cntk.ops package

CNTK core operators. Calling these operators creates nodes in the CNTK computational graph.

AVG_POOLING *= 1*

int – constant used to specify average pooling

MAX_POOLING *= 0*

int – constant used to specify maximum pooling

MAX_UNPOOLING *= 0*

int – constant used to specify maximum unpooling

abs(*x*, *name=''*) [source]

Computes the element-wise absolute of `x`:

$$abs(x) = |x|$$

Example

```
>>> C.abs([-1, 1, -2, 3]).eval()
array([ 1.,  1.,  2.,  3.], dtype=float32)
```

Parameters:
- **x** – numpy array or any `Function` that outputs a tensor
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

alias(*x*, *name=''*) [source]

Create a new Function instance which just aliases the specified 'x' Function/Variable such that the 'Output' of the new 'Function' is same as the 'Output' of the specified 'x' Function/Variable, and has the newly specified name. The purpose of this operator is to create a new distinct reference to a symbolic computation which is different from the original Function/Variable that it aliases and can be used for e.g. to substitute a specific instance of the aliased Function/Variable in the computation graph instead of substituting all usages of the aliased Function/Variable.

Parameters:
- **operand** – The Function/Variable to alias
- **name** (*str, optional*) – the name of the Alias Function in the network

Returns: `Function`

---

argmax(*x, axis=None, name=''*) [source]

Computes the argmax of the input tensor's elements across the specified axis. If no axis is specified, it will return the flatten index of the largest element in tensor x.

Example

```
>>> # create 3x2 matrix in a sequence of length 1 in a batch of one sample
>>> data = [[10, 20],[30, 40],[50, 60]]
```

```
>>> C.argmax(data, 0).eval()
array([[ 2.,  2.]], dtype=float32)
```

```
>>> C.argmax(data, 1).eval()
array([[ 1.],
       [ 1.],
       [ 1.]], dtype=float32)
```

Parameters:
- **x** (*numpy.array* or `Function`) – any `Function` that outputs a tensor.
- **axis** (int or `Axis`) – axis along which the reduction will be performed
- **name** (*str, default to ''*) – the name of the Function instance in the network

Returns: An instance of `Function`

Return type: cntk.ops.functions.Function

---

argmin(*x, axis=None, name=''*) [source]

Computes the argmin of the input tensor's elements across the specified axis. If no axis is specified, it will return the flatten index of the smallest element in tensor x.

Example

```
>>> # create 3x2 matrix in a sequence of length 1 in a batch of one sample
>>> data = [[10, 30],[40, 20],[60, 50]]
```

```
>>> C.argmin(data, 0).eval()
array([[ 0.,  1.]], dtype=float32)
```

```
>>> C.argmin(data, 1).eval()
array([[ 0.],
       [ 1.],
       [ 1.]], dtype=float32)
```

Parameters:
- **x** (*numpy.array* or `Function`) – any `Function` that outputs a tensor.
- **axis** (int or `Axis`) – axis along which the reduction will be performed
- **name** (*str, default to ''*) – the name of the Function instance in the network

Returns: An instance of `Function`

Return type: cntk.ops.functions.Function

---

as_block(*composite, block_arguments_map, block_op_name, block_instance_name=''*) [source]

Create a new block Function instance which just encapsulates the specified composite Function to create a new Function that appears to be a primitive. All the arguments of the composite being encapsulated must be Placeholder variables. The purpose of block Functions is to enable creation of hierarchical Function graphs where details of implementing certain building block operations can be encapsulated away such that the actual structure of the block's implementation is not inlined into the parent graph where the block is used, and instead the block just appears as an opaque primitive. Users still have the ability to peek at the underlying Function graph that implements the actual block Function.

Parameters:
- **composite** – The composite Function that the block encapsulates
- **block_arguments_map** – A list of tuples, mapping from block's underlying composite's arguments to actual variables they are connected to
- **block_op_name** – Name of the op that the block represents
- **block_instance_name** (*str, optional*) – the name of the block Function in the network

Returns: `Function`

---

as_composite(*root_function, name=''*) [source]

Creates a composite Function that has the specified root_function as its root. The composite denotes a higher-level Function encapsulating the entire graph of Functions underlying the specified rootFunction.

