == x$ for large x).

Domain	ai.onnx
Since version	27
Earlier versions	11
Inputs	<code>start</code> : T — Scalar. First entry for the range of output values. <code>limit</code> : T — Scalar. Exclusive upper limit for the range of output values. <code>delta</code> : T — Scalar. Value to step by.
Outputs	<code>output</code> : T — A 1-D tensor with same type as the inputs containing generated range of values.
Attributes	<code>stash_type</code> : int (default is 1) — The data type used for intermediate computation when T is float16 or bfloat16. Defaults to 1 (float). Has no effect for other types.
Type constraints	T : tensor(float), tensor(double), tensor(int16), tensor(int32), tensor(int64), tensor(float16), tensor(bfloat16) — Constrain input types to common numeric type tensors.
Test vectors	test_range_bfloat16_type_positive_delta, test_range_float16_type_positive_delta, test_range_float_type_positive_delta, test_range_int32_type_negative_delta
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.135. Reciprocal

Reciprocal takes one input data (Tensor<T>) and produces one output data (Tensor<T>) where the reciprocal is, $y = 1/x$, is applied to the tensor elementwise.

Domain	ai.onnx
Since version	13
Earlier versions	1, 6
Inputs	X (differentiable) : T—Input tensor
Outputs	Y (differentiable) : T—Output tensor
Attributes	None.
Type constraints	T : tensor(float16), tensor(float), tensor(double), tensor(bfloat16) — Constrain input and output types to float tensors.
Test vectors	test_reciprocal_example, test_reciprocal
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.136. ReduceL1

Computes the L1 norm of the input tensor's elements along the provided axes. The resulting tensor has the same rank as the input if keepdims equals 1. If keepdims equals 0, then the resulting tensor has the reduced dimension pruned. Input tensors of rank zero are valid. Reduction over an empty set of values yields 0.

The above behavior is similar to numpy, with the exception that numpy defaults keepdims to False instead of True.

Domain	ai.onnx
Since version	18
Earlier versions	1, 11, 13
Inputs (1 — 2)	data (differentiable) : T—An input tensor. axes (optional, non-differentiable) : tensor(int64) — Optional input list of integers, along which to reduce. The default is to reduce over empty axes. When axes is empty (either not provided or explicitly empty), behavior depends on 'noop_with_empty_axes': reduction over all axes if 'noop_with_empty_axes' is false, and reduction over the empty set of axes when 'noop_with_empty_axes' is true. Accepted range is [-r, r-1] where r = rank(data).
Outputs	reduced (differentiable) : T—Reduced output tensor.
Attributes	keepdims : int (default is 1) — Keep the reduced dimension or not, default 1 means keep reduced dimension. noop_with_empty_axes : int (default is 0) — Defines behavior when axes is not provided or is empty. If false (default), reduction happens

over all axes (similar to the case when `axis=None` in numpy). If true, reduction happens over an empty set of axes (similar to the case when `axis=()` in numpy). Note that reduction over an empty set of axes means that the reduction step behaves like a no-op (identity function), but composite-reduction operators will still perform the non-reduction steps as needed. Thus, `ReduceLogSum` returns the Log of input tensor, and `ReduceSumSquare` returns the Square of the input tensor, in this case.

Type constraints	<code>T : tensor(uint32), tensor(uint64), tensor(int32), tensor(int64), tensor(float16), tensor(float), tensor(double), tensor(bfloat16)</code> — Constrain input and output types to numeric tensors.
Test vectors	<code>test_reduce_l1_default_axes_keepdims_example, test_reduce_l1_default_axes_keepdims_random, test_reduce_l1_do_not_keepdims_example, test_reduce_l1_do_not_keepdims_random, test_reduce_l1_empty_set, test_reduce_l1_keep_dims_example, test_reduce_l1_keep_dims_random, test_reduce_l1_negative_axes_keep_dims_example, test_reduce_l1_negative_axes_keep_dims_random</code>
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.137. ReduceL2

Computes the L2 norm of the input tensor's elements along the provided axes. The resulting tensor has the same rank as the input if `keepdims` equals 1. If `keepdims` equals 0, then the resulting tensor has the reduced dimension pruned. Input tensors of rank zero are valid. Reduction over an empty set of values yields 0.

The above behavior is similar to numpy, with the exception that numpy defaults `keepdims` to False instead of True.

Domain	<code>ai.onnx</code>
Since version	18
Earlier versions	1, 11, 13
Inputs (1 — 2)	<code>data</code> (differentiable) : <code>T</code> — An input tensor. <code>axes</code> (optional, non-differentiable) : <code>tensor(int64)</code> — Optional input list of integers, along which to reduce. The default is to reduce over empty axes. When <code>axes</code> is empty (either not provided or explicitly empty), behavior depends on 'noop_with_empty_axes': reduction over all axes if 'noop_with_empty_axes' is false, and reduction over the empty set of axes when 'noop_with_empty_axes' is true. Accepted range is <code>[-r, r-1]</code> where <code>r = rank(data)</code> .
Outputs	<code>reduced</code> (differentiable) : <code>T</code> — Reduced output tensor.
Attributes	<code>keepdims</code> : <code>int</code> (default is 1) — Keep the reduced dimension or not, default 1 means keep reduced dimension. <code>noop_with_empty_axes</code> : <code>int</code> (default is 0) — Defines behavior when <code>axes</code> is not provided or is empty. If false (default), reduction happens over all axes (similar to the case when <code>axis=None</code> in numpy). If true, reduction happens over an empty set of axes (similar to the case when <code>axis=()</code> in numpy). Note that reduction over an empty set of axes means

that the reduction step behaves like a no-op (identity function), but composite-reduction operators will still perform the non-reduction steps as needed. Thus, `ReduceLogSum` returns the Log of input tensor, and `ReduceSumSquare` returns the Square of the input tensor, in this case.

Type constraints	<code>T : tensor(uint32), tensor(uint64), tensor(int32), tensor(int64), tensor(float16), tensor(float), tensor(double), tensor(bfloat16)</code> — Constrain input and output types to numeric tensors.
Test vectors	<code>test_reduce_l2_default_axes_keepdims_example</code> , <code>test_reduce_l2_default_axes_keepdims_random</code> , <code>test_reduce_l2_do_not_keepdims_example</code> , <code>test_reduce_l2_do_not_keepdims_random</code> , <code>test_reduce_l2_empty_set</code> , <code>test_reduce_l2_keep_dims_example</code> , <code>test_reduce_l2_keep_dims_random</code> , <code>test_reduce_l2_negative_axes_keep_dims_example</code> , <code>test_reduce_l2_negative_axes_keep_dims_random</code>
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.138. ReduceLogSum

Computes the log sum of the input tensor's elements along the provided axes. The resulting tensor has the same rank as the input if `keepdims` equals 1. If `keepdims` equals 0, then the resulting tensor has the reduced dimension pruned. Input tensors of rank zero are valid. Reduction over an empty set of values yields minus infinity (if supported by the datatype) or undefined otherwise.

The above behavior is similar to numpy, with the exception that numpy defaults `keepdims` to `False` instead of `True`.

Domain	<code>ai.onnx</code>
Since version	28
Earlier versions	1, 11, 13, 18
Inputs (1 — 2)	<code>data</code> (differentiable) : <code>T</code> — An input tensor. <code>axes</code> (optional, non-differentiable) : <code>tensor(int64)</code> — Optional input list of integers, along which to reduce. The default is to reduce over empty axes. When <code>axes</code> is empty (either not provided or explicitly empty), behavior depends on 'noop_with_empty_axes': reduction over all axes if 'noop_with_empty_axes' is false, and reduction over the empty set of axes when 'noop_with_empty_axes' is true. Accepted range is <code>[-r, r-1]</code> where <code>r = rank(data)</code> .
Outputs	<code>reduced</code> (differentiable) : <code>T</code> — Reduced output tensor.
Attributes	<code>keepdims</code> : <code>int</code> (default is 1) — Keep the reduced dimension or not, default 1 means keep reduced dimension. <code>noop_with_empty_axes</code> : <code>int</code> (default is 0) — Defines behavior when <code>axes</code> is not provided or is empty. If false (default), reduction happens over all axes (similar to the case when <code>axis=None</code> in numpy). If true, reduction happens over an empty set of axes (similar to the case when <code>axis=()</code> in numpy). Note that reduction over an empty set of axes means that the reduction step behaves like a no-op (identity function), but composite-reduction operators will still perform the non-reduction steps

as needed. Thus, ReduceLogSum returns the Log of input tensor, and ReduceSumSquare returns the Square of the input tensor, in this case.

Type constraints	<code>T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double)</code> — Constrain input and output types to float tensors.
Test vectors	<code>test_reduce_log_sum_empty_set, test_reduce_log_sum_default, test_reduce_log_sum_negative_axes, test_reduce_log_sum_desc_axes, test_reduce_log_sum_asc_axes</code>
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.139. ReduceLogSumExp

Computes the log sum exponent of the input tensor's elements along the provided axes. The resulting tensor has the same rank as the input if keepdims equals 1. If keepdims equals 0, then the resulting tensor has the reduced dimension pruned. Input tensors of rank zero are valid. Reduction over an empty set of values yields minus infinity (if supported by the datatype) or undefined otherwise.

The above behavior is similar to numpy, with the exception that numpy defaults keepdims to False instead of True.

Domain	<code>ai.onnx</code>
Since version	28
Earlier versions	1, 11, 13, 18
Inputs (1 — 2)	<code>data</code> (differentiable) : <code>T</code> — An input tensor. <code>axes</code> (optional, non-differentiable) : <code>tensor(int64)</code> — Optional input list of integers, along which to reduce. The default is to reduce over empty axes. When <code>axes</code> is empty (either not provided or explicitly empty), behavior depends on 'noop_with_empty_axes': reduction over all axes if 'noop_with_empty_axes' is false, and reduction over the empty set of axes when 'noop_with_empty_axes' is true. Accepted range is <code>[-r, r-1]</code> where <code>r = rank(data)</code> .
Outputs	<code>reduced</code> (differentiable) : <code>T</code> — Reduced output tensor.
Attributes	<code>keepdims</code> : <code>int</code> (default is 1) — Keep the reduced dimension or not, default 1 means keep reduced dimension. <code>noop_with_empty_axes</code> : <code>int</code> (default is 0) — Defines behavior when <code>axes</code> is not provided or is empty. If false (default), reduction happens over all axes (similar to the case when <code>axis=None</code> in numpy). If true, reduction happens over an empty set of axes (similar to the case when <code>axis=()</code> in numpy). Note that reduction over an empty set of axes means that the reduction step behaves like a no-op (identity function), but composite-reduction operators will still perform the non-reduction steps as needed. Thus, ReduceLogSum returns the Log of input tensor, and ReduceSumSquare returns the Square of the input tensor, in this case.
Type constraints	<code>T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double)</code> — Constrain input and output types to float tensors.

Test vectors `test_reduce_log_sum_exp_default_axes_keepdims_example`, `test_reduce_log_sum_exp_default_axes_keepdims_random`, `test_reduce_log_sum_exp_do_not_keepdims_example`, `test_reduce_log_sum_exp_do_not_keepdims_random`, `test_reduce_log_sum_exp_empty_set`, `test_reduce_log_sum_exp_keepdims_example`, `test_reduce_log_sum_exp_keepdims_random`, `test_reduce_log_sum_exp_negative_axes_keepdims_example`, `test_reduce_log_sum_exp_negative_axes_keepdims_random`

Not supplied by the upstream source Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.140. ReduceMax

Computes the max of the input tensor's elements along the provided axes. The resulting tensor has the same rank as the input if `keepdims` equals 1. If `keepdims` equals 0, then the resulting tensor has the reduced dimension pruned. Input tensors of rank zero are valid. Reduction over an empty set of values yields minus infinity (if supported by the datatype) or the minimum value of the data type otherwise.

If the input data type is Boolean, the comparison should consider `False < True`.

The above behavior is similar to numpy, with the exception that numpy defaults `keepdims` to `False` instead of `True`.

Domain `ai.onnx`

Since version 20

Earlier versions 1, 11, 12, 13, 18

Inputs (1 — 2) `data` (differentiable) : `T` — An input tensor.
 `axes` (optional, non-differentiable) : `tensor(int64)` — Optional input list of integers, along which to reduce. The default is to reduce over empty axes. When `axes` is empty (either not provided or explicitly empty), behavior depends on '`noop_with_empty_axes`': reduction over all axes if '`noop_with_empty_axes`' is false, and reduction over the empty set of axes when '`noop_with_empty_axes`' is true. Accepted range is `[-r, r-1]` where `r = rank(data)`.

Outputs `reduced` (differentiable) : `T` — Reduced output tensor.

Attributes `keepdims` : `int` (default is 1) — Keep the reduced dimension or not, default 1 means keep reduced dimension.
 `noop_with_empty_axes` : `int` (default is 0) — Defines behavior when `axes` is not provided or is empty. If false (default), reduction happens over all axes (similar to the case when `axis=None` in numpy). If true, reduction happens over an empty set of axes (similar to the case when `axis=()` in numpy). Note that reduction over an empty set of axes means that the reduction step behaves like a no-op (identity function), but composite-reduction operators will still perform the non-reduction steps as needed. Thus, `ReduceLogSum` returns the Log of input tensor, and `ReduceSumSquare` returns the Square of the input tensor, in this case.

Type constraints `T` : `tensor(uint32)`, `tensor(uint64)`, `tensor(int32)`, `tensor(int64)`, `tensor(float16)`, `tensor(float)`, `tensor(double)`, `tensor(bfloat16)`, `tensor(uint8)`,

`tensor(int8)`, `tensor(bool)` — Constrain input and output types to numeric and Boolean tensors.

Test vectors `test_reduce_max_bool_inputs`, `test_reduce_max_default_axes_keepdim_example`, `test_reduce_max_default_axes_keepdims_random`, `test_reduce_max_do_not_keepdims_example`, `test_reduce_max_do_not_keepdims_random`, `test_reduce_max_empty_set`, `test_reduce_max_empty_set_bool`, `test_reduce_max_keepdims_example`, `test_reduce_max_keepdims_random`, `test_reduce_max_negative_axes_keepdims_example`, `test_reduce_max_negative_axes_keepdims_random`

Not supplied by the upstream source Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.141. ReduceMean

Computes the mean of the input tensor's elements along the provided axes. The resulting tensor has the same rank as the input if `keepdims` equals 1. If `keepdims` equals 0, then the resulting tensor has the reduced dimension pruned. Input tensors of rank zero are valid. Reduction over an empty set of values yields undefined.

The above behavior is similar to numpy, with the exception that numpy defaults `keepdims` to False instead of True.

Domain `ai.onnx`

Since version 18

Earlier versions 1, 11, 13

Inputs (1 — 2) `data` (differentiable) : T — An input tensor.
`axes` (optional, non-differentiable) : `tensor(int64)` — Optional input list of integers, along which to reduce. The default is to reduce over empty axes. When `axes` is empty (either not provided or explicitly empty), behavior depends on 'noop_with_empty_axes': reduction over all axes if 'noop_with_empty_axes' is false, and reduction over the empty set of axes when 'noop_with_empty_axes' is true. Accepted range is `[-r, r-1]` where `r = rank(data)`.

Outputs `reduced` (differentiable) : T — Reduced output tensor.

Attributes `keepdims` : int (default is 1) — Keep the reduced dimension or not, default 1 means keep reduced dimension.
`noop_with_empty_axes` : int (default is 0) — Defines behavior when `axes` is not provided or is empty. If false (default), reduction happens over all axes (similar to the case when `axis=None` in numpy). If true, reduction happens over an empty set of axes (similar to the case when `axis=()` in numpy). Note that reduction over an empty set of axes means that the reduction step behaves like a no-op (identity function), but composite-reduction operators will still perform the non-reduction steps as needed. Thus, `ReduceLogSum` returns the Log of input tensor, and `ReduceSumSquare` returns the Square of the input tensor, in this case.

Type constraints T : `tensor(uint32)`, `tensor(uint64)`, `tensor(int32)`, `tensor(int64)`, `tensor(float16)`, `tensor(float)`,

`tensor(double)`, `tensor(bfloat16)` — Constrain input and output types to numeric tensors.

Test vectors `test_reduce_mean_default_axes_keepdims_example`, `test_reduce_mean_default_axes_keepdims_random`, `test_reduce_mean_do_not_keepdims_example`, `test_reduce_mean_do_not_keepdims_random`, `test_reduce_mean_keepdims_example`, `test_reduce_mean_keepdims_random`, `test_reduce_mean_negative_axes_keepdims_example`, `test_reduce_mean_negative_axes_keepdims_random`

Not supplied by the upstream source Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.142. ReduceMin

Computes the min of the input tensor's elements along the provided axes. The resulting tensor has the same rank as the input if `keepdims` equals 1. If `keepdims` equals 0, then the resulting tensor has the reduced dimension pruned. Input tensors of rank zero are valid. Reduction over an empty set of values yields plus infinity (if supported by the datatype) or the maximum value of the data type otherwise.

If the input data type is Boolean, the comparison should consider `False < True`.

The above behavior is similar to numpy, with the exception that numpy defaults `keepdims` to `False` instead of `True`.

Domain `ai.onnx`

Since version 20

Earlier versions 1, 11, 12, 13, 18

Inputs (1 — 2) `data` (differentiable) : `T` — An input tensor.
`axes` (optional, non-differentiable) : `tensor(int64)` — Optional input list of integers, along which to reduce. The default is to reduce over empty axes. When `axes` is empty (either not provided or explicitly empty), behavior depends on `'noop_with_empty_axes'`: reduction over all axes if `'noop_with_empty_axes'` is false, and reduction over the empty set of axes when `'noop_with_empty_axes'` is true. Accepted range is `[-r, r-1]` where `r = rank(data)`.

Outputs `reduced` (differentiable) : `T` — Reduced output tensor.

Attributes `keepdims` : `int` (default is 1) — Keep the reduced dimension or not, default 1 means keep reduced dimension.
`noop_with_empty_axes` : `int` (default is 0) — Defines behavior when `axes` is not provided or is empty. If false (default), reduction happens over all axes (similar to the case when `axis=None` in numpy). If true, reduction happens over an empty set of axes (similar to the case when `axis=()` in numpy). Note that reduction over an empty set of axes means that the reduction step behaves like a no-op (identity function), but composite-reduction operators will still perform the non-reduction steps as needed. Thus, `ReduceLogSum` returns the Log of input tensor, and `ReduceSumSquare` returns the Square of the input tensor, in this case.

Type constraints `T` : `tensor(uint32)`, `tensor(uint64)`, `tensor(int32)`, `tensor(int64)`, `tensor(float16)`, `tensor(float)`,

tensor(double), tensor(bfloat16), tensor(uint8),
 tensor(int8), tensor(bool) — Constrain input and output types to
 numeric and Boolean tensors.

Test vectors test_reduce_min_bool_inputs, test_reduce_min_default_axes_
 keepdims_example, test_reduce_min_default_axes_keepdims_
 random, test_reduce_min_do_not_keepdims_example, test_reduce_
 min_do_not_keepdims_random, test_reduce_min_empty_set, test_
 reduce_min_keepdims_example, test_reduce_min_keepdims_random,
 test_reduce_min_negative_axes_keepdims_example, test_reduce_
 min_negative_axes_keepdims_random

Not supplied Shape inference, determinism, errors. See the editorial note at the head of
 by the this clause.
 upstream
 source

5.143. ReduceProd

Computes the product of the input tensor's elements along the provided axes. The resulting tensor has the same rank as the input if keepdims equals 1. If keepdims equals 0, then the resulting tensor has the reduced dimension pruned. Input tensors of rank zero are valid. Reduction over an empty set of values yields 1.

The above behavior is similar to numpy, with the exception that numpy defaults keepdims to False instead of True.

Domain ai.onnx

Since version 18

Earlier 1, 11, 13
 versions

Inputs (1 — data (differentiable) : T — An input tensor.
 2) axes (optional, non-differentiable) : tensor(int64) — Optional
 input list of integers, along which to reduce. The default is to reduce over
 empty axes. When axes is empty (either not provided or explicitly empty),
 behavior depends on 'noop_with_empty_axes': reduction over all axes if
 'noop_with_empty_axes' is false, and reduction over the empty set of axes
 when 'noop_with_empty_axes' is true. Accepted range is [-r, r-1] where r =
 rank(data).

Outputs reduced (differentiable) : T — Reduced output tensor.

Attributes keepdims : int (default is 1) — Keep the reduced dimension or not,
 default 1 means keep reduced dimension.
 noop_with_empty_axes : int (default is 0) — Defines behavior
 when axes is not provided or is empty. If false (default), reduction happens
 over all axes (similar to the case when axis=None in numpy). If true,
 reduction happens over an empty set of axes (similar to the case when
 axis=() in numpy). Note that reduction over an empty set of axes means
 that the reduction step behaves like a no-op (identity function), but
 composite-reduction operators will still perform the non-reduction steps
 as needed. Thus, ReduceLogSum returns the Log of input tensor, and
 ReduceSumSquare returns the Square of the input tensor, in this case.

Type T : tensor(uint32), tensor(uint64), tensor(int32),
 constraints tensor(int64), tensor(float16), tensor(float),

tensor(double), tensor(bfloat16) — Constrain input and output types to numeric tensors.

Test vectors test_reduce_prod_default_axes_keepdims_example, test_reduce_prod_default_axes_keepdims_random, test_reduce_prod_do_not_keepdims_example, test_reduce_prod_do_not_keepdims_random, test_reduce_prod_empty_set, test_reduce_prod_keepdims_example, test_reduce_prod_keepdims_random, test_reduce_prod_negative_axes_keepdims_example, test_reduce_prod_negative_axes_keepdims_random

Not supplied by the upstream source Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.144. ReduceSum

Computes the sum of the input tensor's elements along the provided axes. The resulting tensor has the same rank as the input if keepdims equals 1. If keepdims equals 0, then the resulting tensor has the reduced dimension pruned. Input tensors of rank zero are valid. Reduction over an empty set of values yields 0.

The above behavior is similar to numpy, with the exception that numpy defaults keepdims to False instead of True.

Domain ai.onnx

Since version 13

Earlier versions 1, 11

Inputs (1 — 2) data (differentiable) : T — An input tensor.
axes (optional, non-differentiable) : tensor(int64) — Optional input list of integers, along which to reduce. The default is to reduce over empty axes. When axes is empty (either not provided or explicitly empty), behavior depends on 'noop_with_empty_axes': reduction over all axes if 'noop_with_empty_axes' is false, and reduction over the empty set of axes when 'noop_with_empty_axes' is true. Accepted range is [-r, r-1] where r = rank(data).