Parameters:
- **root_function** – Root Function, the graph underlying which, the newly created composite encapsulates
- **name** (*str, optional*) – the name of the Alias Function in the network

Returns: `Function`

---

assign(*ref, input, name=''*) [source]

Assign the value in input to ref and return the new value, ref need to be the same layout as input. Both ref and input can't have dynamic axis and broadcast isn't supported for the assign operator. During forward pass, ref will get the new value after the forward or backward pass finish, so that any part of the graph that depend on ref will get the old value. To get the new value, use the one returned by the assign node. The reason for that is to make `assign` have a deterministic behavior.

If not computing gradients, the ref will be assigned the new value after the forward pass over the entire Function graph is complete; i.e. all uses of ref in the forward pass will use the original (pre-assignment) value of ref.

If computing gradients (training mode), the assignment to ref will happen after completing both the forward and backward passes over the entire Function graph.

The ref must be a Parameter or Constant. If the same ref is used in multiple assign operations, then the order in which the assignment happens is non-deterministic and the final value can be either of the assignments unless an order is established using a data dependence between the assignments.

Example

```
>>> dest = C.constant(shape=(3,4))
>>> data = C.parameter(shape=(3,4), init=2)
>>> C.assign(dest,data).eval()
array([[ 2.,  2.,  2.,  2.],
       [ 2.,  2.,  2.,  2.],
       [ 2.,  2.,  2.,  2.]], dtype=float32)
>>> dest.asarray()
array([[ 2.,  2.,  2.,  2.],
       [ 2.,  2.,  2.,  2.],
       [ 2.,  2.,  2.,  2.]], dtype=float32)
```

```
>>> dest = C.parameter(shape=(3,4), init=0)
>>> a = C.assign(dest, data)
>>> y = dest + data
>>> result = C.combine([y, a]).eval()
>>> result[y.output]
array([[ 2.,  2.,  2.,  2.],
       [ 2.,  2.,  2.,  2.],
       [ 2.,  2.,  2.,  2.]], dtype=float32)
>>> dest.asarray()
array([[ 2.,  2.,  2.,  2.],
       [ 2.,  2.,  2.,  2.],
       [ 2.,  2.,  2.,  2.]], dtype=float32)
>>> result = C.combine([y, a]).eval()
>>> result[y.output]
array([[ 4.,  4.,  4.,  4.],
       [ 4.,  4.,  4.,  4.],
       [ 4.,  4.,  4.,  4.]], dtype=float32)
>>> dest.asarray()
array([[ 2.,  2.,  2.,  2.],
       [ 2.,  2.,  2.,  2.],
       [ 2.,  2.,  2.,  2.]], dtype=float32)
```

Parameters:

- **ref** – class: *~cntk.variables.Constant* or *~cntk.variables.Parameter*.
- **input** – class:*~cntk.ops.functions.Function* that outputs a tensor
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

associative_multi_arg(*f*) [source]

The output of this operation is the result of an operation (*plus*, *log_add_exp*, *element_times*, *element_max*, *element_min*) of two or more input tensors. Broadcasting is supported.

Example

```
>>> C.plus([1, 2, 3], [4, 5, 6]).eval()
array([ 5.,  7.,  9.], dtype=float32)
```

```
>>> C.element_times([5., 10., 15., 30.], [2.]).eval()
array([ 10.,  20.,  30.,  60.], dtype=float32)
```

```
>>> C.plus([-5, -4, -3, -2, -1], [10], [3, 2, 3, 2, 3], [-13], [+42],
'multi_arg_example').eval()
array([ 37.,  37.,  39.,  39.,  41.], dtype=float32)
```

```
>>> C.element_times([5., 10., 15., 30.], [2.], [1., 2., 1., 2.]).eval()
array([  10.,   40.,   30.,  120.], dtype=float32)
```

```
>>> a = np.arange(3,dtype=np.float32)
>>> np.exp(C.log_add_exp(np.log(1+a), np.log(1+a*a)).eval())
array([ 2.,  4.,  8.], dtype=float32)
```

Parameters:

- **left** – left side tensor
- **right** – right side tensor
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

---

batch_normalization(*operand, scale, bias, running_mean, running_inv_std, spatial, normalization_time_constant=5000, blend_time_constant=0, epsilon=1e-05, use_cudnn_engine=False, name='', running_count=None*) [source]

Normalizes layer outputs for every minibatch for each output (feature) independently and applies affine transformation to preserve representation of the layer.

Parameters:

- **operand** – input of the batch normalization operation
- **scale** – parameter tensor that holds the learned componentwise-scaling factors
- **bias** – parameter tensor that holds the learned bias. `scale` and `bias` must have the same dimensions which must be equal to the input dimensions in case of `spatial` = False or number of output convolution feature maps in case of `spatial` = True
- **running_mean** – running mean which is used during evaluation phase and might be used during training as well. You must pass a constant tensor with initial value 0 and the same dimensions as `scale` and `bias`
- **running_inv_std** – running variance. Represented as `running_mean`
- **running_count** – Denotes the total number of samples that have been used so far to compute the `running_mean` and `running_inv_std` parameters. You must pass a scalar (either rank-0 `constant(val)`).
- **spatial** (*bool*) – flag that indicates whether to compute mean/var for each feature in a minibatch independently or, in case of convolutional layers, per future map
- **normalization_time_constant** (*float, default 5000*) – time constant for computing running average of mean and variance as a low-pass filtered version of the batch statistics.
- **blend_time_constant** (*float, default 0*) – constant for smoothing batch estimates with the running statistics
- **epsilon** – conditioner constant added to the variance when computing the inverse standard deviation
- **use_cudnn_engine** (*bool, default True*) –
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

ceil(*arg, name=''*) [source]

The output of this operation is the element wise value rounded to the smallest integer greater than or equal to the input.

Example

```
>>> C.ceil([0.2, 1.3, 4., 5.5, 0.0]).eval()
array([ 1.,  2.,  4.,  6.,  0.], dtype=float32)
```

```
>>> C.ceil([[0.6, 3.3], [1.9, 5.6]]).eval()
array([[ 1.,  4.],
       [ 2.,  6.]], dtype=float32)
```

Parameters:

- **arg** – input tensor
- **name** (*str, optional*) – the name of the Function instance in the network (optional)

Returns: `Function`

clip(*x, min_value, max_value, name=''*) [source]

Computes a tensor with all of its values clipped to fall between `min_value` and `max_value`, i.e. `min(max(x, min_value), max_value)`.

The output tensor has the same shape as `x`.

Example

```
>>> C.clip([1., 2.1, 3.0, 4.1], 2., 4.).eval()
array([ 2. ,  2.1,  3. ,  4. ], dtype=float32)
```

```
>>> C.clip([-10., -5., 0., 5., 10.], [-5., -4., 0., 3., 5.], [5., 4., 1., 4., 9.]).eval()
array([-5., -4.,  0.,  4.,  9.], dtype=float32)
```

Parameters:

- **x** – tensor to be clipped
- **min_value** (*float*) – a scalar or a tensor which represents the minimum value to clip element values to
- **max_value** (*float*) – a scalar or a tensor which represents the maximum value to clip element values to
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

combine(**operands, **kw_name*) [source]

Create a new Function instance which just combines the outputs of the specified list of 'operands' Functions such that the 'Outputs' of the new 'Function' are union of the 'Outputs' of each of the specified 'operands' Functions. E.g., when creating a classification model, typically the CrossEntropy loss Function and the ClassificationError Function comprise the two roots of the computation graph which can be combined to create a single Function with 2 outputs; viz. CrossEntropy loss and ClassificationError output.

Example

```
>>> in1 = C.input_variable((4,))
>>> in2 = C.input_variable((4,))
```

```
>>> in1_data = np.asarray([[1., 2., 3., 4.]], np.float32)
>>> in2_data = np.asarray([[0., 5., -3., 2.]], np.float32)
```

```
>>> plus_operation = in1 + in2
>>> minus_operation = in1 - in2
```

```
>>> forward = C.combine([plus_operation, minus_operation]).eval({in1: in1_data, in2: in2_data})
>>> len(forward)
2
>>> list(forward.values())
[array([[[ 1., -3.,  6.,  2.]]], dtype=float32),
 array([[[ 1.,  7.,  0.,  6.]]], dtype=float32)]
>>> x = C.input_variable((4,))
>>> _ = C.combine(x, x)
>>> _ = C.combine([x, x])
>>> _ = C.combine((x, x))
>>> _ = C.combine(C.combine(x, x), x)
```

Parameters:
- **operands** (*list*) – list of functions or their variables to combine
- **name** (*str, optional*) – the name of the Combine Function in the network

Returns: `Function`

constant(*value=None, shape=None, dtype=None, device=None, name=''*) [source]

It creates a constant tensor initialized from a numpy array

Example

```
>>> constant_data = C.constant([[1., 2.], [3., 4.], [5., 6.]])
>>> constant_data.value
array([[ 1.,  2.],
       [ 3.,  4.],
       [ 5.,  6.]], dtype=float32)
```