Outputs reduced (differentiable) : T — Reduced output tensor.

Attributes keepdims : int (default is 1) — Keep the reduced dimension or not, default 1 means keep reduced dimension.
noop_with_empty_axes : int (default is 0) — Defines behavior when axes is not provided or is empty. If false (default), reduction happens over all axes (similar to the case when axis=None in numpy). If true, reduction happens over an empty set of axes (similar to the case when axis=() in numpy). Note that reduction over an empty set of axes means that the reduction step behaves like a no-op (identity function), but composite-reduction operators will still perform the non-reduction steps as needed. Thus, ReduceLogSum returns the Log of input tensor, and ReduceSumSquare returns the Square of the input tensor, in this case.

Type constraints T : tensor(uint32), tensor(uint64), tensor(int32), tensor(int64), tensor(float16), tensor(float),

`tensor(double)`, `tensor(bfloat16)` — Constrain input and output types to numeric tensors.

Test vectors `test_reduce_sum_default_axes_keepdims_example`, `test_reduce_sum_default_axes_keepdims_random`, `test_reduce_sum_do_not_keepdims_example`, `test_reduce_sum_do_not_keepdims_random`, `test_reduce_sum_empty_axes_input_noop_example`, `test_reduce_sum_empty_axes_input_noop`, `test_reduce_sum_empty_set`, `test_reduce_sum_keepdims_example`, `test_reduce_sum_keepdims_random`, `test_reduce_sum_negative_axes_keepdims_example`, `test_reduce_sum_negative_axes_keepdims_random`, `test_reduce_sum_empty_set_non_reduced_axis_zero`

Not supplied by the upstream source Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.145. ReduceSumSquare

Computes the sum square of the input tensor's elements along the provided axes. The resulting tensor has the same rank as the input if `keepdims` equals 1. If `keepdims` equals 0, then the resulting tensor has the reduced dimension pruned. Input tensors of rank zero are valid. Reduction over an empty set of values yields 0.

The above behavior is similar to `numpy`, with the exception that `numpy` defaults `keepdims` to `False` instead of `True`.

Domain `ai.onnx`

Since version 18

Earlier versions 1, 11, 13

Inputs (1 — 2) `data` (differentiable) : `T` — An input tensor.
`axes` (optional, non-differentiable) : `tensor(int64)` — Optional input list of integers, along which to reduce. The default is to reduce over empty axes. When `axes` is empty (either not provided or explicitly empty), behavior depends on `'noop_with_empty_axes'`: reduction over all axes if `'noop_with_empty_axes'` is false, and reduction over the empty set of axes when `'noop_with_empty_axes'` is true. Accepted range is `[-r, r-1]` where `r = rank(data)`.

Outputs `reduced` (differentiable) : `T` — Reduced output tensor.

Attributes `keepdims` : `int` (default is 1) — Keep the reduced dimension or not, default 1 means keep reduced dimension.
`noop_with_empty_axes` : `int` (default is 0) — Defines behavior when `axes` is not provided or is empty. If false (default), reduction happens over all axes (similar to the case when `axis=None` in `numpy`). If true, reduction happens over an empty set of axes (similar to the case when `axis=()` in `numpy`). Note that reduction over an empty set of axes means that the reduction step behaves like a no-op (identity function), but composite-reduction operators will still perform the non-reduction steps as needed. Thus, `ReduceLogSum` returns the Log of input tensor, and `ReduceSumSquare` returns the Square of the input tensor, in this case.

Type constraints `T` : `tensor(uint32)`, `tensor(uint64)`, `tensor(int32)`, `tensor(int64)`, `tensor(float16)`, `tensor(float)`,

`tensor(double)`, `tensor(bfloat16)` — Constrain input and output types to numeric tensors.

Test vectors `test_reduce_sum_square_default_axes_keepdims_example`, `test_reduce_sum_square_default_axes_keepdims_random`, `test_reduce_sum_square_do_not_keepdims_example`, `test_reduce_sum_square_do_not_keepdims_random`, `test_reduce_sum_square_empty_set`, `test_reduce_sum_square_keepdims_example`, `test_reduce_sum_square_keepdims_random`, `test_reduce_sum_square_negative_axes_keepdims_example`, `test_reduce_sum_square_negative_axes_keepdims_random`

Not supplied by the upstream source Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.146. RegexFullMatch

RegexFullMatch performs a full regex match on each element of the input tensor. If an element fully matches the regex pattern specified as an attribute, the corresponding element in the output is True and it is False otherwise. [RE2](#) regex syntax is used.

Domain	<code>ai.onnx</code>
Since version	20
Inputs	<code>X (non-differentiable) : T1</code> —Tensor with strings to match on.
Outputs	<code>Y (non-differentiable) : T2</code> —Tensor of bools indicating if each input string fully matches the regex pattern specified.
Attributes	<code>pattern : string</code> —Regex pattern to match on. This must be valid RE2 syntax.
Type constraints	<code>T1 : tensor(string)</code> — Inputs must be UTF-8 strings <code>T2 : tensor(bool)</code> — Outputs are bools and are True where there is a full regex match and False otherwise.
Test vectors	<code>test_regex_full_match_basic</code> , <code>test_regex_full_match_email_domain</code> , <code>test_regex_full_match_empty</code>
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.147. Relu

Relu takes one input data (`Tensor<T>`) and produces one output data (`Tensor<T>`) where the rectified linear function, $y = \max(0, x)$, is applied to the tensor elementwise.

Domain	<code>ai.onnx</code>
Since version	14
Earlier versions	1, 6, 13

Inputs	X (differentiable) : T—Input tensor
Outputs	Y (differentiable) : T—Output tensor
Attributes	None.
Type constraints	T : tensor(float), tensor(int32), tensor(int8), tensor(int16), tensor(int64), tensor(float16), tensor(double), tensor(bfloat16) — Constrain input and output types to signed numeric tensors.
Test vectors	test_relu
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.148. Reshape

Reshape the input tensor similar to `numpy.reshape`. First input is the data tensor, second input is a shape tensor which specifies the output shape. It outputs the reshaped tensor. At most one dimension of the new shape can be -1. In this case, the value is inferred from the size of the tensor and the remaining dimensions. A dimension could also be 0, in which case the actual dimension value is unchanged (i.e. taken from the input tensor). If 'allowzero' is set, and the new shape includes 0, the dimension will be set explicitly to zero (i.e. not taken from input tensor). Shape (second input) could be an empty shape, which means converting to a scalar. The input tensor's shape and the output tensor's shape are required to have the same number of elements.

If the attribute 'allowzero' is set, it is invalid for the specified shape to contain both a zero value and -1, as the value of the dimension corresponding to -1 cannot be determined uniquely.

Domain	ai.onnx
Since version	25
Earlier versions	1, 5, 13, 14, 19, 21, 23, 24
Inputs	data (differentiable) : T—An input tensor. shape (non-differentiable) : tensor(int64) — Specified shape for output.
Outputs	reshaped (differentiable) : T—Reshaped data.
Attributes	allowzero : int (default is 0) —(Optional) By default, when any value in the 'shape' input is equal to zero the corresponding dimension value is copied from the input tensor dynamically. allowzero=1 indicates that if any value in the 'shape' input is set to zero, the zero value is honored, similar to NumPy.
Type constraints	T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128), tensor(float8e4m3fn), tensor(float8e4m3fnuz), tensor(float8e5m2), tensor(float8e5m2fnuz), tensor(uint4), tensor(int4), tensor(float4e2m1), tensor(float8e8m0),

`tensor(uint2)`, `tensor(int2)` — Constrain input and output types to all tensor types.

Test vectors `test_reshape_`

Not supplied by the upstream source Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.149. Resize

Resize the input tensor. In general, it calculates every value in the output tensor as a weighted average of neighborhood (a.k.a. sampling locations) in the input tensor. Each dimension value of the output tensor is:

$$\text{output_dimension} = \text{floor}(\text{input_dimension} * (\text{roi_end} - \text{roi_start}) * \text{scale})$$

if input “sizes” is not specified.

Domain `ai.onnx`

Since version 19

Earlier versions 10, 11, 13, 18

Inputs (1 — 4)

- `X` (differentiable) : `T1` — N-D tensor
- `roi` (optional, non-differentiable) : `T2` — 1-D tensor given as `[start1, ..., startN, end1, ..., endN]`, where `N` is the rank of `X` or the length of axes, if provided. The RoIs' coordinates are normalized in the coordinate system of the input image. It only takes effect when `coordinate_transformation_mode` is “`tf_crop_and_resize`”
- `scales` (optional, non-differentiable) : `tensor(float)` — The scale array along each dimension. It takes value greater than 0. If it's less than 1, it's sampling down, otherwise, it's upsampling. The number of elements of 'scales' should be the same as the rank of input 'X' or the length of 'axes', if provided. One of 'scales' and 'sizes' MUST be specified and it is an error if both are specified. If 'sizes' is needed, the user can use an empty string as the name of 'scales' in this operator's input list.
- `sizes` (optional, non-differentiable) : `tensor(int64)` — Target size of the output tensor. Its interpretation depends on the 'keep_aspect_ratio_policy' value. The number of elements of 'sizes' should be the same as the rank of input 'X', or the length of 'axes', if provided. Only one of 'scales' and 'sizes' can be specified.

Outputs `Y` (differentiable) : `T1` — N-D tensor after resizing

Attributes

- `antialias` : `int` (default is 0) — If set to 1, “linear” and “cubic” interpolation modes will use an antialiasing filter when downscaling. Antialiasing is achieved by stretching the resampling filter by a factor `max(1, 1 / scale)`, which means that when downsampling, more input pixels contribute to an output pixel.
- `axes` : `list of ints` — If provided, it specifies a subset of axes that 'roi', 'scales' and 'sizes' refer to. If not provided, all axes are assumed `[0, 1, ..., r-1]`, where `r = rank(data)`. Non-specified dimensions are interpreted as non-resizable. Negative value means counting dimensions from the back. Accepted range is `[-r, r-1]`, where `r = rank(data)`. Behavior is undefined if an axis is repeated.
- `coordinate_transformation_mode` : `string` (default is `half_pixel`) — This attribute describes how to transform the coordinate in the

resized tensor to the coordinate in the original tensor. The coordinate of each dimension is transformed individually. Let's describe a case using axis x as an example. Denote x_{resized} as the coordinate of axis x in the resized tensor, x_{original} as the coordinate of axis x in the original tensor, $\text{length}_{\text{original}}$ as the length of the original tensor in axis x , $\text{length}_{\text{resized}}$ as the length of the resized tensor in axis x , $\text{scale} = \text{length}_{\text{resized}} / \text{length}_{\text{original}}$, output_width the target length on the axis x which can be a fractional number when it is calculated out of a scale factor, and output_width_int the effective output width as an integer. if `coordinate_transformation_mode` is "half_pixel", $x_{\text{original}} = (x_{\text{resized}} + 0.5) / \text{scale} - 0.5$ if `coordinate_transformation_mode` is "half_pixel_symmetric", $\text{adjustment} = \text{output_width_int} / \text{output_width}$ $\text{center} = \text{input_width} / 2$ $\text{offset} = \text{center} * (1 - \text{adjustment})$ $x_{\text{ori}} = \text{offset} + (x + 0.5) / \text{scale} - 0.5$ if `coordinate_transformation_mode` is "pytorch_half_pixel", $x_{\text{original}} = \text{length}_{\text{resized}} > 1 ? (x_{\text{resized}} + 0.5) / \text{scale} - 0.5 : 0$ if `coordinate_transformation_mode` is "align_corners", $x_{\text{original}} = x_{\text{resized}} * (\text{length}_{\text{original}} - 1) / (\text{length}_{\text{resized}} - 1)$ if `coordinate_transformation_mode` is "asymmetric", $x_{\text{original}} = x_{\text{resized}} / \text{scale}$ if `coordinate_transformation_mode` is "tf_crop_and_resize", $x_{\text{original}} = \text{length}_{\text{resized}} > 1 ? \text{start_x} * (\text{length}_{\text{original}} - 1) + x_{\text{resized}} * (\text{end_x} - \text{start_x}) * (\text{length}_{\text{original}} - 1) / (\text{length}_{\text{resized}} - 1) : 0.5 * (\text{start_x} + \text{end_x}) * (\text{length}_{\text{original}} - 1)$.

`cubic_coeff_a` : float (default is -0.75) — The coefficient 'a' used in cubic interpolation. Two common choice are -0.5 (in some cases of TensorFlow) and -0.75 (in PyTorch). Check out Equation (4) in <https://ieeexplore.ieee.org/document/1163711> for the details. This attribute is valid only if mode is "cubic".

`exclude_outside` : int (default is 0) — If set to 1, the weight of sampling locations outside the tensor will be set to 0 and the weight will be renormalized so that their sum is 1.0. The default value is 0.

`extrapolation_value` : float (default is 0.0) — When `coordinate_transformation_mode` is "tf_crop_and_resize" and x_{original} is outside the range $[0, \text{length}_{\text{original}} - 1]$, this value is used as the corresponding output value. Default is 0.0f.

`keep_aspect_ratio_policy` : string (default is stretch) — This attribute describes how to interpret the sizes input with regard to keeping the original aspect ratio of the input, and it is not applicable when the scales input is used. Given a set of sizes, associated with a subset of axes (explicitly provided or default), and assuming $d = \text{axes}[i]$, with i being the index of the provided sizes. If `keep_aspect_ratio_policy` is "stretch", the original aspect ratio is disregarded, and the input is resized to the specified size: $\text{out_size}[d] = \text{sizes}[i]$ If `keep_aspect_ratio_policy` is "not_larger", the sizes are adjusted so that no extent of the output is larger than the specified size, while keeping the original aspect ratio: $\text{scale} = \text{Min}(\text{sizes}[i] / \text{in_size}[d])$ $\text{out_size}[d] = \text{round_int}(\text{scale} * \text{in_size}[d])$ If `keep_aspect_ratio_policy` is "not_smaller", the sizes are adjusted so that no extent of the output is smaller than the specified size, while keeping the original aspect ratio: $\text{scale} = \text{Max}(\text{sizes}[i] / \text{in_size}[d])$ $\text{out_size}[d] = \text{round_int}(\text{scale} * \text{in_size}[d])$ For non-resizable axes (those not specified in axes), the output size will be equal to the input size. Note: `round_int` stands for computing the nearest integer value, rounding halfway cases up.

`mode` : string (default is nearest) — Three interpolation modes: "nearest" (default), "linear" and "cubic". The "linear" mode includes linear interpolation for 1D tensor and N-linear interpolation for N-D tensor (for example, bilinear interpolation for 2D tensor). The "cubic" mode includes cubic interpolation for 1D tensor and N-cubic interpolation for N-D tensor (for example, bicubic interpolation for 2D tensor).

	nearest_mode : string (default is round_prefer_floor) — Four modes: “round_prefer_floor” (default, as known as round half down), “round_prefer_ceil” (as known as round half up), “floor”, “ceil”. Only used by nearest interpolation. It indicates how to get “nearest” pixel in input tensor from x_original, so this attribute is valid only if “mode” is “nearest”.
Type	T1 : tensor(uint8), tensor(uint16), tensor(uint32),
constraints	tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128) — Constrain input ‘X’ and output ‘Y’ to all tensor types. T2 : tensor(float16), tensor(float), tensor(double) — Constrain roi type to float or double.
Test vectors	test_resize_downsample_scales_cubic, test_resize_downsample_scales_cubic_A_n0p5_exclude_outside, test_resize_downsample_scales_cubic_align_corners, test_resize_downsample_scales_cubic_antialias, test_resize_downsample_scales_linear, test_resize_downsample_scales_linear_align_corners, test_resize_downsample_scales_linear_antialias, test_resize_downsample_scales_linear_half_pixel_symmetric, test_resize_downsample_scales_nearest, test_resize_downsample_sizes_cubic, test_resize_downsample_sizes_cubic_antialias, test_resize_downsample_sizes_linear_antialias, test_resize_downsample_sizes_linear_pytorch_half_pixel, test_resize_downsample_sizes_nearest, test_resize_downsample_sizes_nearest_not_larger, test_resize_downsample_sizes_nearest_not_smaller, test_resize_tf_crop_and_resize, test_resize_tf_crop_and_resize_axes_2_3, test_resize_tf_crop_and_resize_axes_3_2, test_resize_tf_crop_and_resize_extrapolation_value, test_resize_upsample_scales_cubic, test_resize_upsample_scales_cubic_A_n0p5_exclude_outside, test_resize_upsample_scales_cubic_align_corners, test_resize_upsample_scales_cubic_asymmetric, test_resize_upsample_scales_linear, test_resize_upsample_scales_linear_align_corners, test_resize_upsample_scales_linear_half_pixel_symmetric, test_resize_upsample_scales_nearest, test_resize_upsample_scales_nearest_axes_2_3, test_resize_upsample_scales_nearest_axes_3_2, test_resize_upsample_sizes_cubic, test_resize_upsample_sizes_nearest, test_resize_upsample_sizes_nearest_axes_2_3, test_resize_upsample_sizes_nearest_axes_3_2, test_resize_upsample_sizes_nearest_half_pixel, test_resize_upsample_sizes_nearest_floor_align_corners, test_resize_upsample_sizes_nearest_not_larger, test_resize_upsample_sizes_nearest_not_smaller, test_resize_upsample_sizes_nearest_round_prefer_ceil_asymmetric
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.150. ReverseSequence

Reverse batch of sequences having different lengths specified by sequence_lens.

For each slice *i* iterating on batch axis, the operator reverses the first sequence_lens[*i*] elements on time axis, and copies elements whose index's beyond sequence_lens[*i*] to the output. So the output

slice i contains reversed sequences on the first $\text{sequence_lens}[i]$ elements, then have original values copied for the other elements.

Example 1: input = [[0.0, 4.0, 8.0, 12.0], [1.0, 5.0, 9.0, 13.0], [2.0, 6.0, 10.0, 14.0], [3.0, 7.0, 11.0, 15.0]]
sequence_lens = [4, 3, 2, 1] time_axis = 0 batch_axis = 1

output = [[3.0, 6.0, 9.0, 12.0], [2.0, 5.0, 8.0, 13.0], [1.0, 4.0, 10.0, 14.0], [0.0, 7.0, 11.0, 15.0]]

Example 2: input = [[0.0, 1.0, 2.0, 3.0], [4.0, 5.0, 6.0, 7.0], [8.0, 9.0, 10.0, 11.0], [12.0, 13.0, 14.0, 15.0]] sequence_lens = [1, 2, 3, 4] time_axis = 1 batch_axis = 0

output = [[0.0, 1.0, 2.0, 3.0], [5.0, 4.0, 6.0, 7.0], [10.0, 9.0, 8.0, 11.0], [15.0, 14.0, 13.0, 12.0]]

Domain	ai.onnx
Since version	28
Earlier versions	10
Inputs	input : T — Tensor of rank $r \geq 2$. sequence_lens : tensor(int64) — Tensor specifying lengths of the sequences in a batch. It has shape [batch_size].
Outputs	Y : T — Tensor with same shape of input.
Attributes	batch_axis : int (default is 1) — (Optional) Specify which axis is batch axis. Must be one of 1 (default), or 0. time_axis : int (default is 0) — (Optional) Specify which axis is time axis. Must be one of 0 (default), or 1.
Type constraints	T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128) — Input and output types can be of any tensor type.
Test vectors	test_reversesequence_batch, test_reversesequence_bfloat16, test_reversesequence_time
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.151. RoiAlign

Region of Interest (RoI) align operation described in the [Mask R-CNN paper](#). RoiAlign consumes an input tensor X and region of interests (rois) to apply pooling across each RoI; it produces a 4-D tensor of shape (num_rois, C, output_height, output_width).

RoiAlign is proposed to avoid the misalignment by removing quantizations while converting from original image into feature map and from feature map into RoI feature; in each ROI bin, the value of the sampled locations are computed directly through bilinear interpolation.

Domain	ai.onnx
Since version	22

Earlier versions	10, 16
Inputs	X : T1 — Input data tensor from the previous operator; 4-D feature map of shape (N, C, H, W), where N is the batch size, C is the number of channels, and H and W are the height and the width of the data. rois : T1 — RoIs (Regions of Interest) to pool over; rois is 2-D input of shape (num_rois, 4) given as [[x1, y1, x2, y2], ...]. The RoIs' coordinates are in the coordinate system of the input image. Each coordinate set has a 1:1 correspondence with the 'batch_indices' input. batch_indices : T2 — 1-D tensor of shape (num_rois,) with each element denoting the index of the corresponding image in the batch.
Outputs	Y : T1 — RoI pooled output, 4-D tensor of shape (num_rois, C, output_height, output_width). The r-th batch element Y[r-1] is a pooled feature map corresponding to the r-th RoI X[r-1].
Attributes	coordinate_transformation_mode : string (default is half_pixel) — Allowed values are 'half_pixel' and 'output_half_pixel'. Use the value 'half_pixel' to pixel shift the input coordinates by -0.5 (the recommended behavior). Use the value 'output_half_pixel' to omit the pixel shift for the input (use this for a backward-compatible behavior). mode : string (default is avg) — The pooling method. Two modes are supported: 'avg' and 'max'. Default is 'avg'. output_height : int (default is 1) — default 1; Pooled output Y's height. output_width : int (default is 1) — default 1; Pooled output Y's width. sampling_ratio : int (default is 0) — Number of sampling points in the interpolation grid used to compute the output value of each pooled output bin. If > 0, then exactly sampling_ratio x sampling_ratio grid points are used. If == 0, then an adaptive number of grid points are used (computed as ceil(roi_width / output_width), and likewise for height). Default is 0. spatial_scale : float (default is 1.0) — Multiplicative spatial scale factor to translate ROI coordinates from their input spatial scale to the scale used when pooling, i.e., spatial scale of the input feature map X relative to the input image. E.g.; default is 1.0f.
Type constraints	T1 : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain types to float tensors. T2 : tensor(int64) — Constrain types to int tensors.
Test vectors	test_roialign_aligned_false, test_roialign_aligned_true, test_roialign_mode_max
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.152. RotaryEmbedding

RotaryEmbedding is the implementation of rotary positional embeddings (RoPE) based on the paper <https://arxiv.org/pdf/2104.09864>. The key advantage of RoPE is that it allows the model to understand both the absolute position of a token and the relative distances between tokens. This is achieved through a rotational mechanism where the extent of rotation is computed based on the token's absolute position (position_ids).