Parameters:

- **value** (*scalar or NumPy array, optional*) – a scalar initial value that would be replicated for every element in the tensor or NumPy array. If `None`, the tensor will be initialized uniformly random.
- **shape** (*tuple or int, optional*) – the shape of the input tensor. If not provided, it will be inferred from `value`.
- **dtype** (*optional*) – data type of the constant. If a NumPy array and `dtype`, are given, then data will be converted if needed. If none given, it will default to `np.float32`.
- **device** (`DeviceDescriptor`) – instance of DeviceDescriptor
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Constant`

---

convolution(*convolution_map, operand, strides=(1,), sharing=[True], auto_padding=[True], max_temp_mem_size_in_samples=0, name=''*) [source]

Computes the convolution of `convolution_map` (typically a tensor of learnable parameters) with `operand` (commonly an image or output of a previous convolution/pooling operation). This operation is used in image and language processing applications. It supports arbitrary dimensions, strides, sharing, and padding.

This function operates on input tensors with dimensions $[C \times M_1 \times M_2 \times \ldots \times M_n]$. This can be understood as a rank-n object, where each entry consists of a C-dimensional vector. For example, an RGB image would have dimensions $[3 \times W \times H]$, i.e. a $[W \times H]$-sized structure, where each entry (pixel) consists of a 3-tuple.

convolution convolves the input `operand` with a $n+2$ rank tensor of (typically learnable) filters called `convolution_map` of shape $[O \times I \times m_1 \times m_2 \times \ldots \times m_n]$ (typically $m_i \ll M_i$). The first dimension, O, is the nunber of convolution filters (i.e. the number of channels in the output). The second dimension, I, must match the number of channels in the input. The last n dimensions are the spatial extent of the filter. I.e. for each output position, a vector of dimension O is computed. Hence, the total number of filter parameters is $O \times I \times m_1 \times m_2 \times \ldots \times m_n$

Example

```
>>> img = np.reshape(np.arange(25.0, dtype = np.float32), (1, 5, 5))
>>> x = C.input_variable(img.shape)
>>> filter = np.reshape(np.array([2, -1, -1, 2], dtype = np.float32), (1, 2, 2))
>>> kernel = C.constant(value = filter)
>>> np.round(C.convolution(kernel, x, auto_padding = [False]).eval({x: [img]}),5)
array([[[[  6.,   8.,  10.,  12.],
         [ 16.,  18.,  20.,  22.],
         [ 26.,  28.,  30.,  32.],
         [ 36.,  38.,  40.,  42.]]]], dtype=float32)
```

Parameters:

- **convolution_map** – convolution filter weights, stored as a tensor of dimensions $[O \times I \times m_1 \times m_2 \times \ldots \times m_n]$, where $[m_1 \times m_2 \times \ldots \times m_n]$ must be the kernel dimensions (spatial extent of the filter).
- **operand** – convolution input. A tensor with dimensions $[I \times M_1 \times M_2 \times \ldots \times M_n]$.
- **strides** (*tuple, optional*) – stride dimensions. If strides[i] > 1 then only pixel positions that are multiples of strides[i] are computed. For example, a stride of 2 will lead to a halving of that dimension. The first stride dimension that lines up with the number of input channels can be set to any non-zero value.
- **sharing** (*bool*) – sharing flags for each input dimension
- **auto_padding** (*bool*) – flags for each input dimension whether it should be padded automatically (that is, symmetrically) or not padded at all. Padding means that the convolution kernel is applied to all pixel positions, where all pixels outside the area are assumed zero ("padded with zeroes"). Without padding, the kernels are only shifted over positions where all inputs to the kernel still fall inside the area. In this case, the output dimension will be less than the input dimension. The last value that lines up with the number of input channels must be false.
- **max_temp_mem_size_in_samples** (*int*) – maximum amount of auxiliary memory (in samples) that should be reserved to perform convolution operations. Some convolution engines (e.g. cuDNN and GEMM-based engines) can benefit from using workspace as it may improve performance. However, sometimes this may lead to higher memory utilization. Default is 0 which means the same as the input samples.
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

---

convolution_transpose(*convolution_map, operand, strides=(1,), sharing=[True], auto_padding=[True], output_shape=None, max_temp_mem_size_in_samples=0, name=''*) [source]

Computes the transposed convolution of `convolution_map` (typically a tensor of learnable parameters) with `operand` (commonly an image or output of a previous convolution/pooling operation). This is also known as `fractionally strided convolutional layers`, or, `deconvolution`. This operation is used in image and language processing applications. It supports arbitrary dimensions, strides, sharing, and padding.

This function operates on input tensors with dimensions $[C \times M_1 \times M_2 \times \ldots \times M_n]$. This can be understood as a rank-n object, where each entry consists of a C-dimensional vector. For example, an RGB image would have dimensions $[3 \times W \times H]$, i.e. a $[W \times H]$-sized structure, where each entry (pixel) consists of a 3-tuple.

convolution_transpose convolves the input `operand` with a $n+2$ rank tensor of (typically learnable) filters called `convolution_map` of shape $[I \times O \times m_1 \times m_2 \times \ldots \times m_n]$ (typically $m_i \ll M_i$). The first dimension, I, must match the number of channels in the input. The second dimension, O, is the number of convolution filters (i.e. the number of channels in the output). The last n dimensions are the spatial extent of the filter. I.e. for each output position, a vector of dimension O is computed. Hence, the total number of filter parameters is $I \times O \times m_1 \times m_2 \times \ldots \times m_n$

Example

```
>>> img = np.reshape(np.arange(9.0, dtype = np.float32), (1, 3, 3))
>>> x = C.input_variable(img.shape)
>>> filter = np.reshape(np.array([2, -1, -1, 2], dtype = np.float32), (1, 2, 2))
>>> kernel = C.constant(value = filter)
>>> np.round(C.convolution_transpose(kernel, x, auto_padding = [False]).eval({x: [img]}),5)
array([[[[  0.,   2.,   3.,  -2.],
         [  6.,   4.,   6.,  -1.],
         [  9.,  10.,  12.,   2.],
         [ -6.,   5.,   6.,  16.]]]], dtype=float32)
```

Parameters:

- **convolution_map** – convolution filter weights, stored as a tensor of dimensions $[I \times O \times m_1 \times m_2 \times \ldots \times m_n]$, where $[m_1 \times m_2 \times \ldots \times m_n]$ must be the kernel dimensions (spatial extent of the filter).
- **operand** – convolution input. A tensor with dimensions $[I \times M_1 \times M_2 \times \ldots \times M_n]$.
- **strides** (*tuple, optional*) – stride dimensions. If strides[i] > 1 then only pixel positions that are multiples of strides[i] are computed. For example, a stride of 2 will lead to a halving of that dimension. The first stride dimension that lines up with the number of input channels can be set to any non-zero value.
- **sharing** (*bool*) – sharing flags for each input dimension
- **auto_padding** (*bool*) – flags for each input dimension whether it should be padded automatically (that is, symmetrically) or not padded at all. Padding means that the convolution kernel is applied to all pixel positions, where all pixels outside the area are assumed zero ("padded with zeroes"). Without padding, the kernels are only shifted over positions where all inputs to the kernel still fall inside the area. In this case, the output dimension will be less than the input dimension. The last value that lines up with the number of input channels must be false.
- **output_shape** – user expected output shape after convolution transpose.
- **max_temp_mem_size_in_samples** (*int*) – maximum amount of auxiliary memory (in samples) that should be reserved to perform convolution operations. Some convolution engines (e.g. cuDNN and GEMM-based engines) can benefit from using workspace as it may improve performance. However, sometimes this may lead to higher memory utilization. Default is 0 which means the same as the input samples.
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

---

cos(*x, name=''*) [source]

Computes the element-wise cosine of `x`:

The output tensor has the same shape as `x`.

Example

```
>>> np.round(C.cos(np.arccos([[1,0.5],[-0.25,-0.75]])).eval(),5)
array([[ 1.  ,  0.5 ],
       [-0.25, -0.75]], dtype=float32)
```

Parameters:

- **x** – numpy array or any `Function` that outputs a tensor
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

cosh(*x, name=''*) [source]

Computes the element-wise cosh of `x`:

The output tensor has the same shape as `x`.

Example

```
>>> np.round(C.cosh([[1,0.5],[-0.25,-0.75]]).eval(),5)
array([[ 1.54308,  1.12763],
       [ 1.03141,  1.29468]], dtype=float32)
```

Parameters:
- **x** – numpy array or any `Function` that outputs a tensor
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

dropout(*x, dropout_rate=0.0, seed=4294967293, name=''*) [source]

Each element of the input is independently set to 0 with probabily `dropout_rate` or to 1 / (1 - `dropout_rate`) times its original value (with probability 1- `dropout_rate`). Dropout is a good way to reduce overfitting.