The rotational mechanism is defined by sine and cosine functions that are used to represent the rotation angles. For each token in the sequence, its positional embedding is computed by rotating its embedding vector. This is done by splitting the embedding vector either into two halves or interleaving every alternate token and applying the rotation matrix to each half of the embedding vector. The rotation matrix is parameterized by the token's position in the sequence. The rotated halves of the embedding vector are concatenated to form the final positional embedding for each token. The rotated positional embeddings are used in the self-attention mechanism. The rotation ensures that the model captures both absolute and relative positional information.

Rotary embeddings are defined using the following algorithm:

```
def rotary_embedding(
    input: np.ndarray,
    cos_cache: np.ndarray,
    sin_cache: np.ndarray,
    position_ids: np.ndarray | None = None,
    interleaved=None,
    rotary_embedding_dim=None,
    num_heads=None,
) -> np.ndarray:
    original_input_shape = input.shape
    # First ensure input to be processed has shape [batch_size, seq_len, num_
heads, head_size]
    if len(input.shape) == 4:
        input = np.transpose(input, (0, 2, 1, 3))
    batch_size = input.shape[0]
    sequence_length = input.shape[1]
    if len(input.shape) == 3:
        hidden_size = input.shape[2]
        assert num_heads != 0
        head_size = int(hidden_size / num_heads)
        new_shape = [batch_size, sequence_length, num_heads, head_size]
        input = np.reshape(input, new_shape)
    assert len(input.shape) == 4
    head_size = input.shape[3]

    # Fully or partially perform rotation on input based on rotary_embedding_dim
attribute
    if rotary_embedding_dim is None or rotary_embedding_dim == 0:
        # If rotary_embedding_dim not provided, perform full rotation by using
head_size
        rotary_embedding_dim = head_size
    x_rotate = input[:, :, :, :rotary_embedding_dim]
    x_not_rotate = input[:, :, :, rotary_embedding_dim:]
    rotary_embedding_dim_half = int(rotary_embedding_dim / 2)

    # Retrieve sin and cos caches using position ids
    if position_ids is not None:
        cos_cache = cos_cache[
            position_ids
        ] # Shape: [batch_size, sequence_length, rotary_embedding_dim/2]
        sin_cache = sin_cache[
            position_ids
        ] # Shape: [batch_size, sequence_length, rotary_embedding_dim/2]

    # Shape: [batch_size, sequence_length, rotary_embedding_dim/2]
    if cos_cache.shape[-1] != rotary_embedding_dim_half:
        raise ValueError(
            f"Last dimension of cos cache ({cos_cache.shape[-1]}) does not match
rotary_embedding_dim/2 ({rotary_embedding_dim_half})."
```

```

    )
    if sin_cache.shape[-1] != rotary_embedding_dim_half:
        raise ValueError(
            f"Last dimension of sin cache ({sin_cache.shape[-1]}) does not match "
            f"rotary_embedding_dim/2 ({rotary_embedding_dim_half})."
        )

    cos_cache = np.expand_dims(
        cos_cache, axis=2
    ) # Shape: [batch_size, sequence_length, 1, rotary_embedding_dim/2]
    sin_cache = np.expand_dims(
        sin_cache, axis=2
    ) # Shape: [batch_size, sequence_length, 1, rotary_embedding_dim/2]

    # Either divide the input in halves or interleave (based on interleaved
    attribute)
    if interleaved:
        x1 = x_rotate[:, :, :, 0::2]
        x2 = x_rotate[:, :, :, 1::2]
    else:
        x1, x2 = np.split(x_rotate, 2, axis=-1)

    # Calculate real and imaginary values
    real = (cos_cache * x1) - (sin_cache * x2)
    imag = (sin_cache * x1) + (cos_cache * x2)

    # Inserted rotated embeddings back to the original input
    if interleaved:
        # x_rotate[:, :, :, 0::2] = real
        # x_rotate[:, :, :, 1::2] = imag
        real = np.expand_dims(real, axis=-1)
        imag = np.expand_dims(imag, axis=-1)
        x_rotate_concat = np.concatenate((real, imag), axis=-1)
        x_rotate = np.reshape(x_rotate_concat, x_rotate.shape)
    else:
        x_rotate = np.concatenate((real, imag), axis=-1)
    output = np.concatenate((x_rotate, x_not_rotate), axis=-1)
    if len(original_input_shape) == 3:
        output = np.reshape(output, original_input_shape)
    else:
        output = np.transpose(output, (0, 2, 1, 3))
    return output

```

Domain ai.onnx

Since version 23

Inputs (3 — 4) X : T — The input tensor representing the token embeddings. 4D tensor with shape (batch_size, num_heads, sequence_length, head_size) or 3D tensor with shape (batch_size, sequence_length, hidden_size). For cases with a 4D input tensor, head_size has to be even. For cases with a 3D input tensor, num_heads attribute must be provided and hidden_size must be an even multiple of num_heads where hidden_size = num_heads * head_size

cos_cache : T — The cosine values for the rotation. 2D tensor with shape (max_position_id_plus_1, head_size / 2) for full rotation or (max_position_id_plus_1, rotary_embedding_dim / 2) for partial rotation when position_ids are provided. 3D tensor with shape (batch_size, sequence_length, head_size / 2) for full rotation or (batch_size, sequence_length, rotary_embedding_dim / 2) for partial

	rotation when <code>position_ids</code> are not provided. <code>max_position_id_plus_1</code> is a parameter to the model. <code>sin_cache</code> : T — The sine values for the rotation. 2D tensor with shape $(\text{max_position_id_plus_1}, \text{head_size} / 2)$ for full rotation or $(\text{max_position_id_plus_1}, \text{rotary_embedding_dim} / 2)$ for partial rotation when <code>position_ids</code> are provided. 3D tensor with shape $(\text{batch_size}, \text{sequence_length}, \text{head_size} / 2)$ for full rotation or $(\text{batch_size}, \text{sequence_length}, \text{rotary_embedding_dim} / 2)$ for partial rotation when <code>position_ids</code> are not provided. <code>max_position_id_plus_1</code> is a parameter to the model. <code>position_ids</code> (optional) : M — The position indices for the tokens. 2D tensor with shape $(\text{batch_size}, \text{sequence_length})$
Outputs	Y : T — Tensor with same shape as input.
Attributes	<code>interleaved</code> : int (default is 0) — Rotate using interleaved pattern. Default value is 0 (False). <code>num_heads</code> : int — Number of attention heads. Must be provided when input is a 3D tensor. <code>rotary_embedding_dim</code> : int (default is 0) — Rotary embedding dimension used to apply partial rotary embeddings.
Type constraints	T : <code>tensor(float)</code>, <code>tensor(float16)</code>, <code>tensor(bfloat16)</code> — Constrain input and output types to float tensors. M : <code>tensor(int64)</code> — Constrain input and output types to integer tensors.
Test vectors	<code>test_rotary_embedding</code> , <code>test_rotary_embedding_3d_input</code> , <code>test_rotary_embedding_interleaved</code> , <code>test_rotary_embedding_no_position_ids</code> , <code>test_rotary_embedding_no_position_ids_interleaved</code> , <code>test_rotary_embedding_no_position_ids_rotary_dim</code> , <code>test_rotary_embedding_with_interleaved_rotary_dim</code> , <code>test_rotary_embedding_with_rotary_dim</code>
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.153. Round

Round takes one input Tensor and rounds the values, element-wise, meaning it finds the nearest integer for each value. In case of halves, the rule is to round them to the nearest even integer. If input `x` is integral, `+0`, `-0`, `NaN`, or infinite, `x` itself is returned. The output tensor has the same shape and type as the input.

Examples:

```
round([0.9]) = [1.0]
round([2.5]) = [2.0]
round([2.3]) = [2.0]
round([1.5]) = [2.0]
round([-4.5]) = [-4.0]
```

Domain ai.onnx

Since version 22

Earlier versions 11

Inputs	X (non-differentiable) : T—Input tensor
Outputs	Y (non-differentiable) : T—Output tensor
Attributes	None.
Type constraints	T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain input and output types to float tensors.
Test vectors	test_round
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.154. STFT

Computes the Short-time Fourier Transform of the signal.

The STFT is computed by sliding a window of length `frame_length` over the signal with a step size of `frame_step`, computing a DFT of each windowed frame.

The number of frames in the output is computed as:

$$\text{frames} = \text{floor}((\text{signal_length} - \text{frame_length}) / \text{frame_step}) + 1$$

Constraints on inputs:

- `frame_step` must be a scalar.
- `frame_length` must be a scalar. When omitted and `window` is provided, `frame_length` is inferred from `window.shape[0]`. When both `window` and `frame_length` are omitted, `frame_length` defaults to `signal_length`.
- `window` must be a 1-D tensor. When omitted, a rectangular (all-ones) window of length `frame_length` is used. When both `window` and `frame_length` are provided, the length of the window tensor must equal `frame_length`.

Domain ai.onnx

Since version 17

Inputs (2 — 4) `signal` (non-differentiable) : T1—Input tensor representing a real or complex valued signal. For real input, the following shape is expected: `[batch_size][signal_length][1]`. For complex input, the following shape is expected: `[batch_size][signal_length][2]`, where `[batch_size][signal_length][0]` represents the real component and `[batch_size][signal_length][1]` represents the imaginary component of the signal. The tensor is expected to have rank 3.

`frame_step` (non-differentiable) : T2—A scalar representing the number of samples to step between successive DFTs.

`window` (optional, non-differentiable) : T1—An optional 1-D tensor representing the window function to be applied to each frame of the signal before computing the DFT. The length of the window (`window.shape[0]`) determines the frame length when `frame_length` is not specified. If both `window` and `frame_length` are provided, the length of the window must equal `frame_length`. When omitted, a rectangular (all-ones) window of length `frame_length` is used.

`frame_length` (optional, non-differentiable) : T2—An optional scalar representing the length of each frame (i.e., the DFT size). When omitted and `window` is provided, `frame_length` is inferred from `window.shape[0]`. When both `window` and `frame_length` are omitted, `frame_`

length defaults to <code>signal_length</code> . If both <code>frame_length</code> and <code>window</code> are provided, the length of the window must equal <code>frame_length</code> .	
Outputs	output (non-differentiable) : T1 — The Short-time Fourier Transform of the signal. The number of frames in the output is <code>frames = floor((signal_length - frame_length) / frame_step) + 1</code> . If <code>onesided</code> is 1, the output has the shape: <code>[batch_size][frames][dft_unique_bins][2]</code> , where <code>dft_unique_bins</code> is <code>frame_length // 2 + 1</code> (the unique components of the DFT). If <code>onesided</code> is 0, the output has the shape: <code>[batch_size][frames][frame_length][2]</code> , where <code>frame_length</code> is the length of the DFT. The last dimension of size 2 represents the real and imaginary parts of each complex value.
Attributes	<code>onesided</code> : int (default is 1) — If <code>onesided</code> is 1, only values for <code>w</code> in <code>[0, 1, 2, ..., floor(n_fft/2) + 1]</code> are returned because the real-to-complex Fourier transform satisfies the conjugate symmetry, i.e., $X[m, w] = X[m, n_fft - w]^*$. Note if the input or window tensors are complex, then <code>onesided</code> output is not possible. Enabling <code>onesided</code> with real inputs performs a Real-valued fast Fourier transform (RFFT). When invoked with real or complex valued input, the default value is 1. Values can be 0 or 1.
Type constraints	T1 : tensor(float), tensor(float16), tensor(double), tensor(bfloat16) — Constrain signal and output to float tensors. T2 : tensor(int32), tensor(int64) — Constrain scalar length types to int64_t.
Test vectors	<code>test_stft</code> , <code>test_stft_with_window</code>
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.155. Scan

Scan can be used to iterate over one or more `scan_input` tensors, constructing zero or more `scan_output` tensors. It combines ideas from general recurrences, functional programming constructs such as `scan`, `fold`, `map`, and `zip`, and is intended to enable generalizations of RNN-like constructs for sequence-to-sequence processing. Other tensors (referred to as `state_variables` here) can be used to carry a state when iterating from one element to another (similar to hidden-state in RNNs, also referred to as loop-carried dependences in the context of loops). Many common usages involve a single `scan_input` tensor (where functionality similar to `scan`, `fold` and `map` can be obtained). When more than one `scan_input` is used, a behavior similar to `zip` is obtained.

The attribute body must be a graph, specifying the computation to be performed in every iteration. It takes as input the current values of the `state_variables` and the current iterated element of the `scan_inputs`. It must return the (updated) values of the `state_variables` and zero or more `scan_output_element` tensors. The values of the `scan_output_element` tensors are concatenated over all the iterations to produce the `scan_output` values of the `scan` construct (similar to the concatenated intermediate hidden-state values of RNN-like constructs). All the output tensors (`state_variables` as well as `scan_output_element` tensors) are required to have the same shape in each iteration of the loop (a restriction imposed to enable efficient memory allocation).

Note that the iterated element passed to the body subgraph does not have a sequence axis. It will have a rank one less than the rank of the corresponding `scan_input`.

The `scan` operation returns the final values of the `state_variables` as well as the `scan_outputs`.

The optional attribute `scan_input_directions` specifies the direction (forward or backward) for each scan input. If this attribute is omitted, all sequences are scanned in the forward direction. A

bidirectional scan may be performed by specifying the same tensor input twice in the scan_inputs, once with a forward direction, and once with a backward direction.

The scan_output of the operation is produced by concatenating the scan_output_element values produced by the body in each iteration. The optional attribute scan_output_directions specifies the direction in which scan_output is constructed (by appending or prepending the scan_output_element to scan_output in each iteration) for each scan_output. If this attribute is omitted, the scan_output_element is appended to the scan_output in each iteration.

The optional attribute scan_input_axes specifies the axis to be scanned for each scan_input. If omitted, every scan_input will be scanned in axis 0. For example, if axis 0 is the batch axis and axis 1 is the time axis (to be scanned), specify an axis value of 1. Note that scanning a non-zero axis may be less efficient than scanning axis zero.

The optional attribute scan_output_axes specifies the axis along which the scan_outputs are accumulated for each scan_output. For example, if axis 1 is the time axis (to be scanned) for both inputs and outputs, specify a scan_input axis and scan_output axis value of 1.

Note that because of the ONNX restriction that only the last parameter of an operator can be variadic, the initial-states and scan-inputs are listed together as one input parameter. Similarly, the final-states and scan-outputs are listed together as one output parameter. The attribute num_scan_inputs indicates the number M of scan-inputs.

The behavior of

```
Scan <
  num_scan_inputs = m,
  body = loop-body,
  scan_input_axes = [axis_1, ..., axis_m]
> (init_1, ..., init_n, scan_1, ..., scan_m)
```

is equivalent to the following pseudo-code:

```
// scan_i.shape[axis_i] denotes the (max) sequence-length of scan_i
// scan_i.shape[axis_i] is required to be equal to scan_j.shape[axis_j] for all i,j.
sequence_length = scan_1.shape[axis_1];

// initialize state-variables
st_1 = init_1; ... st_n = init_n;
// initialize scan-output variables: [] denotes an empty tensor
scan_out_1 = []; ...; scan_out_k = [];
// identify number of iterations:

// execute loop
for (int t = 0; t < sequence_length; ++t) {
  // generate the scan-input elements: the notation `T<axis=k>`[t] indicates
the sub-tensor
  // of rank one less than T obtained by indexing T at position t along axis k.
  si_1 = `scan_1<axis=axis_1>`[t];
  ... ;
  si_m = `scan_m<axis=axis_m>`[t];
  // execute loop-body
  st_1, ..., st_n, so_1, ..., so_k = loop-body(st_1, ..., st_n, si_1, ..., si_m)
  // accumulate the scan-output elements
  scan_out_1 = `Concat<axis=0>`(scan_out_1, so_1); ... ; scan_out_k = `Concat
<axis=0>`(scan_out_k, so_k);
}
```

```
return st_1, ..., st_n, scan_out_1, ..., scan_out_k;
```

Sample usage: Encoding RNN using a Scan

The following example shows how a simple RNN over an input tensor %X, with weight tensor %Wi, recurrence weight tensor %Ri, bias tensors %Wbi and %Rbi, and initial hidden-state %H_0 can be encoded as a ScanLoop. Note that the loop-body is a nested graph, and it directly computes %Wi, %Ri, %Wbi, and %Rbi (typically constants or initializers in the body graph). If these values are computed in the outer graph, they need to be passed in as extra state_variables.

```
graph rnn-encoding {
  %H_0 = ...
  %X = ...
  %Y_h, %Y = Scanbody = <graph rnn-cell-1>, num_scan_inputs=1
  return %Y, %Y_h
}
```

```
graph rnn-cell-1 (
  %H_tminus1[FLOAT, tensor]
  %X_t[FLOAT, tensor]
) {
  %Wi = ...
  %Ri = ...
  %Wbi = ...
  %Rbi = ...
  %t1 = X_t * (Wi^T)
  %t2 = H_tminus1*(Ri^T)
  %t3 = Add(%t1, %t2)
  %t4 = Add(%t3, %Wbi)
  %t5 = Add(%t4, %Rbi)
  %Ht = Tanh(%t5)
  %Accumulate = Identity(%Ht)
  return %Ht, %Accumulate
}
```

Domain ai.onnx

Since version 25

Earlier versions 8, 9, 11, 16, 19, 21, 23, 24

Inputs (1 — unbounded) initial_state_and_scan_inputs (variadic, heterogeneous) : V — Initial values of the loop's N state variables followed by M scan_inputs

Outputs (1 — unbounded) final_state_and_scan_outputs (variadic, heterogeneous) : V — Final values of the loop's N state variables followed by K scan_outputs

Attributes

- body : graph (required) — The graph run each iteration. It has N +M inputs: (loop state variables..., scan_input_elts...). It has N+K outputs: (loop state variables..., scan_output_elts...). Each scan_output is created by concatenating the value of the specified scan_output_elt value at the end of each iteration of the loop. It is an error if the dimensions of these values change across loop iterations.
- num_scan_inputs : int (required) — An attribute specifying the number of scan_inputs M.
- scan_input_axes : list of ints — An optional list of M flags. The i-th element of the list specifies the axis to be scanned (the sequence axis) for the i-th scan_input. If omitted, 0 will be used as the scan axis for every scan_input. Negative value for an axis means counting dimensions from the back. Accepted range is [-r, r-1] where r = rank(input).

	<code>scan_input_directions</code> : list of ints — An optional list of M flags. The i-th element of the list specifies the direction to be scanned for the i-th <code>scan_input</code> tensor: 0 indicates forward direction and 1 indicates reverse direction. If omitted, all <code>scan_input</code> tensors will be scanned in the forward direction. <code>scan_output_axes</code> : list of ints — An optional list of K flags. The i-th element of the list specifies the axis for the i-th <code>scan_output</code>. The scan outputs are accumulated along the specified axis. If omitted, 0 will be used as the scan axis for every <code>scan_output</code>. Negative value for an axis means counting dimensions from the back. Accepted range is $[-r, r-1]$. <code>scan_output_directions</code> : list of ints — An optional list of K flags, one for each <code>scan_output</code>. The i-th element of the list specifies whether the i-th <code>scan_output</code> should be constructed by appending or prepending a new value in each iteration: 0 indicates appending and 1 indicates prepending. If omitted, all <code>scan_output</code> tensors will be produced by appending a value in each iteration.
Type constraints	V : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128), tensor(float8e4m3fn), tensor(float8e4m3fnuz), tensor(float8e5m2), tensor(float8e5m2fnuz), tensor(uint4), tensor(int4), tensor(float4e2m1), tensor(float8e8m0), tensor(uint2), tensor(int2) — All Tensor types up to IRv13.
Test vectors	<code>test_scan_sum</code>, <code>test_scan9_sum</code>, <code>test_scan9_multi_state</code>, <code>test_scan9_scalar</code>
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.156. Scatter

This operator is deprecated. Please use `ScatterElements`, which provides the same functionality.

Scatter takes three inputs `data`, `updates`, and `indices` of the same rank $r \geq 1$ and an optional attribute `axis` that identifies an axis of `data` (by default, the outer-most axis, that is axis 0). The output of the operation is produced by creating a copy of the input `data`, and then updating its value to values specified by `updates` at specific index positions specified by `indices`. Its output shape is the same as the shape of `data`.

For each entry in `updates`, the target index in `data` is obtained by combining the corresponding entry in `indices` with the index of the entry itself: the index-value for dimension = `axis` is obtained from the value of the corresponding entry in `indices` and the index-value for dimension $\neq$ `axis` is obtained from the index of the entry itself.

For instance, in a 2-D tensor case, the update corresponding to the `[i][j]` entry is performed as below:

```
output[indices[i][j]][j] = updates[i][j] if axis = 0,
output[i][indices[i][j]] = updates[i][j] if axis = 1,
```

This operator is the inverse of `GatherElements`. It is similar to Torch's `Scatter` operation.

Example 1:

```

data = [
    [0.0, 0.0, 0.0],
    [0.0, 0.0, 0.0],
    [0.0, 0.0, 0.0],
]
indices = [
    [1, 0, 2],
    [0, 2, 1],
]
updates = [
    [1.0, 1.1, 1.2],
    [2.0, 2.1, 2.2],
]
output = [
    [2.0, 1.1, 0.0]
    [1.0, 0.0, 2.2]
    [0.0, 2.1, 1.2]
]

```

Example 2:

```

data = [[1.0, 2.0, 3.0, 4.0, 5.0]]
indices = [[1, 3]]
updates = [[1.1, 2.1]]
axis = 1
output = [[1.0, 1.1, 3.0, 2.1, 5.0]]

```

Domain	ai.onnx
Since version	11
Status	Deprecated. Clause 14.5 of ONNX 1-1 applies: a deprecated operator is not removed, and a consumer continues to evaluate it.
Earlier versions	9
Inputs	Not stated by the source.
Outputs	Not stated by the source.
Attributes	None.
Type constraints	Not stated by the source.
Test vectors	test_scatter_with_axis, test_scatter_without_axis
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.157. ScatterElements

ScatterElements takes three inputs `data`, `updates`, and `indices` of the same rank $r \geq 1$ and an optional attribute `axis` that identifies an axis of `data` (by default, the outer-most axis, that is axis 0). The output of the operation is produced by creating a copy of the input `data`, and then updating its value to values specified by `updates` at specific index positions specified by `indices`. Its output shape is the same as the shape of `data`.

For each entry in `updates`, the target index in `data` is obtained by combining the corresponding entry in `indices` with the index of the entry itself: the index-value for dimension = `axis` is obtained

from the value of the corresponding entry in indices and the index-value for dimension != axis is obtained from the index of the entry itself.

reduction allows specification of an optional reduction operation, which is applied to all values in updates tensor into output at the specified indices. In cases where reduction is set to "none", indices should not have duplicate entries: that is, if $idx1 \neq idx2$, then $indices[idx1] \neq indices[idx2]$. For instance, in a 2-D tensor case, the update corresponding to the $[i][j]$ entry is performed as below:

```
output[indices[i][j]][j] = updates[i][j] if axis = 0,
output[i][indices[i][j]] = updates[i][j] if axis = 1,
```

When reduction is set to some reduction function f , the update corresponding to the $[i][j]$ entry is performed as below:

```
output[indices[i][j]][j] = f(output[indices[i][j]][j], updates[i][j]) if axis =
0,
output[i][indices[i][j]] = f(output[i][indices[i][j]], updates[i][j]) if axis =
1,
```

where the f is +, *, max or min as specified.

This operator is the inverse of GatherElements. It is similar to Torch's Scatter operation.

(Opset 18 change): Adds max/min to the set of allowed reduction ops.

Example 1:

```
data = [
    [0.0, 0.0, 0.0],
    [0.0, 0.0, 0.0],
    [0.0, 0.0, 0.0],
]
indices = [
    [1, 0, 2],
    [0, 2, 1],
]
updates = [
    [1.0, 1.1, 1.2],
    [2.0, 2.1, 2.2],
]
output = [
    [2.0, 1.1, 0.0]
    [1.0, 0.0, 2.2]
    [0.0, 2.1, 1.2]
]
```

Example 2:

```
data = [[1.0, 2.0, 3.0, 4.0, 5.0]]
indices = [[1, 3]]
updates = [[1.1, 2.1]]
axis = 1
output = [[1.0, 1.1, 3.0, 2.1, 5.0]]
```

Domain ai.onnx

Since version 18

Earlier versions 11, 13, 16

Inputs data (differentiable) : T — Tensor of rank $r \geq 1$.

	<code>indices</code> (non-differentiable) : <code>Tind</code> — Tensor of <code>int32/int64</code> indices, of $r \geq 1$ (same rank as input). All index values are expected to be within bounds $[-s, s-1]$ along axis of size s. It is an error if any of the index values are out of bounds. <code>updates</code> (differentiable) : <code>T</code> — Tensor of rank $r \geq 1$ (same rank and shape as <code>indices</code>)
Outputs	<code>output</code> (differentiable) : <code>T</code> — Tensor of rank $r \geq 1$ (same rank as input).
Attributes	<code>axis</code> : <code>int</code> (default is <code>0</code>) — Which axis to scatter on. Negative value means counting dimensions from the back. Accepted range is $[-r, r-1]$ where $r = \text{rank}(\text{data})$. <code>reduction</code> : <code>string</code> (default is <code>none</code>) — Type of reduction to apply: <code>none</code> (default), <code>add</code>, <code>mul</code>, <code>max</code>, <code>min</code>. 'none': no reduction applied. 'add': reduction using the addition operation. 'mul': reduction using the multiplication operation. 'max': reduction using the maximum operation. 'min': reduction using the minimum operation.
Type constraints	<code>T</code> : <code>tensor(uint8)</code>, <code>tensor(uint16)</code>, <code>tensor(uint32)</code>, <code>tensor(uint64)</code>, <code>tensor(int8)</code>, <code>tensor(int16)</code>, <code>tensor(int32)</code>, <code>tensor(int64)</code>, <code>tensor(bfloat16)</code>, <code>tensor(float16)</code>, <code>tensor(float)</code>, <code>tensor(double)</code>, <code>tensor(string)</code>, <code>tensor(bool)</code>, <code>tensor(complex64)</code>, <code>tensor(complex128)</code> — Input and output types can be of any tensor type. <code>Tind</code> : <code>tensor(int32)</code>, <code>tensor(int64)</code> — Constrain indices to integer types
Test vectors	<code>test_scatter_elements_with_axis</code> , <code>test_scatter_elements_with_duplicate_indices</code> , <code>test_scatter_elements_with_negative_indices</code> , <code>test_scatter_elements_with_reduction_max</code> , <code>test_scatter_elements_with_reduction_min</code> , <code>test_scatter_elements_with_reduction_mul</code> , <code>test_scatter_elements_without_axis</code>
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.158. ScatterND

ScatterND takes three inputs `data` tensor of rank $r \geq 1$, `indices` tensor of rank $q \geq 1$, and `updates` tensor of rank $q + r - \text{indices.shape}[-1] - 1$. The output of the operation is produced by creating a copy of the input `data`, and then updating its value to values specified by `updates` at specific index positions specified by `indices`. Its output shape is the same as the shape of `data`.

`indices` is an integer tensor. Let k denote `indices.shape[-1]`, the last dimension in the shape of `indices`. `indices` is treated as a $(q-1)$ -dimensional tensor of k -tuples, where each k -tuple is a partial-index into `data`. Hence, k can be a value at most the rank of `data`. When k equals $\text{rank}(\text{data})$, each update entry specifies an update to a single element of the tensor. When k is less than $\text{rank}(\text{data})$ each update entry specifies an update to a slice of the tensor. Index values are allowed to be negative, as per the usual convention for counting backwards from the end, but are expected in the valid range.

`updates` is treated as a $(q-1)$ -dimensional tensor of replacement-slice-values. Thus, the first $(q-1)$ dimensions of `updates.shape` must match the first $(q-1)$ dimensions of `indices.shape`. The remaining dimensions of `updates` correspond to the dimensions of the replacement-slice-values. Each replacement-slice-value is a $(r-k)$ dimensional tensor, corresponding to the

trailing (r-k) dimensions of data. Thus, the shape of updates must equal `indices.shape[0:q-1] ++ data.shape[k:r]`, where ++ denotes the concatenation of shapes.

The output is calculated via the following equation:

```
output = np.copy(data)
update_indices = indices.shape[:-1]
for idx in np.ndindex(update_indices):
    output[tuple(indices[idx])] = updates[idx]
```

The order of iteration in the above loop is not specified. In particular, indices should not have duplicate entries: that is, if `idx1 != idx2`, then `indices[idx1] != indices[idx2]`. This ensures that the output value does not depend on the iteration order.

reduction allows specification of an optional reduction operation, which is applied to all values in updates tensor into output at the specified indices. In cases where reduction is set to “none”, indices should not have duplicate entries: that is, if `idx1 != idx2`, then `indices[idx1] != indices[idx2]`. This ensures that the output value does not depend on the iteration order. When reduction is set to some reduction function *f*, output is calculated as follows:

```
output = np.copy(data)
update_indices = indices.shape[:-1]
for idx in np.ndindex(update_indices):
    output[tuple(indices[idx])] = f(output[tuple(indices[idx])], updates[idx])
```

where the *f* is +, *, max or min as specified.

This operator is the inverse of GatherND.

(Opset 18 change): Adds max/min to the set of allowed reduction ops.

Example 1:

```
data      = [1, 2, 3, 4, 5, 6, 7, 8]
indices   = [[4], [3], [1], [7]]
updates   = [9, 10, 11, 12]
output    = [1, 11, 3, 10, 9, 6, 7, 12]
```

Example 2:

```
data      = [[1, 2, 3, 4], [5, 6, 7, 8], [8, 7, 6, 5], [4, 3, 2, 1]],
             [[1, 2, 3, 4], [5, 6, 7, 8], [8, 7, 6, 5], [4, 3, 2, 1]],
             [[8, 7, 6, 5], [4, 3, 2, 1], [1, 2, 3, 4], [5, 6, 7, 8]],
             [[8, 7, 6, 5], [4, 3, 2, 1], [1, 2, 3, 4], [5, 6, 7, 8]]]
indices   = [[0], [2]]
updates   = [[5, 5, 5, 5], [6, 6, 6, 6], [7, 7, 7, 7], [8, 8, 8, 8]],
             [[1, 1, 1, 1], [2, 2, 2, 2], [3, 3, 3, 3], [4, 4, 4, 4]]]
output    = [[5, 5, 5, 5], [6, 6, 6, 6], [7, 7, 7, 7], [8, 8, 8, 8]],
             [[1, 2, 3, 4], [5, 6, 7, 8], [8, 7, 6, 5], [4, 3, 2, 1]],
             [[1, 1, 1, 1], [2, 2, 2, 2], [3, 3, 3, 3], [4, 4, 4, 4]],
             [[8, 7, 6, 5], [4, 3, 2, 1], [1, 2, 3, 4], [5, 6, 7, 8]]]
```

Domain ai.onnx

Since version 18

Earlier versions 11, 13, 16

Inputs data (differentiable) : T — Tensor of rank $r \geq 1$.
 indices (non-differentiable) : tensor(int64) — Tensor of rank $q \geq 1$.

	updates (differentiable) : T — Tensor of rank $q + r$ — indices_shape[-1] — 1.
Outputs	output (differentiable) : T — Tensor of rank $r \geq 1$.
Attributes	reduction : string (default is none) — Type of reduction to apply: none (default), add, mul, max, min. 'none': no reduction applied. 'add': reduction using the addition operation. 'mul': reduction using the addition operation. 'max': reduction using the maximum operation. 'min': reduction using the minimum operation.
Type constraints	T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128) — Constrain input and output types to any tensor type.
Test vectors	test_scatternd, test_scatternd_add, test_scatternd_max, test_scatternd_max_with_element_indices, test_scatternd_min, test_scatternd_min_with_element_indices, test_scatternd_multiply
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.159. Selu

Selu takes one input data (Tensor<T>) and produces one output data (Tensor<T>) where the scaled exponential linear unit function, $y = \gamma * (\alpha * e^x - \alpha)$ for $x \leq 0$, $y = \gamma * x$ for $x > 0$, is applied to the tensor elementwise.

Domain	ai.onnx
Since version	22
Earlier versions	1, 6
Inputs	X (differentiable) : T — Input tensor
Outputs	Y (differentiable) : T — Output tensor
Attributes	alpha : float (default is 1.67326) — Coefficient of SELU default to 1.67326319217681884765625 (i.e., float32 approximation of 1.6732632423543772848170429916717). gamma : float (default is 1.0507) — Coefficient of SELU default to 1.05070102214813232421875 (i.e., float32 approximation of 1.0507009873554804934193349852946).
Type constraints	T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain input and output types to float tensors.
Test vectors	test_selu_example, test_selu, test_selu_default
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.160. SequenceAt

Outputs a tensor copy from the tensor at 'position' in 'input_sequence'. Accepted range for 'position' is in $[-n, n - 1]$, where n is the number of tensors in 'input_sequence'. Negative value means counting positions from the back.

Domain	ai.onnx
Since version	11
Inputs	input_sequence : S — Input sequence. position : I — Position of the tensor in the sequence. Negative value means counting positions from the back. Accepted range in $[-n, n - 1]$, where n is the number of tensors in 'input_sequence'. It is an error if any of the index values are out of bounds. It must be a scalar(tensor of empty shape).
Outputs	tensor : T — Output tensor at the specified position in the input sequence.
Attributes	None.
Type constraints	S : seq(tensor(uint8)), seq(tensor(uint16)), seq(tensor(uint32)), seq(tensor(uint64)), seq(tensor(int8)), seq(tensor(int16)), seq(tensor(int32)), seq(tensor(int64)), seq(tensor(float16)), seq(tensor(float)), seq(tensor(double)), seq(tensor(string)), seq(tensor(bool)), seq(tensor(complex64)), seq(tensor(complex128)) — Constrain to any tensor type. T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128) — Constrain to any tensor type. I : tensor(int32), tensor(int64) — Constrain position to integral tensor. It must be a scalar(tensor of empty shape).
Test vectors	None published for this operator.
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.161. SequenceConstruct

Construct a tensor sequence containing 'inputs' tensors. All tensors in 'inputs' must have the same data type.

Domain	ai.onnx
Since version	11
Inputs (1 — unbounded)	inputs (variadic) : T — Tensors.
Outputs	output_sequence : S — Sequence enclosing the input tensors.

Attributes	None.
Type constraints	T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128) — Constrain input types to any tensor type. S : seq(tensor(uint8)), seq(tensor(uint16)), seq(tensor(uint32)), seq(tensor(uint64)), seq(tensor(int8)), seq(tensor(int16)), seq(tensor(int32)), seq(tensor(int64)), seq(tensor(float16)), seq(tensor(float)), seq(tensor(double)), seq(tensor(string)), seq(tensor(bool)), seq(tensor(complex64)), seq(tensor(complex128)) — Constrain output types to any tensor type.
Test vectors	None published for this operator.
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.162. SequenceEmpty

Construct an empty tensor sequence, with given data type.

Domain	ai.onnx
Since version	11
Inputs	None.
Outputs	output : S — Empty sequence.
Attributes	dtype : int — (Optional) The data type of the tensors in the output sequence. The default type is 'float'.
Type constraints	S : seq(tensor(uint8)), seq(tensor(uint16)), seq(tensor(uint32)), seq(tensor(uint64)), seq(tensor(int8)), seq(tensor(int16)), seq(tensor(int32)), seq(tensor(int64)), seq(tensor(float16)), seq(tensor(float)), seq(tensor(double)), seq(tensor(string)), seq(tensor(bool)), seq(tensor(complex64)), seq(tensor(complex128)) — Constrain output types to any tensor type.
Test vectors	None published for this operator.
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.163. SequenceErase

Outputs a tensor sequence that removes the tensor at 'position' from 'input_sequence'. Accepted range for 'position' is in $[-n, n - 1]$, where n is the number of tensors in 'input_sequence'.

Negative value means counting positions from the back. 'position' is optional, by default it erases the last tensor from 'input_sequence'.

Domain	ai.onnx
Since version	11
Inputs (1 — 2)	input_sequence : S — Input sequence. position (optional) : I — Position of the tensor in the sequence. Negative value means counting positions from the back. Accepted range in $[-n, n - 1]$, where n is the number of tensors in 'input_sequence'. It is an error if any of the index values are out of bounds. It must be a scalar(tensor of empty shape).
Outputs	output_sequence : S — Output sequence that has the tensor at the specified position removed.
Attributes	None.
Type constraints	S : seq(tensor(uint8)), seq(tensor(uint16)), seq(tensor(uint32)), seq(tensor(uint64)), seq(tensor(int8)), seq(tensor(int16)), seq(tensor(int32)), seq(tensor(int64)), seq(tensor(float16)), seq(tensor(float)), seq(tensor(double)), seq(tensor(string)), seq(tensor(bool)), seq(tensor(complex64)), seq(tensor(complex128)) — Constrain to any tensor type. I : tensor(int32), tensor(int64) — Constrain position to integral tensor. It must be a scalar(tensor of empty shape).
Test vectors	None published for this operator.
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.164. SequenceInsert

Outputs a tensor sequence that inserts 'tensor' into 'input_sequence' at 'position'. 'tensor' must have the same data type as 'input_sequence'. Accepted range for 'position' is in $[-n, n]$, where n is the number of tensors in 'input_sequence'. Negative value means counting positions from the back. 'position' is optional, by default it inserts 'tensor' to the back of 'input_sequence'.

Domain	ai.onnx
Since version	11
Inputs (2 — 3)	input_sequence : S — Input sequence. tensor : T — Input tensor to be inserted into the input sequence. position (optional) : I — Position in the sequence where the new tensor is inserted. It is optional and default is to insert to the back of the sequence. Negative value means counting positions from the back. Accepted range in $[-n, n]$, where n is the number of tensors in 'input_sequence'. It is an error if any of the index values are out of bounds. It must be a scalar(tensor of empty shape).
Outputs	output_sequence : S — Output sequence that contains the inserted tensor at given position.

Attributes	None.
Type constraints	T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128) — Constrain to any tensor type. S : seq(tensor(uint8)), seq(tensor(uint16)), seq(tensor(uint32)), seq(tensor(uint64)), seq(tensor(int8)), seq(tensor(int16)), seq(tensor(int32)), seq(tensor(int64)), seq(tensor(float16)), seq(tensor(float)), seq(tensor(double)), seq(tensor(string)), seq(tensor(bool)), seq(tensor(complex64)), seq(tensor(complex128)) — Constrain to any tensor type. I : tensor(int32), tensor(int64) — Constrain position to integral tensor. It must be a scalar(tensor of empty shape).
Test vectors	test_sequence_insert_
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.165. SequenceLength

Produces a scalar(tensor of empty shape) containing the number of tensors in 'input_sequence'.

Domain	ai.onnx
Since version	11
Inputs	input_sequence : S — Input sequence.
Outputs	length : I — Length of input sequence. It must be a scalar(tensor of empty shape).
Attributes	None.
Type constraints	S : seq(tensor(uint8)), seq(tensor(uint16)), seq(tensor(uint32)), seq(tensor(uint64)), seq(tensor(int8)), seq(tensor(int16)), seq(tensor(int32)), seq(tensor(int64)), seq(tensor(float16)), seq(tensor(float)), seq(tensor(double)), seq(tensor(string)), seq(tensor(bool)), seq(tensor(complex64)), seq(tensor(complex128)) — Constrain to any tensor type. I : tensor(int64) — Constrain output to integral tensor. It must be a scalar(tensor of empty shape).
Test vectors	None published for this operator.
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.166. SequenceMap

Applies a sub-graph to each sample in the input sequence(s).

Inputs can be either tensors or sequences, with the exception of the first input which must be a sequence. The length of the first input sequence will determine the number of samples in the outputs. Any other sequence inputs should have the same number of samples. The number of inputs and outputs, should match the one of the subgraph.

For each i -th element in the output, a sample will be extracted from the input sequence(s) at the i -th position and the sub-graph will be applied to it. The outputs will contain the outputs of the sub-graph for each sample, in the same order as in the input.

This operator assumes that processing each sample is independent and could be executed in parallel or in any order. Users cannot expect any specific ordering in which each subgraph is computed.

Domain	ai.onnx
Since version	17
Inputs (1 — unbounded)	input_sequence : S — Input sequence. additional_inputs (variadic, heterogeneous) : V — Additional inputs to the graph
Outputs (1 — unbounded)	out_sequence (variadic, heterogeneous) : S — Output sequence(s)
Attributes	body : graph (required) — The graph to be run for each sample in the sequence(s). It should have as many inputs and outputs as inputs and outputs to the SequenceMap function.
Type constraints	S : seq(tensor(uint8)), seq(tensor(uint16)), seq(tensor(uint32)), seq(tensor(uint64)), seq(tensor(int8)), seq(tensor(int16)), seq(tensor(int32)), seq(tensor(int64)), seq(tensor(float16)), seq(tensor(float)), seq(tensor(double)), seq(tensor(string)), seq(tensor(bool)), seq(tensor(complex64)), seq(tensor(complex128)) — Constrain input types to any sequence type. V : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128), seq(tensor(uint8)), seq(tensor(uint16)), seq(tensor(uint32)), seq(tensor(uint64)), seq(tensor(int8)), seq(tensor(int16)), seq(tensor(int32)), seq(tensor(int64)), seq(tensor(float16)), seq(tensor(float)), seq(tensor(double)), seq(tensor(string)), seq(tensor(bool)), seq(tensor(complex64)), seq(tensor(complex128)) — Constrain to any tensor or sequence type.
Test vectors	test_sequence_map_add_1_sequence_1_tensor, test_sequence_map_add_2_sequences, test_sequence_map_extract_shapes, test_sequence_map_identity_1_sequence, test_sequence_map_identity_1_sequence_1_tensor, test_sequence_map_identity_2_sequences
Not supplied by the	Shape inference, determinism, errors. See the editorial note at the head of this clause.

upstream
source

5.167. Shape

Takes a tensor as input and outputs an 1D int64 tensor containing the shape of the input tensor. Optional attributes start and end can be used to compute a slice of the input tensor's shape. If start axis is omitted, the slice starts from axis 0. The end axis, if specified, is exclusive (and the returned value will not include the size of that axis). If the end axis is omitted, the axes upto the last one will be included. Negative axes indicate counting back from the last axis. Note that axes will be clamped to the range $[0, r]$, where r is the rank of the input tensor if they are out-of-range (after adding r in the case of negative axis). Thus, specifying any end value $> r$ is equivalent to specifying an end value of r , and specifying any start value $< -r$ is equivalent to specifying a start value of 0. If $\text{start} > \text{end}$, the result will be an empty shape.

Examples:

Input tensor with shape: [2, 3, 4]
No attributes specified.
Output: [2, 3, 4]

Input tensor with shape: [2, 3, 4]
start: -1
Output: [4]

Input tensor with shape: [2, 3, 4]
end: -1
Output: [2, 3]

Input tensor with shape: [2, 3, 4]
start: 1
end: 2
Output: [3]

Domain	ai.onnx
Since version	25
Earlier versions	1, 13, 15, 19, 21, 23, 24
Inputs	data (non-differentiable) : T — An input tensor.
Outputs	shape (non-differentiable) : T1 — Shape of the input tensor
Attributes	end : int — (Optional) Ending axis for slicing the shape. Negative value means counting dimensions from the back. If omitted, sizes of all axes upto (including) the last one will be included. start : int (default is 0) — (Optional) Starting axis for slicing the shape. Default value is 0. Negative value means counting dimensions from the back.
Type constraints	T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128), tensor(float8e4m3fn), tensor(float8e4m3fnuz), tensor(float8e5m2), tensor(float8e5m2fnuz), tensor(uint4), tensor(int4), tensor(float4e2m1), tensor(float8e8m0), tensor(uint2), tensor(int2) — Input tensor can be of arbitrary type.

T1 : tensor(int64) — Constrain output to int64 tensor.

Test vectors None published for this operator.

Not supplied by the upstream source Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.168. Shrink

Shrink takes one input data (Tensor<numeric>) and produces one Tensor output, having same datatype and shape with input. It has two attributes, lambd and bias. The formula of this operator is: If $x < -\text{lambd}$, $y = x + \text{bias}$; If $x > \text{lambd}$, $y = x - \text{bias}$; Otherwise, $y = 0$.

Domain ai.onnx

Since version 9

Inputs input (differentiable) : T—The input data as Tensor.

Outputs output (differentiable) : T—The output.

Attributes bias : float (default is 0.0) — The bias value added to output. Default is 0.
lambd : float (default is 0.5) — The lambd value for the Shrink formulation. Default is 0.5.

Type constraints T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(float16), tensor(float), tensor(double) — Constrain input to only numeric types.

Test vectors test_shrink_hard, test_shrink_soft

Not supplied by the upstream source Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.169. Sigmoid

Sigmoid takes one input data (Tensor<T>) and produces one output data (Tensor<T>) where the sigmoid function, $y = 1 / (1 + \exp(-x))$, is applied to the tensor elementwise.

Domain ai.onnx

Since version 13

Earlier versions 1, 6

Inputs X (differentiable) : T—Input tensor

Outputs Y (differentiable) : T—Output tensor

Attributes None.

Type constraints	$T : \text{tensor}(\text{float16}), \text{tensor}(\text{float}), \text{tensor}(\text{double}), \text{tensor}(\text{bfloat16})$ — Constrain input and output types to float tensors.
Test vectors	<code>test_sigmoid_example</code> , <code>test_sigmoid</code>
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.170. Sign

Calculate the sign of the given input tensor element-wise. If input > 0, output 1. if input < 0, output -1. if input == 0, output 0.

Domain	<code>ai.onnx</code>
Since version	13
Earlier versions	9
Inputs	<code>input</code> (non-differentiable) : T — Input tensor
Outputs	<code>output</code> (non-differentiable) : T — The sign of the input tensor computed element-wise. It has the same shape and type of the input.
Attributes	None.
Type constraints	$T : \text{tensor}(\text{uint8}), \text{tensor}(\text{uint16}), \text{tensor}(\text{uint32}), \text{tensor}(\text{uint64}), \text{tensor}(\text{int8}), \text{tensor}(\text{int16}), \text{tensor}(\text{int32}), \text{tensor}(\text{int64}), \text{tensor}(\text{float16}), \text{tensor}(\text{float}), \text{tensor}(\text{double}), \text{tensor}(\text{bfloat16})$ — Constrain input and output types to all numeric tensors.
Test vectors	<code>test_sign</code>
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.171. Sin

Calculates the sine of the given input tensor, element-wise.

Domain	<code>ai.onnx</code>
Since version	22
Earlier versions	7
Inputs	<code>input</code> (differentiable) : T — Input tensor
Outputs	<code>output</code> (differentiable) : T — The sine of the input tensor computed element-wise
Attributes	None.

Type constraints	$T : \text{tensor}(\text{bfloat16}), \text{tensor}(\text{float16}), \text{tensor}(\text{float}), \text{tensor}(\text{double})$ — Constrain input and output types to float tensors.
Test vectors	<code>test_sin_example</code> , <code>test_sin</code>
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.172. Sinh

Calculates the hyperbolic sine of the given input tensor element-wise.

Domain	<code>ai.onnx</code>
Since version	22
Earlier versions	9
Inputs	<code>input (differentiable)</code> : T — Input tensor
Outputs	<code>output (differentiable)</code> : T — The hyperbolic sine values of the input tensor computed element-wise
Attributes	None.
Type constraints	$T : \text{tensor}(\text{bfloat16}), \text{tensor}(\text{float16}), \text{tensor}(\text{float}), \text{tensor}(\text{double})$ — Constrain input and output types to float tensors.
Test vectors	<code>test_sinh_example</code> , <code>test_sinh</code>
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.173. Size

Takes a tensor as input and outputs a int64 scalar that equals to the total number of elements of the input tensor.

Domain	<code>ai.onnx</code>
Since version	25
Earlier versions	1, 13, 19, 21, 23, 24
Inputs	<code>data (non-differentiable)</code> : T — An input tensor.
Outputs	<code>size (non-differentiable)</code> : T_1 — Total number of elements of the input tensor
Attributes	None.
Type constraints	$T : \text{tensor}(\text{uint8}), \text{tensor}(\text{uint16}), \text{tensor}(\text{uint32}), \text{tensor}(\text{uint64}), \text{tensor}(\text{int8}), \text{tensor}(\text{int16}), \text{tensor}(\text{int32}), \text{tensor}(\text{int64}), \text{tensor}(\text{bfloat16}), \text{tensor}(\text{float16}), \text{tensor}(\text{float}), \text{tensor}(\text{double}), \text{tensor}(\text{string})$

`tensor(bool), tensor(complex64), tensor(complex128),`
`tensor(float8e4m3fn), tensor(float8e4m3fnuz),`
`tensor(float8e5m2), tensor(float8e5m2fnuz), tensor(uint4),`
`tensor(int4), tensor(float4e2m1), tensor(float8e8m0),`
`tensor(uint2), tensor(int2)` — Input tensor can be of arbitrary type.
`T1 : tensor(int64)` — Constrain output to int64 tensor, which should be a scalar though.

Test vectors `test_size_example, test_size`

Not supplied by the upstream source Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.174. Slice

Produces a slice of the input tensor along multiple axes. Similar to numpy: <https://numpy.org/doc/stable/user/basics.indexing.html?highlight=slice#slicing-and-striding>

Slice uses the starts, ends, axes and steps inputs to select a sub-tensor of its input data tensor.

An effective `starts[i]`, `ends[i]`, and `steps[i]` must be computed for each `i` in `[0, ... r-1]` where `r = rank(input)` as follows:

If axes are omitted, they are set to `[0, ..., r-1]`. If steps are omitted, they are set to `[1, ..., 1]` of length `len(starts)`

The effective values are initialized as `starts[i] = 0`, `ends[i] = dims[i]` where `dims` are the dimensions of input and `steps[i] = 1`.

All negative elements of axes are made non-negative by adding `r` to them, where `r = rank(input)`.

All negative values in `starts[i]` and `ends[i]` have `dims[axes[i]]` added to them, where `dims` are the dimensions of input. Then `starts[axes[i]]` is clamped to range `[0, dims[axes[i]]]` for positive stepping, or to range `[0, dims[axes[i]]-1]` for negative stepping.

The clamping for the adjusted `ends[i]` depends on the sign of `steps[i]` and must accommodate copying 0 through `dims[axes[i]]` elements, so for positive stepping `ends[axes[i]]` is clamped to `[0, dims[axes[i]]]`, while for negative stepping it is clamped to `[-1, dims[axes[i]]-1]`.

Finally, `steps[axes[i]] = steps[i]`.

For slicing to the end of a dimension with unknown size, it is recommended to pass in `INT_MAX` when slicing forward and `INT_MIN` when slicing backward.

Example 1:

```

data = [
    [1, 2, 3, 4],
    [5, 6, 7, 8],
]
axes = [0, 1]
starts = [1, 0]
ends = [2, 3]
steps = [1, 2]
result = [
    [5, 7],
  
```

]

Example 2:

```
data = [
    [1, 2, 3, 4],
    [5, 6, 7, 8],
]
starts = [0, 1]
ends = [-1, 1000]
result = [
    [2, 3, 4],
]
```

Domain ai.onnx

Since version 13

Earlier versions 1, 10, 11

Inputs (3 — 5) data (differentiable) : T — Tensor of data to extract slices from.
 starts (non-differentiable) : Tind — 1-D tensor of starting indices of corresponding axis in axes
 ends (non-differentiable) : Tind — 1-D tensor of ending indices (exclusive) of corresponding axis in axes
 axes (optional, non-differentiable) : Tind — 1-D tensor of axes that starts and ends apply to. Negative value means counting dimensions from the back. Accepted range is [-r, r-1] where r = rank(data). Behavior is undefined if an axis is repeated.
 steps (optional, non-differentiable) : Tind — 1-D tensor of slice step of corresponding axis in axes. Negative value means slicing backward. 'steps' cannot be 0. Defaults to 1s.

Outputs output (differentiable) : T — Sliced data tensor.

Attributes None.

Type constraints T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128) — Constrain input and output types to all tensor types.
 Tind : tensor(int32), tensor(int64) — Constrain indices to integer types

Test vectors test_slice, test_slice_default_axes, test_slice_default_steps, test_slice_end_out_of_bounds, test_slice_neg, test_slice_neg_steps, test_slice_negative_axes, test_slice_start_out_of_bounds

Not supplied by the upstream source Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.175. Softmax

The operator computes the normalized exponential values for the given input:

$\text{Softmax}(\text{input}, \text{axis}) = \text{Exp}(\text{input}) / \text{ReduceSum}(\text{Exp}(\text{input}), \text{axis}=\text{axis}, \text{keepdims}=1)$

The “axis” attribute indicates the dimension along which Softmax will be performed. The output tensor has the same shape and contains the Softmax values of the corresponding input.

Domain	ai.onnx
Since version	13
Earlier versions	1, 11
Inputs	input (differentiable) : T — The input tensor of rank $\geq$ axis.
Outputs	output (differentiable) : T — The output values with the same shape as the input tensor.
Attributes	axis : int (default is -1) — Describes the dimension Softmax will be performed on. Negative value means counting dimensions from the back. Accepted range is $[-r, r-1]$ where $r = \text{rank}(\text{input})$.
Type constraints	T : tensor(float16), tensor(float), tensor(double), tensor(bfloat16) — Constrain input and output types to float tensors.
Test vectors	test_softmax_example, test_softmax_large_number, test_softmax_axis_0, test_softmax_axis_1, test_softmax_axis_2, test_softmax_negative_axis, test_softmax_default_axis
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.176. SoftmaxCrossEntropyLoss

Loss function that measures the softmax cross entropy between ‘scores’ and ‘labels’. This operator first computes a loss tensor whose shape is identical to the labels input. If the input is 2-D with shape (N, C), the loss tensor may be a N-element vector $L = (l_1, l_2, \dots, l_N)$. If the input is N-D tensor with shape (N, C, D1, D2, ..., Dk), the loss tensor L may have (N, D1, D2, ..., Dk) as its shape and $L[i][j_1][j_2] \dots [j_k]$ denotes a scalar element in L. After L is available, this operator can optionally do a reduction operator.

- shape(scores): (N, C) where C is the number of classes, or (N, C, D1, D2, ..., Dk), with $K \geq 1$ in case of K-dimensional loss.
- shape(labels): (N) where each value is $0 \leq \text{labels}[i] \leq C-1$, or (N, D1, D2, ..., Dk), with $K \geq 1$ in case of K-dimensional loss.

The loss for one sample, l_i , can be calculated as follows:

$l[i][d1][d2] \dots [dk] = -y[i][c][d1][d2] \dots [dk]$, where i is the index of classes.
or

$l[i][d1][d2] \dots [dk] = -y[i][c][d1][d2] \dots [dk] * \text{weights}[c]$, if ‘weights’ is provided.

loss is zero for the case when label-value equals ignore_index.

$l[i][d1][d2] \dots [dk] = 0$, when $\text{labels}[n][d1][d2] \dots [dk] = \text{ignore_index}$
where:

```

p = Softmax(scores)
y = Log(p)
c = labels[i][d1][d2]...[dk]

```

Finally, L is optionally reduced:

- If reduction = 'none', the output is L with shape (N, D1, D2, ..., Dk).
- If reduction = 'sum', the output is scalar: Sum(L).
- If reduction = 'mean', the output is scalar: ReduceMean(L), or if weight is provided: ReduceSum(L) / ReduceSum(W), where tensor W is of shape (N, D1, D2, ..., Dk) and $W[n][d1][d2]...[dk] = \text{weights}[\text{labels}[i][d1][d2]...[dk]]$.

Domain ai.onnx

Since 13

version

Earlier 12

versions

Inputs (2 — 3) scores (differentiable) : T — The predicted outputs with shape [batch_size, class_size], or [batch_size, class_size, D1, D2, ..., Dk], where K is the number of dimensions.

labels (non-differentiable) : Tind — The ground truth output tensor, with shape [batch_size], or [batch_size, D1, D2, ..., Dk], where K is the number of dimensions. Labels element value shall be in range of [0, C). If ignore_index is specified, it may have a value outside [0, C) and the label values should either be in the range [0, C) or have the value ignore_index.

weights (optional, non-differentiable) : T — A manual rescaling weight given to each class. If given, it has to be a 1D Tensor assigning weight to each of the classes. Otherwise, it is treated as if having all ones.

Outputs (1 — 2) output (differentiable) : T — Weighted loss float Tensor. If reduction is 'none', this has the shape of [batch_size], or [batch_size, D1, D2, ..., Dk] in case of K-dimensional loss. Otherwise, it is a scalar.

log_prob (optional, differentiable) : T — Log probability tensor. If the output of softmax is prob, its value is log(prob).

Attributes ignore_index : int — Specifies a target value that is ignored and does not contribute to the input gradient. It's an optional value.

reduction : string (default is mean) — Type of reduction to apply to loss: none, sum, mean(default). 'none': no reduction will be applied, 'sum': the output will be summed. 'mean': the sum of the output will be divided by the number of elements in the output.

Type T : tensor(float16), tensor(float), tensor(double),

constraints tensor(bfloat16) — Constrain input and output types to float tensors.

Tind : tensor(int32), tensor(int64) — Constrain target to integer types

Test vectors test_sce_NCd1_mean_weight_negative_ii, test_sce_NCd1_mean_weight_negative_ii_log_prob, test_sce_NCd1d2d3_none_no_weight_negative_ii, test_sce_NCd1d2d3_none_no_weight_negative_ii_log_prob, test_sce_NCd1d2d3_sum_weight_high_ii, test_sce_NCd1d2d3_sum_weight_high_ii_log_prob, test_sce_NCd1d2d3d4d5_mean_weight, test_sce_NCd1d2d3d4d5_mean_weight_log_prob, test_sce_NCd1d2d3d4d5_none_no_weight, test_sce_NCd1d2d3d4d5_none_no_weight_log_prob, test_sce_mean, test_sce_mean_3d, test_sce_mean_3d_log_prob, test_sce_mean_log_prob, test_sce_mean_no_weight_ii, test_sce_mean_no_weight_ii_3d, test_sce_mean_no_weight_ii_3d_log_prob, test_sce_mean_no_weight_ii_4d, test_sce_mean_no_weight_ii_4d_log_prob, test_sce_mean_no_weight_ii_log_prob, test_sce_mean_weight, test_sce_mean_weight_ii, test_sce_mean_weight_ii_3d, test_sce_mean_weight_ii_3d_log_prob, test_sce_mean_weight_ii_4d, test_sce_mean_weight_ii_4d_log_prob, test_sce_mean_weight_ii_log_prob, test_sce_mean_weight_log_prob, test_sce_none, test_sce_none_log_prob, test_sce_none_weights,

	test_sce_none_weights_log_prob, test_sce_sum, test_sce_sum_log_prob
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.177. Softplus

Softplus takes one input data (Tensor<T>) and produces one output data (Tensor<T>) where the softplus function, $y = \ln(\exp(x) + 1)$, is applied to the tensor elementwise.

Domain	ai.onnx
Since version	22
Earlier versions	1
Inputs	X (differentiable) : T—Input tensor
Outputs	Y (differentiable) : T—Output tensor
Attributes	None.
Type constraints	T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain input and output types to float tensors.
Test vectors	test_softplus_example, test_softplus
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.178. Softsign

Table 8 — Table from the upstream description of Softsign

Calculates the softsign ($x/(1+|x|)$) of the given input tensor element-wise.

Domain	ai.onnx
Since version	22
Earlier versions	1
Inputs	input (differentiable) : T—Input tensor
Outputs	output (differentiable) : T—[cols=3*] === The softsign ($x/(1+ x)$) values of the input tensor computed element-wise ===
Attributes	None.
Type constraints	T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain input and output types to float tensors.

Test vectors	<code>test_softsign_example, test_softsign</code>
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.179. SpaceToDepth

SpaceToDepth rearranges blocks of spatial data into depth. More specifically, this op outputs a copy of the input tensor where values from the height and width dimensions are moved to the depth dimension. `mode` determines whether blocks are ordered depth-column-row (DCR, the default) or column-row-depth (CRD).

Domain	<code>ai.onnx</code>
Since version	28
Earlier versions	1, 13
Inputs	<code>input</code> (differentiable) : T — Input tensor of [N,C,H,W], where N is the batch axis, C is the channel or depth, H is the height and W is the width.
Outputs	<code>output</code> (differentiable) : T — Output tensor of [N, C * blocksize * blocksize, H/blocksize, W/blocksize].
Attributes	<code>blocksize</code> : int (required) — Blocks of [blocksize, blocksize] are moved. <code>mode</code> : string (default is DCR) — DCR (default) for depth-column-row order re-arrangement. Use CRD for column-row-depth order.
Type constraints	T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128) — Constrain input and output types to all tensor types.
Test vectors	<code>test_spacetodepth_crd_mode_example, test_spacetodepth_dcr_mode_example, test_spacetodepth_example, test_spacetodepth</code>
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.180. Split

Split a tensor into a list of tensors, along the specified 'axis'. Either input 'split' or the attribute 'num_outputs' should be specified, but not both. If the attribute 'num_outputs' is specified, then the tensor is split into equal sized parts. If the tensor is not evenly splittable into num_outputs, the last chunk will be smaller. If the input 'split' is specified, it indicates the sizes of each output in the split.

Domain	<code>ai.onnx</code>
Since version	18

Earlier versions	1, 2, 11, 13
Inputs (1 — 2)	input (differentiable) : T — The tensor to split split (optional, non-differentiable) : tensor(int64) — Optional length of each output. Values should be ≥ 0 . Sum of the values must be equal to the dim value at 'axis' specified.
Outputs (1 — unbounded)	outputs (variadic, differentiable) : T — One or more outputs forming list of tensors after splitting
Attributes	axis : int (default is 0) — Which axis to split on. A negative value means counting dimensions from the back. Accepted range is $[-\text{rank}, \text{rank}-1]$ where $r = \text{rank}(\text{input})$. num_outputs : int — Number of outputs to split parts of the tensor into. If the tensor is not evenly splittable the last chunk will be smaller.
Type constraints	T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128) — Constrain input and output types to all tensor types.
Test vectors	test_split_equal_parts_1d_opset13, test_split_variable_parts_1d_opset13, test_split_equal_parts_1d_opset18, test_split_variable_parts_1d_opset18, test_split_1d_uneven_split_opset18, test_split_equal_parts_2d_opset13, test_split_variable_parts_2d_opset13, test_split_equal_parts_2d, test_split_variable_parts_2d_opset18, test_split_2d_uneven_split_opset18, test_split_equal_parts_default_axis_opset13, test_split_variable_parts_default_axis_opset13, test_split_equal_parts_default_axis_opset18, test_split_variable_parts_default_axis_opset18, test_split_zero_size_splits_opset13, test_split_zero_size_splits_opset18
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.181. SplitToSequence

Split a tensor into a sequence of tensors, along the specified 'axis'. Lengths of the parts can be specified using the optional argument 'split'. If the argument 'split' is not specified, a default scalar value of 1 is used as the value of 'split'. 'split' must contain only positive numbers. 'split' is either a scalar (tensor of empty shape), or a 1-D tensor. If 'split' is a scalar, then 'input' will be split into chunks all of size 'split' if possible. The last chunk alone may be smaller than 'split' if the 'input' size along the given axis 'axis' is not divisible by 'split'. If 'split' is a 1-dimensional tensor, the input tensor is split into 'size(split)' chunks, with lengths of the parts on 'axis' specified in 'split'. In this scenario, the sum of entries in 'split' must be equal to the dimension size of input tensor on 'axis'.

Domain	ai.onnx
Since version	24
Earlier versions	11
Inputs (1 — 2)	input : T — The tensor to split

	<code>split</code> (optional) : I — Length of each output. It can be either a scalar(tensor of empty shape), or a 1-D tensor. All values must be ≥ 0 .
Outputs	<code>output_sequence</code> : S — One or more outputs forming a sequence of tensors after splitting
Attributes	<code>axis</code> : int (default is 0) — Which axis to split on. A negative value means counting dimensions from the back. Accepted range is $[-\text{rank}, \text{rank}-1]$. <code>keepdims</code> : int (default is 1) — Keep the split dimension or not. Default 1, which means we keep split dimension. If input 'split' is specified, this attribute is ignored.
Type constraints	T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128) — Constrain input types to all tensor types. I : tensor(int32), tensor(int64) — Constrain split size to integral tensor. S : seq(tensor(uint8)), seq(tensor(uint16)), seq(tensor(uint32)), seq(tensor(uint64)), seq(tensor(int8)), seq(tensor(int16)), seq(tensor(int32)), seq(tensor(int64)), seq(tensor(bfloat16)), seq(tensor(float16)), seq(tensor(float)), seq(tensor(double)), seq(tensor(string)), seq(tensor(bool)), seq(tensor(complex64)), seq(tensor(complex128)) — Constrain output types to all tensor types.
Test vectors	<code>test_split_to_sequence_nokeepdims</code> , <code>test_split_to_sequence_1</code> , <code>test_split_to_sequence_2</code>
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.182. Sqrt

Square root takes one input data (Tensor<T>) and produces one output data (Tensor<T>) where the square root is, $y = x^{0.5}$, is applied to the tensor elementwise. If x is negative, then it will return NaN.

Domain	ai.onnx
Since version	13
Earlier versions	1, 6
Inputs	X (differentiable) : T — Input tensor
Outputs	Y (differentiable) : T — Output tensor
Attributes	None.

Type constraints	$T : \text{tensor}(\text{float16}), \text{tensor}(\text{float}), \text{tensor}(\text{double}), \text{tensor}(\text{bfloat16})$ — Constrain input and output types to float tensors.
Test vectors	<code>test_sqrt_example</code> , <code>test_sqrt</code>
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.183. Squeeze

Remove single-dimensional entries from the shape of a tensor. Takes an input axes with a list of axes to squeeze. If axes is not provided, all the single dimensions will be removed from the shape. If an axis is selected with shape entry not equal to one, an error is raised.

Domain	<code>ai.onnx</code>
Since version	25
Earlier versions	1, 11, 13, 21, 23, 24
Inputs (1 — 2)	<code>data</code> (differentiable) : T — Tensors with at least $\max(\text{dims})$ dimensions. <code>axes</code> (optional, non-differentiable) : $\text{tensor}(\text{int64})$ — 1D tensor of integers indicating the dimensions to squeeze. Negative value means counting dimensions from the back. Accepted range is $[-r, r-1]$ where $r = \text{rank}(\text{data})$.
Outputs	<code>squeezed</code> (differentiable) : T — Reshaped tensor with same data as input.
Attributes	None.
Type constraints	$T : \text{tensor}(\text{uint8}), \text{tensor}(\text{uint16}), \text{tensor}(\text{uint32}), \text{tensor}(\text{uint64}), \text{tensor}(\text{int8}), \text{tensor}(\text{int16}), \text{tensor}(\text{int32}), \text{tensor}(\text{int64}), \text{tensor}(\text{bfloat16}), \text{tensor}(\text{float16}), \text{tensor}(\text{float}), \text{tensor}(\text{double}), \text{tensor}(\text{string}), \text{tensor}(\text{bool}), \text{tensor}(\text{complex64}), \text{tensor}(\text{complex128}), \text{tensor}(\text{float8e4m3fn}), \text{tensor}(\text{float8e4m3fnuz}), \text{tensor}(\text{float8e5m2}), \text{tensor}(\text{float8e5m2fnuz}), \text{tensor}(\text{uint4}), \text{tensor}(\text{int4}), \text{tensor}(\text{float4e2m1}), \text{tensor}(\text{float8e8m0}), \text{tensor}(\text{uint2}), \text{tensor}(\text{int2})$ — Constrain input and output types to all tensor types up to IRv13.
Test vectors	<code>test_squeeze</code> , <code>test_squeeze_negative_axes</code>
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.184. StringConcat

StringConcat concatenates string tensors elementwise (with NumPy-style broadcasting support)

Domain	<code>ai.onnx</code>
--------	----------------------

Since version	20
Inputs	X (non-differentiable) : T — Tensor to prepend in concatenation Y (non-differentiable) : T — Tensor to append in concatenation
Outputs	Z (non-differentiable) : T — Concatenated string tensor
Attributes	None.
Type constraints	T : tensor(string) — Inputs and outputs must be UTF-8 strings
Test vectors	test_string_concat, test_string_concat_broadcasting, test_string_concat_zero_dimensional, test_string_concat_empty_string, test_string_concat_utf8
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.185. StringNormalizer

StringNormalization performs string operations for basic cleaning. This operator has only one input (denoted by X) and only one output (denoted by Y). This operator first examines the elements in the X, and removes elements specified in “stopwords” attribute. After removing stop words, the intermediate result can be further lowercased, uppercased, or just returned depending the “case_change_action” attribute. This operator only accepts [C]- and [1, C]-tensor. If all elements in X are dropped, the output will be the empty value of string tensor with shape [1] if input shape is [C] and shape [1, 1] if input shape is [1, C].

Domain	ai.onnx
Since version	10
Inputs	X : tensor(string) — UTF-8 strings to normalize
Outputs	Y : tensor(string) — UTF-8 Normalized strings
Attributes	case_change_action : string (default is NONE) — string enum that cases output to be lowercased/uppercases/unchanged. Valid values are “LOWER”, “UPPER”, “NONE”. Default is “NONE” is_case_sensitive : int (default is 0) — Boolean. Whether the identification of stop words in X is case-sensitive. Default is false locale : string — Environment dependent string that denotes the locale according to which output strings needs to be upper/lowercased. Default en_US or platform specific equivalent as decided by the implementation. stopwords : list of strings — List of stop words. If not set, no word would be removed from X.
Type constraints	None.
Test vectors	test_stnormalizer_export_monday_casesensitive_lower, test_stnormalizer_export_monday_casesensitive_nochangecase, test_stnormalizer_export_monday_casesensitive_upper, test_stnormalizer_export_monday_empty_output, test_stnormalizer_

export_monday_insensitive_upper_twodim, test_strnormalizer_nostopwords_nochangecase

Not supplied by the upstream source Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.186. StringSplit

StringSplit splits a string tensor's elements into substrings based on a delimiter attribute and a maxsplit attribute.

The first output of this operator is a tensor of strings representing the substrings from splitting each input string on the delimiter substring. This tensor has one additional rank compared to the input tensor in order to store the substrings for each input element (where the input tensor is not empty). Note that, in order to ensure the same number of elements are present in the final dimension, this tensor will pad empty strings as illustrated in the examples below. Consecutive delimiters are not grouped together and are deemed to delimit empty strings, except if the delimiter is unspecified or is the empty string (""). In the case where the delimiter is unspecified or the empty string, consecutive whitespace characters are regarded as a single separator and leading or trailing whitespace is removed in the output.

The second output tensor represents the number of substrings generated. maxsplit can be used to limit the number of splits performed — after the maxsplitth split if the string is not fully split, the trailing suffix of input string after the final split point is also added. For elements where fewer splits are possible than specified in maxsplit, it has no effect.

Domain	ai.onnx
Since version	20
Inputs	X (non-differentiable) : T1 — Tensor of strings to split.
Outputs	Y (non-differentiable) : T2 — Tensor of substrings representing the outcome of splitting the strings in the input on the delimiter. Note that to ensure the same number of elements are present in the final rank, this tensor will pad any necessary empty strings. Z (non-differentiable) : T3 — The number of substrings generated for each input element.
Attributes	delimiter : string — Delimiter to split on. If left unset or set to the empty string (""), the input is split on consecutive whitespace. maxsplit : int — Maximum number of splits (from left to right). If left unset (or if the number of possible splits are less than maxsplit), it will make as many splits as possible. Note that the maximum possible number of substrings returned with maxsplit specified is maxsplit+1 since the remaining suffix after the maxsplitth split is included in the output.
Type constraints	T1 : tensor(string) — The input must be a UTF-8 string tensor T2 : tensor(string) — Tensor of substrings. T3 : tensor(int64) — The number of substrings generated.
Test vectors	test_string_split_basic, test_string_split_consecutive_delimiters, test_string_split_empty_tensor, test_string_split_maxsplit

Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.
-------------------------------------	--

5.187. Sub

Performs element-wise binary subtraction (with Numpy-style broadcasting support).

This operator supports **multidirectional (i.e., Numpy-style) broadcasting**; for more details please check the doc.

(Opset 14 change): Extend supported types to include uint8, int8, uint16, and int16.

Domain	ai.onnx
Since version	14
Earlier versions	1, 6, 7, 13
Inputs	A (differentiable) : T — First operand. B (differentiable) : T — Second operand.
Outputs	C (differentiable) : T — Result, has same element type as two inputs
Attributes	None.
Type constraints	T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(float16), tensor(float), tensor(double), tensor(bfloat16) — Constrain input and output types to all numeric tensors.
Test vectors	test_sub_example, test_sub, test_sub_int8, test_sub_int16, test_sub_uint8, test_sub_uint16, test_sub_uint32, test_sub_uint64, test_sub_bcast
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.188. Sum

Element-wise sum of each of the input tensors (with Numpy-style broadcasting support). All inputs and outputs must have the same data type. This operator supports **multidirectional (i.e., Numpy-style) broadcasting**; for more details please check the doc.

Domain	ai.onnx
Since version	13
Earlier versions	1, 6, 8
Inputs (1 — unbounded)	data_0 (variadic, differentiable) : T — List of tensors for sum.
Outputs	sum (differentiable) : T — Output tensor.

Attributes	None.
Type constraints	$T : \text{tensor}(\text{float16}), \text{tensor}(\text{float}), \text{tensor}(\text{double}), \text{tensor}(\text{bfloat16})$ — Constrain input and output types to float tensors.
Test vectors	test_sum_example, test_sum_one_input, test_sum_two_inputs
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.189. SwiGLU

SwiGLU is a gated activation that takes two inputs, a gate A and a linear (value) input B, and produces one output Y. It applies the Swish activation to the gate and multiplies the result elementwise by the linear input:

$$Y = \text{Swish_alpha}(A) * B$$

The gate activation Swish_alpha is exactly the Swish operator with the same alpha, i.e. $\text{Swish_alpha}(a) = a * \text{Sigmoid}(\text{alpha} * a)$. Inputs A and B must have identical shapes; broadcasting is not applied and the output Y has the same shape as the inputs.

Exporters typically produce A and B in one of two ways: for the common two-projection form (e.g. Llama's gate_proj/up_proj) wire the two projection outputs directly to A (gate) and B (value); for a fused/packed single projection, split it upstream into A and B with Split (contiguous layout) or Slice/Gather (interleaved layout).

Domain	ai.onnx
Since version	28
Inputs	A (differentiable) : T — Gate input tensor B (differentiable) : T — Linear (value) input tensor
Outputs	Y (differentiable) : T — Output tensor
Attributes	alpha : float (default is 1.0) — Coefficient that scales the gate input inside the sigmoid of the Swish activation. The default value is 1.0.
Type constraints	$T : \text{tensor}(\text{bfloat16}), \text{tensor}(\text{float16}), \text{tensor}(\text{float}), \text{tensor}(\text{double})$ — Constrain input and output types to float tensors.
Test vectors	test_swiglu_alpha, test_swiglu_float16, test_swiglu
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.190. Swish

Swish function takes one input data (Tensor<T>) and produces one output data (Tensor<T>) of the same shape, where $\text{Swish}(x) = x * \text{sigmoid}(\text{alpha} * x)$.

Domain	ai.onnx
--------	---------

Since version	24
Inputs	X (differentiable) : T —Input tensor
Outputs	Y (differentiable) : T —Output tensor
Attributes	α : float (default is 1.0) — Coefficient to multiply with input before sigmoid.
Type constraints	T : tensor(float16), tensor(float), tensor(bfloat16), tensor(double) — Constrain input and output types to float tensors.
Test vectors	test_swish
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.191. Tan

Calculates the tangent of the given input tensor, element-wise.

Domain	ai.onnx
Since version	22
Earlier versions	7
Inputs	input (differentiable) : T —Input tensor
Outputs	output (differentiable) : T —The tangent of the input tensor computed element-wise
Attributes	None.
Type constraints	T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain input and output types to float tensors.
Test vectors	test_tan_example, test_tan
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.192. Tanh

Calculates the hyperbolic tangent of the given input tensor element-wise.

Domain	ai.onnx
Since version	13
Earlier versions	1, 6
Inputs	input (differentiable) : T —Input tensor

Outputs	output (differentiable) : T—The hyperbolic tangent values of the input tensor computed element-wise
Attributes	None.
Type constraints	T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain input and output types to float tensors.
Test vectors	test_tanh_example, test_tanh
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.193. TensorScatter

TensorScatter is a generic tensor update operation, motivated by the requirements for KV cache updates for Attention ops commonly found in LLMs. It is a functional operation that models an in-place update to a KV cache buffer.

The past and present cache tensors have the same shape (batch_size, D1, D2, ..., max_sequence_length, ..., Dn), with the sequence dimension (indicated by the axis attribute) being max_sequence_length, so the sizes of these tensors do not need to grow between iterations. The update tensor's shape only differs from the cache tensors in the sequence dimension: (batch_size, D1, D2, ..., sequence_length, ..., Dn), where sequence_length ≤ max_sequence_length.

The optional write_indices input indicates the write index for each sample in the batch, assumed to be zero if not provided. When the mode attribute is set to "circular", the write index is modulo max_sequence_length. The operation can be described using the following pseudocode:

```
for prefix_idx in np.ndindex(past_cache.shape[:axis]):
    batch_idx = prefix_idx[0]
    for sequence_idx in range(sequence_length):
        cache_sequence_idx = write_indices[batch_idx] + sequence_idx
        if mode == "circular":
            cache_sequence_idx = cache_sequence_idx % max_sequence_length
        cache_idx = (*prefix_idx, cache_sequence_idx)
        update_idx = (*prefix_idx, sequence_idx)
        present_cache[cache_idx] = update[update_idx]
```

During the prefill phase of attention, only the first two inputs are needed. During the decode phase, write_indices is also needed so that the incoming key or value update can be appended after the last valid token for each sample in the batch.

Domain ai.onnx

Since version 24

Inputs (2 — 3)

- past_cache (differentiable) : T—Past state cache for key or value with shape (batch_size, D1, D2, ..., max_sequence_length, ..., Dn).
- update (differentiable) : T—New update tensor with shape (batch_size, D1, D2, ..., sequence_length, ..., Dn).
- write_indices (optional, non-differentiable) : tensor(int64) — Write indices for the incoming update tensor in the cache. Shape is (batch_size,). Assumed to be all zeros if not provided.

Outputs	<code>present_cache</code> (differentiable) : T — Updated cache. Same shape as <code>past_cache</code> .
Attributes	<code>axis</code> : int (default is -2) — Sequence dimension of the <code>past_cache</code> and update tensors. It cannot be 0 (the batch dimension). Default is -2. <code>mode</code> : string (default is <code>linear</code>) — Write mode of cache update. Supported modes include <code>linear</code> and <code>circular</code> . <code>linear</code> mode requires <code>write_indices+sequence_length<=max_sequence_length</code> . For <code>circular</code> mode, the updates happen in wrap-around fashion, ie, the update index is modulo <code>max_sequence_length</code>
Type constraints	T : <code>tensor(uint8)</code> , <code>tensor(uint16)</code> , <code>tensor(uint32)</code> , <code>tensor(uint64)</code> , <code>tensor(int8)</code> , <code>tensor(int16)</code> , <code>tensor(int32)</code> , <code>tensor(int64)</code> , <code>tensor(bfloat16)</code> , <code>tensor(float16)</code> , <code>tensor(float)</code> , <code>tensor(double)</code> , <code>tensor(string)</code> , <code>tensor(bool)</code> , <code>tensor(complex64)</code> , <code>tensor(complex128)</code> , <code>tensor(float8e4m3fn)</code> , <code>tensor(float8e4m3fnuz)</code> , <code>tensor(float8e5m2)</code> , <code>tensor(float8e5m2fnuz)</code> , <code>tensor(uint4)</code> , <code>tensor(int4)</code> , <code>tensor(float4e2m1)</code> , <code>tensor(float8e8m0)</code> — Constrain input and output types to any tensor type.
Test vectors	<code>test_tensortscatter</code> , <code>test_tensortscatter_3d</code> , <code>test_tensortscatter_circular</code>
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.194. TfIdfVectorizer

This transform extracts n-grams from the input sequence and save them as a vector. Input can be either a 1-D or 2-D tensor. For 1-D input, output is the n-gram representation of that input. For 2-D input, the output is also a 2-D tensor whose i-th row is the n-gram representation of the i-th input row. More specifically, if input shape is [C], the corresponding output shape would be [`max(ngram_indexes) + 1`]. If input shape is [N, C], this operator produces a [N, `max(ngram_indexes) + 1`]-tensor.

In contrast to standard n-gram extraction, here, the indexes of extracting an n-gram from the original sequence are not necessarily consecutive numbers. The discontinuity between indexes are controlled by the number of skips. If the number of skips is 2, we should skip two tokens when scanning through the original sequence. Let's consider an example. Assume that input sequence is [94, 17, 36, 12, 28] and the number of skips is 2. The associated 2-grams are [94, 12] and [17, 28] respectively indexed by [0, 3] and [1, 4]. If the number of skips becomes 0, the 2-grams generated are [94, 17], [17, 36], [36, 12], [12, 28] indexed by [0, 1], [1, 2], [2, 3], [3, 4], respectively.

The output vector (denoted by Y) stores the count of each n-gram; `Y[ngram_indexes[i]]` indicates the times that the i-th n-gram is found. The attribute `ngram_indexes` is used to determine the mapping between index i and the corresponding n-gram's output coordinate. If `pool_int64s` is [94, 17, 17, 36], `ngram_indexes` is [1, 0], `ngram_counts`=[0, 0], then the `Y[0]` (first element in Y) and `Y[1]` (second element in Y) are the counts of [17, 36] and [94, 17], respectively. An n-gram which cannot be found in `pool_strings/pool_int64s` should be ignored and has no effect on the output. Note that we may consider all skips up to S when generating the n-grams.

The examples used above are true if mode is "TF". If mode is "IDF", all the counts larger than 1 would be truncated to 1 and the i-th element in weights would be used to scale (by multiplication) the count of the i-th n-gram in pool. If mode is "TFIDF", this operator first computes the counts of all n-grams and then scale them by the associated values in the weights attribute.

Only one of `pool_strings` and `pool_int64s` can be set. If `pool_int64s` is set, the input should be an integer tensor. If `pool_strings` is set, the input must be a string tensor.

Domain	<code>ai.onnx</code>
Since version	9
Inputs	<code>X</code> (non-differentiable) : <code>T</code> — Input for n-gram extraction
Outputs	<code>Y</code> (non-differentiable) : <code>T1</code> — Ngram results
Attributes	<code>max_gram_length</code> : <code>int</code> (required) — Maximum n-gram length. If this value is 3, 3-grams will be used to generate the output. <code>max_skip_count</code> : <code>int</code> (required) — Maximum number of items (integers/strings) to be skipped when constructing an n-gram from <code>X</code>. If <code>max_skip_count=1</code>, <code>min_gram_length=2</code>, <code>max_gram_length=3</code>, this operator may generate 2-grams with <code>skip_count=0</code> and <code>skip_count=1</code>, and 3-grams with <code>skip_count=0</code> and <code>skip_count=1</code> <code>min_gram_length</code> : <code>int</code> (required) — Minimum n-gram length. If this value is 2 and <code>max_gram_length</code> is 3, output may contain counts of 2-grams and 3-grams. <code>mode</code> : <code>string</code> (required) — The weighting criteria. It can be one of "TF" (term frequency), "IDF" (inverse document frequency), and "TFIDF" (the combination of TF and IDF) <code>ngram_counts</code> : <code>list of ints</code> (required) — The starting indexes of 1-grams, 2-grams, and so on in pool. It is useful when determining the boundary between two consecutive collections of n-grams. For example, if <code>ngram_counts</code> is <code>[0, 17, 36]</code>, the first index (zero-based) of 1-gram/2-gram/3-gram in pool are 0/17/36. This format is essentially identical to CSR (or CSC) sparse matrix format, and we choose to use this due to its popularity. <code>ngram_indexes</code> : <code>list of ints</code> (required) — list of <code>int64s</code> (type: <code>AttributeProto::INTS</code>). This list is parallel to the specified '<code>pool_*</code>' attribute. The <code>i</code>-th element in <code>ngram_indexes</code> indicate the coordinate of the <code>i</code>-th n-gram in the output tensor. <code>pool_int64s</code> : <code>list of ints</code> — List of <code>int64</code> n-grams learned from the training set. Either this or <code>pool_strings</code> attributes must be present but not both. It's an 1-D tensor starting with the collections of all 1-grams and ending with the collections of n-grams. The <code>i</code>-th element in pool stores the n-gram that should be mapped to coordinate <code>ngram_indexes[i]</code> in the output vector. <code>pool_strings</code> : <code>list of strings</code> — List of strings n-grams learned from the training set. Either this or <code>pool_int64s</code> attributes must be present but not both. It's an 1-D tensor starting with the collections of all 1-grams and ending with the collections of n-grams. The <code>i</code>-th element in pool stores the n-gram that should be mapped to coordinate <code>ngram_indexes[i]</code> in the output vector. <code>weights</code> : <code>list of floats</code> — list of floats. This attribute stores the weight of each n-gram in pool. The <code>i</code>-th element in <code>weights</code> is the weight of the <code>i</code>-th n-gram in pool. Its length equals to the size of <code>ngram_indexes</code>. By default, <code>weights</code> is an all-one tensor. This attribute is used when <code>mode</code> is "IDF" or "TFIDF" to scale the associated word counts.
Type constraints	<code>T</code> : <code>tensor(string)</code>, <code>tensor(int32)</code>, <code>tensor(int64)</code> — Input is either string UTF-8 or <code>int32/int64</code> <code>T1</code> : <code>tensor(float)</code> — 1-D tensor of floats
Test vectors	<code>test_tfidfvectorizer_tf_batch_onlybigrams_skip0</code> , <code>test_tfidfvectorizer_tf_batch_onlybigrams_skip5</code> , <code>test_tfidfvectorizer_tf_batch_uniandbigrams_skip5</code> , <code>test_</code>

```
tfidfvectorizer_tf_only_bigrams_skip0, test_tfidfvectorizer_
tf_onlybigrams_levelempty, test_tfidfvectorizer_tf_
onlybigrams_skip5, test_tfidfvectorizer_tf_uniandbigrams_
skip5
```

Not supplied by the upstream source Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.195. ThresholdedRelu

ThresholdedRelu takes one input data (Tensor<T>) and produces one output data (Tensor<T>) where the rectified linear function, $y = x$ for $x > \alpha$, $y = 0$ otherwise, is applied to the tensor elementwise.

Domain	ai.onnx
Since version	22
Earlier versions	10
Inputs	X (differentiable) : T — Input tensor
Outputs	Y (differentiable) : T — Output tensor
Attributes	α : float (default is 1.0) — Threshold value
Type constraints	T : tensor(bfloat16), tensor(float16), tensor(float), tensor(double) — Constrain input and output types to float tensors.
Test vectors	test_thresholdedrelu_default, test_thresholdedrelu_example, test_thresholdedrelu
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.196. Tile

Constructs a tensor by tiling a given tensor. This is the same as function `tile` in Numpy, but no broadcast. For example $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$, $B = [1, 2]$, $\text{tile}(A, B) = \begin{bmatrix} 1 & 2 & 1 & 2 \\ 3 & 4 & 3 & 4 \end{bmatrix}$

Domain	ai.onnx
Since version	13
Earlier versions	1, 6
Inputs	input (differentiable) : T — Input tensor of any shape. repeats (non-differentiable) : T1 — 1D int64 tensor of the same length as input's dimension number, includes numbers of repeated copies along input's dimensions.
Outputs	output (differentiable) : T — Output tensor of the same dimensions and type as tensor input. $\text{output_dim}[i] = \text{input_dim}[i] * \text{repeats}[i]$

Attributes	None.
Type constraints	T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128) — Constrain input and output types to all tensor types. T1 : tensor(int64) — Constrain repeat's type to int64 tensors.
Test vectors	test_tile, test_tile_precomputed
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.197. TopK

Retrieve the top-K largest or smallest elements along a specified axis. Given an input tensor of shape $[a_0, a_1, \dots, a_{n-1}]$ and integer argument k , return two outputs:

- Value tensor of shape $[a_0, a_1, \dots, a_{\text{axis}-1}, k, a_{\text{axis}+1}, \dots, a_{n-1}]$ which contains the values of the top k elements along the specified axis
- Index tensor of shape $[a_0, a_1, \dots, a_{\text{axis}-1}, k, a_{\text{axis}+1}, \dots, a_{n-1}]$ which contains the indices of the top k elements (original indices from the input tensor).
- If “largest” is 1 (the default value) then the k largest elements are returned.
- If “sorted” is 1 (the default value) then the resulting k elements will be sorted.
- If “sorted” is 0, order of returned ‘Values’ and ‘Indices’ are undefined.

Given two equivalent values, this operator uses the indices along the axis as a tiebreaker. That is, the element with the lower index will appear first.

Domain	ai.onnx
Since version	24
Earlier versions	1, 10, 11
Inputs	X (differentiable) : T — Tensor of shape $[a_0, a_1, \dots, a_{n-1}]$ K (non-differentiable) : tensor(int64) — A 1-D tensor containing a single positive value corresponding to the number of top elements to retrieve
Outputs	Values (differentiable) : T — Tensor of shape $[a_0, a_1, \dots, a_{\text{axis}-1}, k, a_{\text{axis}+1}, \dots, a_{n-1}]$ containing top K values from the input tensor Indices (non-differentiable) : I — Tensor of shape $[a_0, a_1, \dots, a_{\text{axis}-1}, k, a_{\text{axis}+1}, \dots, a_{n-1}]$ containing the corresponding input tensor indices for the top K values.
Attributes	axis : int (default is -1) — Dimension on which to do the sort. Negative value means counting dimensions from the back. Accepted range is $[-r, r-1]$ where $r = \text{rank}(\text{input})$. largest : int (default is 1) — Whether to return the top-K largest or smallest elements. sorted : int (default is 1) — Whether to return the elements in sorted order.

Type constraints	T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(float16), tensor(float), tensor(double), tensor(bfloat16) — Constrain input and output types to numeric tensors. I : tensor(int64) — Constrain index tensor to int64
Test vectors	test_top_k, test_top_k_negative_axis, test_top_k_same_values, test_top_k_same_values_2d, test_top_k_same_values_largest, test_top_k_smallest, test_top_k_uint64
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.198. Transpose

Returns a transpose of the input tensor. (Similar to `numpy.transpose`). The optional attribute `perm` specifies the permutation of the axes of the input tensor. `perm` must contain each axis index in `[0, n-1]` exactly once, so its length is equal to the rank `n` of the input tensor.

Axis `i` of the output tensor corresponds to axis `perm[i]` of the input tensor.

If the attribute is omitted, its default value is `(n-1, ..., 0)`, where `n` is the rank of the input tensor (that is, the dimensions are reversed).

For example, when `perm=(1, 0, 2)`, given an input tensor of shape `(1, 2, 3)`, the output shape will be `(2, 1, 3)`. When `perm=(1, 2, 0)`, given an input tensor of shape `(1, 2, 3)`, the output shape will be `(2, 3, 1)`. A 0-D or 1-D input is valid; in those cases the output has the same shape as the input.

Domain	ai.onnx
Since version	25
Earlier versions	1, 13, 21, 23, 24
Inputs	data (differentiable) : T — An input tensor.
Outputs	transposed (differentiable) : T — Transposed output.
Attributes	perm : list of ints — A list of integers. By default, reverse the dimensions; otherwise permute the axes according to the values given. Its length must be equal to the rank of the input, and each value must be in the range <code>[0, rank-1]</code> .
Type constraints	T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128), tensor(float8e4m3fn), tensor(float8e4m3fnuz), tensor(float8e5m2), tensor(float8e5m2fnuz), tensor(uint4), tensor(int4), tensor(float4e2m1), tensor(float8e8m0), tensor(uint2), tensor(int2) — Constrain input and output types to all tensor types.
Test vectors	test_transpose_default

Not supplied by the upstream source Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.199. Trilu

Given a 2-D matrix or batches of 2-D matrices, returns the upper or lower triangular part of the tensor(s). The attribute “upper” determines whether the upper or lower part is retained. If set to true, the upper triangular matrix is retained. Lower triangular matrix is retained otherwise. Default value for the “upper” attribute is true. Trilu takes one input tensor of shape $[*, N, M]$, where $*$ is zero or more batch dimensions. The upper triangular part consists of the elements on and above the given diagonal (k). The lower triangular part consists of elements on and below the diagonal. All other elements in the matrix are set to zero. If $k = 0$, the triangular part on and above/below the main diagonal is retained. If upper is set to true, a positive k retains the upper triangular matrix excluding the main diagonal and $(k-1)$ diagonals above it. A negative k value retains the main diagonal and $|k|$ diagonals below it. If upper is set to false, a positive k retains the lower triangular matrix including the main diagonal and k diagonals above it. A negative k value excludes the main diagonal and $(|k|-1)$ diagonals below it.

Domain	ai.onnx
Since version	14
Inputs (1 — 2)	input (differentiable) : T — Input tensor of rank 2 or higher. k (optional, non-differentiable) : tensor(int64) — A 0-D tensor containing a single value corresponding to the number diagonals above or below the main diagonal to exclude or include. Default value is 0 if it's not specified.
Outputs	output (differentiable) : T — Output tensor of the same type and shape as the input tensor.
Attributes	upper : int (default is 1) — Boolean. Indicates whether upper or lower part of matrix is retained. Default is true.
Type constraints	T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128) — Constrain input and output types to all tensor types.
Test vectors	test_tril, test_tril_neg, test_tril_one_row_neg, test_tril_out_neg, test_tril_out_pos, test_tril_pos, test_tril_square, test_tril_square_neg, test_tril_zero, test_triu, test_triu_neg, test_triu_one_row, test_triu_out_neg_out, test_triu_out_pos, test_triu_pos, test_triu_square, test_triu_square_neg, test_triu_zero
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.200. Unique

Find the unique elements of a tensor. When an optional attribute 'axis' is provided, unique subensors sliced along the 'axis' are returned. Otherwise the input tensor is flattened and unique values of the flattened tensor are returned.

This operator returns the unique values or sliced unique subensors of the input tensor and three optional outputs. The first output tensor 'Y' contains all unique values or subensors of the input. The second optional output tensor 'indices' contains indices of 'Y' elements' first occurrence in 'X'. The third optional output tensor 'inverse_indices' contains, for elements of 'X', its corresponding indices in 'Y'. The fourth optional output tensor 'counts' contains the count of each element of 'Y' in the input.

Outputs are either sorted in ascending order or optionally in the order of the first occurrence of the values in the input.

<https://docs.scipy.org/doc/numpy/reference/generated/numpy.unique.html>

Example 1:

```
input_X = [2, 1, 1, 3, 4, 3]
attribute_sorted = 0
attribute_axis = None
output_Y = [2, 1, 3, 4]
output_indices = [0, 1, 3, 4]
output_inverse_indices = [0, 1, 1, 2, 3, 2]
output_counts = [1, 2, 2, 1]
```

Example 2:

```
input_X = [[1, 3], [2, 3]]
attribute_sorted = 1
attribute_axis = None
output_Y = [1, 2, 3]
output_indices = [0, 2, 1]
output_inverse_indices = [0, 2, 1, 2]
output_counts = [1, 1, 2]
```

Example 3:

```
input_X = [[1, 0, 0], [1, 0, 0], [2, 3, 4]]
attribute_sorted = 1
attribute_axis = 0
output_Y = [[1, 0, 0], [2, 3, 4]]
output_indices = [0, 2]
output_inverse_indices = [0, 0, 1]
output_counts = [2, 1]
```

Example 4:

```
input_x = [[[1., 1.], [0., 1.], [2., 1.], [0., 1.]],
            [[1., 1.], [0., 1.], [2., 1.], [0., 1.]]]
attribute_sorted = 1
attribute_axis = 1
```

intermediate data are presented below for better understanding: there are 4 subensors sliced along axis 1 of input_x (shape = (2, 4, 2)):

```
A: [[1, 1], [1, 1]],
    [[0, 1], [0, 1]],
    [[2, 1], [2, 1]],
```

```
[[0, 1], [0, 1]].
```

there are 3 unique subtensors:

```
[[1, 1], [1, 1]],
[[0, 1], [0, 1]],
[[2, 1], [2, 1]].
```

sorted unique subtensors:

```
B: [[0, 1], [0, 1]],
    [[1, 1], [1, 1]],
    [[2, 1], [2, 1]].
```

output_Y is constructed from B:

```
[[[0. 1.], [1. 1.], [2. 1.]],
 [[0. 1.], [1. 1.], [2. 1.]]]
```

output_indices is to map from B to A:

```
[1, 0, 2]
```

output_inverse_indices is to map from A to B:

```
[1, 0, 2, 0]
```

output_counts:

```
[2, 1, 1]
```

Domain ai.onnx

Since version 28

Earlier versions 11

Inputs X (non-differentiable) : T — A N-D input tensor that is to be processed.

Outputs (1 — 4) Y (non-differentiable) : T — A tensor of the same type as 'X' containing all the unique values or subtensors sliced along a provided 'axis' in 'X', either sorted or maintained in the same order they occur in input 'X'
 indices (optional, non-differentiable) : tensor(int64) — A 1-D INT64 tensor containing indices of 'Y' elements' first occurrence in 'X'. When 'axis' is provided, it contains indices to subtensors in input 'X' on the 'axis'. When 'axis' is not provided, it contains indices to values in the flattened input tensor.
 inverse_indices (optional, non-differentiable) : tensor(int64) — A 1-D INT64 tensor containing, for elements of 'X', its corresponding indices in 'Y'. When 'axis' is provided, it contains indices to subtensors in output 'Y' on the 'axis'. When 'axis' is not provided, it contains indices to values in output 'Y'.
 counts (optional, non-differentiable) : tensor(int64) — A 1-D INT64 tensor containing the count of each element of 'Y' in input 'X'

Attributes axis : int — (Optional) The dimension to apply unique. If not specified, the unique elements of the flattened input are returned. Negative value means counting dimensions from the back. Accepted range is [-r, r-1] where r = rank(input).

	sorted : int (default is 1) — (Optional) Whether to sort the unique elements in ascending order before returning as output. Must be one of 0, or 1 (default).
Type constraints	T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128) — Input can be of any tensor type.
Test vectors	test_unique_length_1, test_unique_not_sorted_without_axis, test_unique_sorted_with_axis, test_unique_sorted_with_axis_3d, test_unique_sorted_with_negative_axis, test_unique_sorted_without_axis, test_unique_bfloat16_sorted_without_axis
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.201. Unsqueeze

Insert single-dimensional entries to the shape of an input tensor (data). Takes one required input axes — which contains a list of dimension indices and this operator will insert a dimension of value 1 into the corresponding index of the output tensor (expanded).

For example, given an input tensor (data) of shape [3, 4, 5], then Unsqueeze(data, axes=[0, 4]) outputs a tensor (expanded) containing same data as data but with shape [1, 3, 4, 5, 1].

The input axes should not contain any duplicate entries. It is an error if it contains duplicates. The rank of the output tensor (output_rank) is the rank of the input tensor (data) plus the number of values in axes. Each value in axes should be within the (inclusive) range [-output_rank, output_rank - 1]. The order of values in axes does not matter and can come in any order.

Domain	ai.onnx
Since version	25
Earlier versions	1, 11, 13, 21, 23, 24
Inputs	data (differentiable) : T — Original tensor axes (non-differentiable) : tensor(int64) — 1D tensor of integers indicating the dimensions to be inserted. Negative value means counting dimensions from the back. Accepted range is [-r, r-1] where r = rank(expanded).
Outputs	expanded (differentiable) : T — Reshaped tensor with same data as input.
Attributes	None.
Type constraints	T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128), tensor(float8e4m3fn), tensor(float8e4m3fnuz), tensor(float8e5m2), tensor(float8e5m2fnuz), tensor(uint4),

tensor(int4), tensor(float4e2m1), tensor(float8e8m0),
 tensor(uint2), tensor(int2) — Constrain input and output types to all
 tensor types up to IRv13.

Test vectors test_unsqueeze_negative_axes, test_unsqueeze_axis_, test_
 unsqueeze_three_axes, test_unsqueeze_two_axes, test_unsqueeze_
 unsorted_axes

Not supplied by the upstream source Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.202. Upsample

Upsample the input tensor. Each dimension value of the output tensor is: $\text{output_dimension} = \text{floor}(\text{input_dimension} * \text{scale})$.

Domain	ai.onnx
Since version	10
Status	Deprecated. Clause 14.5 of ONNX 1-1 applies: a deprecated operator is not removed, and a consumer continues to evaluate it.
Earlier versions	7, 9
Inputs	Not stated by the source.
Outputs	Not stated by the source.
Attributes	None.
Type constraints	Not stated by the source.
Test vectors	test_upsample_nearest
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.203. Where

Return elements, either from X or Y, depending on condition. Where behaves like [numpy.where](#) with three parameters.

This operator supports **multidirectional (i.e., Numpy-style) broadcasting**; for more details please check the doc.

Domain	ai.onnx
Since version	16
Earlier versions	9
Inputs	condition (non-differentiable) : B — When True (nonzero), yield X, otherwise yield Y

	X (differentiable) : T — values selected at indices where condition is True Y (differentiable) : T — values selected at indices where condition is False
Outputs	output (differentiable) : T — Tensor of shape equal to the broadcasted shape of condition, X, and Y.
Attributes	None.
Type constraints	B : tensor(bool) — Constrain to boolean tensors. T : tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(bfloat16), tensor(float16), tensor(float), tensor(double), tensor(string), tensor(bool), tensor(complex64), tensor(complex128) — Constrain input and output types to all tensor types (including bfloat).
Test vectors	test_where_long_example, test_where_example
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

5.204. Xor

Returns the tensor resulted from performing the xor logical operation elementwise on the input tensors A and B (with Numpy-style broadcasting support).

This operator supports **multidirectional (i.e., Numpy-style) broadcasting**; for more details please check the doc.

Domain	ai.onnx
Since version	7
Earlier versions	1
Inputs	A (non-differentiable) : T — First input operand for the logical operator. B (non-differentiable) : T — Second input operand for the logical operator.
Outputs	C (non-differentiable) : T1 — Result tensor.
Attributes	None.
Type constraints	T : tensor(bool) — Constrain input to boolean tensor. T1 : tensor(bool) — Constrain output to boolean tensor.
Test vectors	test_xor2d, test_xor3d, test_xor4d, test_xor_bcast3v1d, test_xor_bcast3v2d, test_xor_bcast4v2d, test_xor_bcast4v3d, test_xor_bcast4v4d
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

6. The ai.onnx.ml domain

6.1. General

This clause specifies the operators of the `ai.onnx.ml` domain, in the form required by [Clause 4](#).

Editorial note —

This clause is generated from the upstream operator documentation, which does not supply three of the elements [Clause 4](#) makes mandatory: shape inference, determinism and error conditions. Every operator below records their absence rather than omitting the elements, so that the incompleteness is visible where it matters instead of only in Annex C of [ONNX 1-1](#).

Supplying them is the substance of the work this part represents. They cannot be generated, because the upstream source does not contain them; they have to be written, per operator, and agreed.

Operator prose below also refers to upstream documents that [ONNX 1-1](#) does not yet restate, broadcasting among them. Those references appear as plain text, because a normative cross-reference to a clause that does not exist would be worse than none. Annex C of [ONNX 1-1](#) records the omission.

NOTE Examples and sample implementations are not reproduced. They are informative under [Clause 4](#), and the reference implementation is not restated here; see Annex D of [ONNX 1-1](#). The test vector names extracted from the examples are retained, because [ONNX 1-3](#) needs them.

6.2. ai.onnx.ml.ArrayFeatureExtractor

Select elements of the input tensor based on the indices passed.

The indices are applied to the last axes of the tensor.

Domain	<code>ai.onnx.ml</code>
Since version	1
Inputs	X : T — Data to be selected Y : <code>tensor(int64)</code> — The indices, based on 0 as the first index of any dimension.
Outputs	Z : T — Selected output data as an array
Attributes	None.
Type constraints	T : <code>tensor(float)</code> , <code>tensor(double)</code> , <code>tensor(int64)</code> , <code>tensor(int32)</code> , <code>tensor(string)</code> — The input must be a tensor of a numeric type or string. The output will be of the same tensor type.
Test vectors	<code>test_ai_onnx_ml_array_feature_extractor</code>
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

6.3. ai.onnx.ml.Binarizer

Maps the values of the input tensor to either 0 or 1, element-wise, based on the outcome of a comparison against a threshold value.

ONNX 1-2 (unofficial draft)

Domain	ai.onnx.ml
Since version	1
Inputs	X : T — Data to be binarized
Outputs	Y : T — Binarized output data
Attributes	threshold : float (default is 0.0) — Values greater than this are mapped to 1, others to 0.
Type constraints	T : tensor(float), tensor(double), tensor(int64), tensor(int32) — The input must be a tensor of a numeric type. The output will be of the same tensor type.
Test vectors	test_ai_onnx_ml_binarizer
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

6.4. ai.onnx.ml.CastMap

Converts a map to a tensor. The map key must be an int64 and the values will be ordered in ascending order based on this key. The operator supports dense packing or sparse packing. If using sparse packing, the key cannot exceed the max_map-1 value.

Domain	ai.onnx.ml
Since version	1
Inputs	X : T1 — The input map that is to be cast to a tensor
Outputs	Y : T2 — A tensor representing the same data as the input map, ordered by their keys
Attributes	cast_to : string (default is TO_FLOAT) — A string indicating the desired element type of the output tensor, one of 'TO_FLOAT', 'TO_STRING', 'TO_INT64'. map_form : string (default is DENSE) — Indicates whether to only output as many values as are in the input (dense), or position the input based on using the key of the map as the index of the output (sparse). One of 'DENSE', 'SPARSE'. max_map : int (default is 1) — If the value of map_form is 'SPARSE,' this attribute indicates the total length of the output tensor.
Type constraints	T1 : map(int64, string), map(int64, float) — The input must be an integer map to either string or float. T2 : tensor(string), tensor(float), tensor(int64) — The output is a 1-D tensor of string, float, or integer.
Test vectors	None published for this operator.
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

6.5. ai.onnx.ml.CategoryMapper

Converts strings to integers and vice versa.

Two sequences of equal length are used to map between integers and strings, with strings and integers at the same index detailing the mapping.

Each operator converts either integers to strings or strings to integers, depending on which default value attribute is provided. Only one default value attribute should be defined.

If the string default value is set, it will convert integers to strings.
If the int default value is set, it will convert strings to integers.

Domain	ai.onnx.ml
Since version	1
Inputs	X : T1 — Input data
Outputs	Y : T2 — Output data. If strings are input, the output values are integers, and vice versa.
Attributes	cats_int64s : list of ints — The integers of the map. This sequence must be the same length as the 'cats_strings' sequence. cats_strings : list of strings — The strings of the map. This sequence must be the same length as the 'cats_int64s' sequence default_int64 : int (default is -1) — An integer to use when an input string value is not found in the map. One and only one of the 'default_*' attributes must be defined. default_string : string (default is _Unused) — A string to use when an input integer value is not found in the map. One and only one of the 'default_*' attributes must be defined.
Type constraints	T1 : tensor(string), tensor(int64) — The input must be a tensor of strings or integers, either [N,C] or [C]. T2 : tensor(string), tensor(int64) — The output is a tensor of strings or integers. Its shape will be the same as the input shape.
Test vectors	None published for this operator.
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

6.6. ai.onnx.ml.DictVectorizer

Uses an index mapping to convert a dictionary to an array.

Given a dictionary, each key is looked up in the vocabulary attribute corresponding to the key type. The index into the vocabulary array at which the key is found is then used to index the output 1-D tensor 'Y' and insert into it the value found in the dictionary 'X'.

The key type of the input map must correspond to the element type of the defined vocabulary attribute.

Therefore, the output array will be equal in length to the index mapping vector parameter.

All keys in the input dictionary must be present in the index mapping vector.

For each item in the input dictionary, insert its value in the output array.

Any keys not present in the input dictionary, will be zero in the output array.

For example: if the `string_vocabulary` parameter is set to `["a", "c", "b", "z"]`,

then an input of `{"a": 4, "c": 8}` will produce an output of `[4, 8, 0, 0]`.

Domain	ai.onnx.ml
Since version	1
Inputs	X : T1 — A dictionary.
Outputs	Y : T2 — A 1-D tensor holding values from the input dictionary.
Attributes	<code>int64_vocabulary</code> : list of ints — An integer vocabulary array. One and only one of the vocabularies must be defined. <code>string_vocabulary</code> : list of strings — A string vocabulary array. One and only one of the vocabularies must be defined.
Type constraints	T1 : <code>map(string, int64)</code>, <code>map(int64, string)</code>, <code>map(int64, float)</code>, <code>map(int64, double)</code>, <code>map(string, float)</code>, <code>map(string, double)</code> — The input must be a map from strings or integers to either strings or a numeric type. The key and value types cannot be the same. T2 : <code>tensor(int64)</code>, <code>tensor(float)</code>, <code>tensor(double)</code>, <code>tensor(string)</code> — The output will be a tensor of the value type of the input map. It's shape will be <code>[1,C]</code>, where C is the length of the input dictionary.
Test vectors	None published for this operator.
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

6.7. ai.onnx.ml.FeatureVectorizer

Concatenates input tensors into one continuous output.

All input shapes are 2-D and are concatenated along the second dimension. 1-D tensors are treated as `[1,C]`.

Inputs are copied to the output maintaining the order of the input arguments.

All inputs must be integers or floats, while the output will be all floating point values.

Domain	ai.onnx.ml
Since version	1
Inputs (1 — unbounded)	X (variadic) : T1 — An ordered collection of tensors, all with the same element type.
Outputs	Y : <code>tensor(float)</code> — The output array, elements ordered as the inputs.

Attributes	<code>inputdimensions</code> : list of ints — The size of each input in the input list
Type constraints	<code>T1</code> : <code>tensor(int32)</code> , <code>tensor(int64)</code> , <code>tensor(float)</code> , <code>tensor(double)</code> — The input type must be a tensor of a numeric type.
Test vectors	None published for this operator.
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

6.8. `ai.onnx.ml.Imputer`

Replaces inputs that equal one value with another, leaving all other elements alone.

This operator is typically used to replace missing values in situations where they have a canonical representation, such as `-1`, `0`, `NaN`, or some extreme value.

One and only one of `imputed_value_floats` or `imputed_value_int64s` should be defined -- floats if the input tensor holds floats, integers if the input tensor holds integers. The imputed values must all fit within the width of the tensor element type. One and only one of the `replaced_value_float` or `replaced_value_int64` should be defined, which one depends on whether floats or integers are being processed.

The `imputed_value` attribute length can be 1 element, or it can have one element per input feature. In other words, if the input tensor has the shape `[*,F]`, then the length of the attribute array may be 1 or `F`. If it is 1, then it is broadcast along the last dimension and applied to each feature.

Domain	<code>ai.onnx.ml</code>
Since version	1
Inputs	<code>X</code> : <code>T</code> — Data to be processed.
Outputs	<code>Y</code> : <code>T</code> — Imputed output data
Attributes	<code>imputed_value_floats</code> : list of floats — Value(s) to change to <code>imputed_value_int64s</code> : list of ints — Value(s) to change to. <code>replaced_value_float</code> : float (default is <code>0.0</code>) — A value that needs replacing. <code>replaced_value_int64</code> : int (default is <code>0</code>) — A value that needs replacing.
Type constraints	<code>T</code> : <code>tensor(float)</code> , <code>tensor(double)</code> , <code>tensor(int64)</code> , <code>tensor(int32)</code> — The input type must be a tensor of a numeric type, either <code>[N,C]</code> or <code>[C]</code> . The output type will be of the same tensor type and shape.
Test vectors	None published for this operator.
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

6.9. ai.onnx.ml.LabelEncoder

Maps each element in the input tensor to another value.

The mapping is determined by the two parallel attributes, 'keys_*' and 'values_*' attribute. The *i*-th value in the specified 'keys_*' attribute would be mapped to the *i*-th value in the specified 'values_*' attribute. It implies that input's element type and the element type of the specified 'keys_*' should be identical while the output type is identical to the specified 'values_*' attribute. Note that the 'keys_*' and 'values_*' attributes must have the same length. If an input element can not be found in the specified 'keys_*' attribute, the 'default_*' that matches the specified 'values_*' attribute may be used as its output value. The type of the 'default_*' attribute must match the 'values_*' attribute chosen.

Let's consider an example which maps a string tensor to an integer tensor. Assume and 'keys_strings' is ["Amy", "Sally"], 'values_int64s' is [5, 6], and 'default_int64' is '-1'. The input ["Dori", "Amy", "Amy", "Sally", "Sally"] would be mapped to [-1, 5, 5, 6, 6].

Since this operator is an one-to-one mapping, its input and output shapes are the same. Notice that only one of 'keys_*/'values_*' can be set.

Float keys with value 'NaN' match any input 'NaN' value regardless of bit value. If a key is repeated, the last key takes precedence.

Domain	ai.onnx.ml
Since version	4
Earlier versions	1, 2
Inputs	X : T1 — Input data. It must have the same element type as the keys_* attribute set.
Outputs	Y : T2 — Output data. This tensor's element type is based on the values_* attribute set.
Attributes	default_float : float (default is -0.0) — A float. default_int64 : int (default is -1) — An integer. default_string : string (default is _Unused) — A string. default_tensor : tensor — A default tensor. {"Unused"} if values_* has string type, {-1} if values_* has integral type, and {-0.f} if values_* has float type. keys_floats : list of floats — A list of floats. keys_int64s : list of ints — A list of ints. keys_strings : list of strings — A list of strings. keys_tensor : tensor — Keys encoded as a 1D tensor. One and only one of 'keys_*'s should be set. values_floats : list of floats — A list of floats. values_int64s : list of ints — A list of ints. values_strings : list of strings — A list of strings. values_tensor : tensor — Values encoded as a 1D tensor. One and only one of 'values_*'s should be set.
Type constraints	T1 : tensor(string), tensor(int64), tensor(float), tensor(int32), tensor(int16), tensor(double) — The input type is a tensor of any shape.

T2 : tensor(string), tensor(int64), tensor(float), tensor(int32), tensor(int16), tensor(double) — Output type is determined by the specified 'values_*' attribute.

Test vectors test_ai_onnx_ml_label_encoder_string_int, test_ai_onnx_ml_label_encoder_string_int_no_default, test_ai_onnx_ml_label_encoder_tensor_mapping, test_ai_onnx_ml_label_encoder_tensor_value_only_mapping

Not supplied by the upstream source Shape inference, determinism, errors. See the editorial note at the head of this clause.

6.10. ai.onnx.ml.LinearClassifier

Linear classifier

Domain	ai.onnx.ml
Since version	1
Inputs	X : T1 — Data to be classified.
Outputs	Y : T2 — Classification outputs (one class per example). Z : tensor(float) — Classification scores ([N,E] — one score for each class and example
Attributes	classlabels_ints : list of ints — Class labels when using integer labels. One and only one 'classlabels' attribute must be defined. classlabels_strings : list of strings — Class labels when using string labels. One and only one 'classlabels' attribute must be defined. coefficients : list of floats (required) — A collection of weights of the model(s). intercepts : list of floats — A collection of intercepts. multi_class : int (default is 0) — Indicates whether to do OvR or multinomial (0=OvR is the default). post_transform : string (default is NONE) — Indicates the transform to apply to the scores vector. One of 'NONE,' 'SOFTMAX,' 'LOGISTIC,' 'SOFTMAX_ZERO,' or 'PROBIT'
Type constraints	T1 : tensor(float), tensor(double), tensor(int64), tensor(int32) — The input must be a tensor of a numeric type, and of shape [N,C] or [C]. In the latter case, it will be treated as [1,C] T2 : tensor(string), tensor(int64) — The output will be a tensor of strings or integers.
Test vectors	None published for this operator.
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

6.11. ai.onnx.ml.LinearRegressor

Generalized linear regression evaluation.

If targets is set to 1 (default) then univariate regression is performed.

If targets is set to M then M sets of coefficients must be passed in as a sequence and M results will be output for each input n in N.

The coefficients array is of length n, and the coefficients for each target are contiguous.

Intercepts are optional but if provided must match the number of targets.

Domain	ai.onnx.ml
Since version	1
Inputs	X : T — Data to be regressed.
Outputs	Y : tensor(float) — Regression outputs (one per target, per example).
Attributes	coefficients : list of floats — Weights of the model(s). intercepts : list of floats — Weights of the intercepts, if used. post_transform : string (default is NONE) — Indicates the transform to apply to the regression output vector. One of 'NONE,' 'SOFTMAX,' 'LOGISTIC,' 'SOFTMAX_ZERO,' or 'PROBIT' targets : int (default is 1) — The total number of regression targets, 1 if not defined.
Type constraints	T : tensor(float), tensor(double), tensor(int64), tensor(int32) — The input must be a tensor of a numeric type.
Test vectors	None published for this operator.
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

6.12. ai.onnx.ml.Normalizer

Normalize the input. There are three normalization modes, which have the corresponding formulas, defined using element-wise infix operators '/' and '^' and tensor-wide functions 'max' and 'sum':

Max: $Y = X / \max(X)$

L1: $Y = X / \text{sum}(X)$

L2: $Y = \sqrt{X^2 / \text{sum}(X^2)}$

In all modes, if the divisor is zero, $Y == X$.

For batches, that is, [N,C] tensors, normalization is done along the C axis. In other words, each row of the batch is normalized independently.

Domain	ai.onnx.ml
Since version	1

Inputs	<code>X</code> : <code>T</code> — Data to be encoded, a tensor of shape <code>[N,C]</code> or <code>[C]</code>
Outputs	<code>Y</code> : <code>tensor(float)</code> — Encoded output data
Attributes	<code>norm</code> : <code>string</code> (default is <code>MAX</code>) — One of <code>'MAX','L1','L2'</code>
Type constraints	<code>T</code> : <code>tensor(float)</code> , <code>tensor(double)</code> , <code>tensor(int64)</code> , <code>tensor(int32)</code> — The input must be a tensor of a numeric type.
Test vectors	None published for this operator.
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

6.13. `ai.onnx.ml.OneHotEncoder`

Replace each input element with an array of ones and zeros, where a single one is placed at the index of the category that was passed in. The total category count will determine the size of the extra dimension of the output array `Y`.

For example, if we pass a tensor with a single value of 4, and a category count of 8, the output will be a tensor with ```[0,0,0,0,1,0,0,0]```.

This operator assumes every input feature is from the same set of categories.

If the input is a tensor of `float`, `int32`, or `double`, the data will be cast to integers and the `cats_int64s` category list will be used for the lookups.

Domain	<code>ai.onnx.ml</code>
Since version	1
Inputs	<code>X</code> : <code>T</code> — Data to be encoded.
Outputs	<code>Y</code> : <code>tensor(float)</code> — Encoded output data, having one more dimension than <code>X</code> .
Attributes	<code>cats_int64s</code> : <code>list of ints</code> — List of categories, ints. One and only one of the <code>'cats_*</code> attributes must be defined. <code>cats_strings</code> : <code>list of strings</code> — List of categories, strings. One and only one of the <code>'cats_*</code> attributes must be defined. <code>zeros</code> : <code>int</code> (default is 1) — If true and category is not present, will return all zeros; if false and a category if not found, the operator will fail.
Type constraints	<code>T</code> : <code>tensor(string)</code> , <code>tensor(int64)</code> , <code>tensor(int32)</code> , <code>tensor(float)</code> , <code>tensor(double)</code> — The input must be a tensor of a numeric type.
Test vectors	None published for this operator.
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

6.14. ai.onnx.ml.SVMClassifier

Support Vector Machine classifier

Domain	ai.onnx.ml
Since version	1
Inputs	X : T1 — Data to be classified.
Outputs	Y : T2 — Classification outputs (one class per example). Z : tensor(float) — Class scores (one per class per example), if prob_a and prob_b are provided they are probabilities for each class, otherwise they are raw scores.
Attributes	classlabels_ints : list of ints — Class labels if using integer labels. One and only one of the 'classlabels_*' attributes must be defined. classlabels_strings : list of strings — Class labels if using string labels. One and only one of the 'classlabels_*' attributes must be defined. coefficients : list of floats kernel_params : list of floats — List of 3 elements containing gamma, coef0, and degree, in that order. Zero if unused for the kernel. kernel_type : string (default is LINEAR) — The kernel type, one of 'LINEAR', 'POLY', 'RBF', 'SIGMOID'. post_transform : string (default is NONE) — Indicates the transform to apply to the score. One of 'NONE', 'SOFTMAX', 'LOGISTIC', 'SOFTMAX_ZERO', or 'PROBIT' prob_a : list of floats — First set of probability coefficients. prob_b : list of floats — Second set of probability coefficients. This array must be same size as prob_a. If these are provided then output Z are probability estimates, otherwise they are raw scores. rho : list of floats support_vectors : list of floats vectors_per_class : list of ints
Type constraints	T1 : tensor(float), tensor(double), tensor(int64), tensor(int32) — The input must be a tensor of a numeric type, either [C] or [N,C]. T2 : tensor(string), tensor(int64) — The output type will be a tensor of strings or integers, depending on which of the classlabels_* attributes is used. Its size will match the batch size of the input.
Test vectors	None published for this operator.
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

6.15. ai.onnx.ml.SVMRegressor

Support Vector Machine regression prediction and one-class SVM anomaly detection.

Domain	ai.onnx.ml
--------	------------

Since version	1
Inputs	X : T — Data to be regressed.
Outputs	Y : tensor(float) — Regression outputs (one score per target per example).
Attributes	coefficients : list of floats — Support vector coefficients. kernel_params : list of floats — List of 3 elements containing gamma, coef0, and degree, in that order. Zero if unused for the kernel. kernel_type : string (default is LINEAR) — The kernel type, one of 'LINEAR,' 'POLY,' 'RBF,' 'SIGMOID'. n_supports : int (default is 0) — The number of support vectors. one_class : int (default is 0) — Flag indicating whether the regression is a one-class SVM or not. post_transform : string (default is NONE) — Indicates the transform to apply to the score. One of 'NONE,' 'SOFTMAX,' 'LOGISTIC,' 'SOFTMAX_ZERO,' or 'PROBIT.' rho : list of floats support_vectors : list of floats — Chosen support vectors
Type constraints	T : tensor(float), tensor(double), tensor(int64), tensor(int32) — The input type must be a tensor of a numeric type, either [C] or [N,C].
Test vectors	None published for this operator.
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

6.16. ai.onnx.ml.Scaler

Rescale input data, for example to standardize features by removing the mean and scaling to unit variance.

Domain	ai.onnx.ml
Since version	1
Inputs	X : T — Data to be scaled.
Outputs	Y : tensor(float) — Scaled output data.
Attributes	offset : list of floats — First, offset by this. Can be length of features in an [N,F] tensor or length 1, in which case it applies to all features, regardless of dimension count. scale : list of floats — Second, multiply by this. Can be length of features in an [N,F] tensor or length 1, in which case it applies to all features, regardless of dimension count. Must be same length as 'offset'
Type constraints	T : tensor(float), tensor(double), tensor(int64), tensor(int32) — The input must be a tensor of a numeric type.
Test vectors	None published for this operator.

Not supplied by the upstream source Shape inference, determinism, errors. See the editorial note at the head of this clause.

6.17. ai.onnx.ml.TreeEnsemble

Tree Ensemble operator. Returns the regressed values for each input in a batch. Inputs have dimensions $[N, F]$ where N is the input batch size and F is the number of input features. Outputs have dimensions $[N, \text{num_targets}]$ where N is the batch size and num_targets is the number of targets, which is a configurable attribute.

The encoding of this attribute is split along interior nodes and the leaves of the trees. Notably, attributes with the prefix ``nodes_*` are associated with interior nodes, and attributes with the prefix ``leaf_*` are associated with leaves.

The attributes ``nodes_*` must all have the same length and encode a sequence of tuples, as defined by taking all the ``nodes_*` fields at a given position.

All fields prefixed with ``leaf_*` represent tree leaves, and similarly define tuples of leaves and must have identical length.

This operator can be used to implement both the previous ``TreeEnsembleRegressor`` and ``TreeEnsembleClassifier`` nodes.

The ``TreeEnsembleRegressor`` node maps directly to this node and requires changing how the nodes are represented.

The ``TreeEnsembleClassifier`` node can be implemented by adding a ``ArgMax`` node after this node to determine the top class.

To encode class labels, a ``LabelEncoder`` or ``GatherND`` operator may be used.

Domain `ai.onnx.ml`

Since version 5

Inputs $X : T$ — Input of shape [Batch Size, Number of Features]

Outputs $Y : T$ — Output of shape [Batch Size, Number of targets]

Attributes `aggregate_function` : int (default is 1) — Defines how to aggregate leaf values within a target. One of 'AVERAGE' (0) 'SUM' (1) 'MIN' (2) 'MAX' (3) defaults to 'SUM' (1)
`leaf_targetids` : list of ints (required) — The index of the target that this leaf contributes to (this must be in range $[0, \text{num_targets})$).
`leaf_weights` : tensor (required) — The weight for each leaf.
`membership_values` : tensor — Members to test membership of for each set membership node. List all of the members to test again in the order that the 'BRANCH_MEMBER' mode appears in `node_modes`, delimited by NaNs. Will have the same number of sets of values as nodes with mode 'BRANCH_MEMBER'. This may be omitted if the node doesn't contain any 'BRANCH_MEMBER' nodes.
`num_targets` : int — The total number of targets.
`nodes_falseleaves` : list of ints (required) — 1 if false branch is leaf for each node and 0 if an interior node. To represent a tree that is a leaf (only has one node), one can do so by having a single `nodes_*` entry with true and false branches referencing the same `leaf_*` entry
`nodes_falsenodeids` : list of ints (required) — If `nodes_falseleaves` is false at an entry, this represents the position of the false branch node. This position can be used to index into a `nodes_*` entry. If `nodes_falseleaves` is false, it is an index into the `leaf_*` attributes.

`nodes_featureids` : list of ints (required) — Feature id for each node.
`nodes_hitrates` : tensor — Popularity of each node, used for performance and may be omitted.
`nodes_missing_value_tracks_true` : list of ints — For each node, define whether to follow the true branch (if attribute value is 1) or false branch (if attribute value is 0) in the presence of a NaN input feature. This attribute may be left undefined and the default value is false (0) for all nodes.
`nodes_modes` : tensor (required) — The comparison operation performed by the node. This is encoded as an enumeration of 0 ('BRANCH_LEQ'), 1 ('BRANCH_LT'), 2 ('BRANCH_GTE'), 3 ('BRANCH_GT'), 4 ('BRANCH_EQ'), 5 ('BRANCH_NEQ'), and 6 ('BRANCH_MEMBER'). Note this is a tensor of type uint8.
`nodes_splits` : tensor (required) — Thresholds to do the splitting on for each node with mode that is not 'BRANCH_MEMBER'.
`nodes_trueleafs` : list of ints (required) — 1 if true branch is leaf for each node and 0 an interior node. To represent a tree that is a leaf (only has one node), one can do so by having a single `nodes_*` entry with true and false branches referencing the same `leaf_*` entry.
`nodes_truenodeids` : list of ints (required) — If `nodes_trueleafs` is false at an entry, this represents the position of the true branch node. This position can be used to index into a `nodes_*` entry. If `nodes_trueleafs` is false, it is an index into the `leaf_*` attributes.
`post_transform` : int (default is 0) — Indicates the transform to apply to the score. One of 'NONE' (0), 'SOFTMAX' (1), 'LOGISTIC' (2), 'SOFTMAX_ZERO' (3) or 'PROBIT' (4), defaults to 'NONE' (0).
`tree_roots` : list of ints (required) — Index into `nodes_*` for the root of each tree. The tree structure is derived from the branching of each node.

Type constraints	<code>T</code> : tensor(float), tensor(double), tensor(float16) — The input type must be a tensor of a numeric type.
Test vectors	<code>test_ai_onnx_ml_tree_ensemble_set_membership</code> , <code>test_ai_onnx_ml_tree_ensemble_single_tree</code>
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

6.18. ai.onnx.ml.TreeEnsembleClassifier

This operator is DEPRECATED. Please use `TreeEnsemble` which provides similar functionality. In order to determine the top class, the `ArgMax` node can be applied to the output of `TreeEnsemble`. To encode class labels, use a `LabelEncoder` operator. Tree Ensemble classifier. Returns the top class for each of N inputs.

The attributes named '`nodes_X`' form a sequence of tuples, associated by index into the sequences, which must all be of equal length. These tuples define the nodes.

Similarly, all fields prefixed with '`class_`' are tuples of votes at the leaves. A leaf may have multiple votes, where each vote is weighted by the associated `class_weights` index.

One and only one of `classlabels_strings` or `classlabels_int64s` will be defined. The `class_ids` are indices into this list.

ONNX 1-2 (unofficial draft)

All fields ending with `__as_tensor__` can be used instead of the same parameter without the suffix if the element type is double and not float.

Domain	ai.onnx.ml
Since version	5
Status	Deprecated. Clause 14.5 of ONNX 1-1 applies: a deprecated operator is not removed, and a consumer continues to evaluate it.
Earlier versions	1, 3
Inputs	Not stated by the source.
Outputs	Not stated by the source.
Attributes	None.
Type constraints	Not stated by the source.
Test vectors	None published for this operator.
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

6.19. ai.onnx.ml.TreeEnsembleRegressor

This operator is DEPRECATED. Please use TreeEnsemble instead which provides the same functionality.

Tree Ensemble regressor. Returns the regressed values for each input in N.

All args with `nodes_` are fields of a tuple of tree nodes, and it is assumed they are the same length, and an index `i` will decode the tuple across these inputs. Each node id can appear only once for each tree id.

All fields prefixed with `target_` are tuples of votes at the leaves.

A leaf may have multiple votes, where each vote is weighted by the associated `target_weights` index.

All fields ending with `__as_tensor__` can be used instead of the same parameter without the suffix if the element type is double and not float. All trees must have their node ids start at 0 and increment by 1.

Mode enum is `BRANCH_LEQ`, `BRANCH_LT`, `BRANCH_GTE`, `BRANCH_GT`, `BRANCH_EQ`, `BRANCH_NEQ`, `LEAF`

Domain	ai.onnx.ml
Since version	5
Status	Deprecated. Clause 14.5 of ONNX 1-1 applies: a deprecated operator is not removed, and a consumer continues to evaluate it.
Earlier versions	1, 3

Inputs	Not stated by the source.
Outputs	Not stated by the source.
Attributes	None.
Type constraints	Not stated by the source.
Test vectors	None published for this operator.
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

6.20. **ai.onnx.ml.ZipMap**

Creates a map from the input and the attributes.

The values are provided by the input tensor, while the keys are specified by the attributes.

Must provide keys in either `classlabels_strings` or `classlabels_int64s` (but not both).

The columns of the tensor correspond one-by-one to the keys specified by the attributes. There must be as many columns as keys.

Domain	<code>ai.onnx.ml</code>
Since version	1
Inputs	<code>X : tensor(float)</code> — The input values
Outputs	<code>Z : T</code> — The output map
Attributes	<code>classlabels_int64s : list of ints</code> — The keys when using int keys. One and only one of the ' <code>classlabels_*</code> ' attributes must be defined. <code>classlabels_strings : list of strings</code> — The keys when using string keys. One and only one of the ' <code>classlabels_*</code> ' attributes must be defined.
Type constraints	<code>T : seq(map(string, float)), seq(map(int64, float))</code> — The output will be a sequence of string or integer maps to float.
Test vectors	None published for this operator.
Not supplied by the upstream source	Shape inference, determinism, errors. See the editorial note at the head of this clause.

Annex A

(normative)

Operator registry

This annex lists every operator of every domain specified by this document, with the operator set versions in which each definition took effect, as an index into the clauses above.

Editorial note —

The registry is the machine-readable face of this part: a table of domain, operator, and the versions at which a definition exists. It is what a validating consumer checks a node against under Clause 10 of [ONNX 1-1](#), and what a conformance statement enumerates when it lists unsupported operators.

It is to be generated together with the clauses it indexes, not maintained separately; a registry that can drift from the definitions is worse than none.

Annex B (informative) Changelog

This annex records, for each operator set version, the operators added, the operators given a new definition, and the type constraints widened.

Editorial note —

To be generated from the registry in Annex A by comparing consecutive operator set versions, rather than maintained by hand. Upstream keeps an equivalent changelog; the vendored copy at `upstream/onnx/docs/Changelog.md` is the material this annex restates.

Bibliography

- [1] ONNX, *Open Neural Network Exchange*, <https://onnx.ai>