This behavior only happens during training. During inference dropout is a no-op. In the paper that introduced dropout it was suggested to scale the weights during inference In CNTK's implementation, because the values that are not set to 0 are multiplied with (1 / (1 - `dropout_rate`)), this is not necessary.

Example

```
>>> data = [[10, 20],[30, 40],[50, 60]]
>>> C.dropout(data, 0.5).eval()
array([[  0.,  40.],
       [  0.,  80.],
       [  0.,   0.]], dtype=float32)
```

```
>>> C.dropout(data, 0.75).eval()
array([[   0.,    0.],
       [   0.,  160.],
       [   0.,  240.]], dtype=float32)
```

Parameters:
- **x** – input tensor
- **dropout_rate** (*float, [0,1)*) – probability that an element of `x` will be set to zero
- **seed** (*int*) – random seed.
- **name** (`str` , optional) – the name of the Function instance in the network

Returns: `Function`

---

element_divide(*left, right, name=''*) [source]

The output of this operation is the element-wise division of the two input tensors. It supports broadcasting.

Example

```
>>> C.element_divide([1., 1., 1., 1.], [0.5, 0.25, 0.125, 0.]).eval()
array([ 2.,  4.,  8.,  0.], dtype=float32)
```

```
>>> C.element_divide([5., 10., 15., 30.], [2.]).eval()
array([  2.5,   5. ,   7.5,  15. ], dtype=float32)
```

Parameters:
- **left** – left side tensor
- **right** – right side tensor
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

---

element_max(*left, right, name=''*) [source]

The output of this operation is the element-wise max of the two or more input tensors. It supports broadcasting.

Parameters:
- **arg1** – left side tensor
- **arg2** – right side tensor
- ***more_args** – additional inputs
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

---

element_min(*left, right, name=''*) [source]

The output of this operation is the element-wise min of the two or more input tensors. It supports broadcasting.

Parameters:

- **arg1** – left side tensor
- **arg2** – right side tensor
- ***more_args** – additional inputs
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

---

element_select(*flag, value_if_true, value_if_false, name=''*) [source]

return either `value_if_true` or `value_if_false` based on the value of `flag`. If `flag` != 0 `value_if_true` is returned, otherwise `value_if_false`. Behaves analogously to numpy.where(...).

Example

```
>>> C.element_select([-10, -1, 0, 0.3, 100], [1, 10, 100, 1000, 10000], [ 2, 20, 200, 2000,
20000]).eval()
array([     1.,     10.,    200.,   1000.,  10000.], dtype=float32)
```

Parameters:

- **flag** – condition tensor
- **value_if_true** – true branch tensor
- **value_if_false** – false branch tensor
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

---

element_times(*left, right, name=''*) [source]

The output of this operation is the element-wise product of the two or more input tensors. It supports broadcasting.

Example

```
>>> C.element_times([1., 1., 1., 1.], [0.5, 0.25, 0.125, 0.]).eval()
array([ 0.5  ,  0.25 ,  0.125,  0.   ], dtype=float32)
```

```
>>> C.element_times([5., 10., 15., 30.], [2.]).eval()
array([ 10.,  20.,  30.,  60.], dtype=float32)
```

```
>>> C.element_times([5., 10., 15., 30.], [2.], [1., 2., 1., 2.]).eval()
array([  10.,   40.,   30.,  120.], dtype=float32)
```

Parameters:
- **arg1** – left side tensor
- **arg2** – right side tensor
- ***more_args** – additional inputs
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

---

elu(*x, name=''*) [source]

Exponential linear unit operation. Computes the element-wise exponential linear of `x`: `max(x, 0)` for `x >= 0` and `x`: `exp(x)-1` otherwise.

The output tensor has the same shape as `x`.

Example

```
>>> C.elu([[-1, -0.5, 0, 1, 2]]).eval()
array([[-0.632121, -0.393469,  0.      ,  1.      ,  2.      ]], dtype=float32)
```

Parameters:
- **x** (*numpy.array* or `Function`) – any `Function` that outputs a tensor.
- **name** (*str*, default to '') – the name of the Function instance in the network

Returns: An instance of `Function`

Return type: cntk.ops.functions.Function

---

equal(*left, right, name=''*) [source]

Elementwise 'equal' comparison of two tensors. Result is 1 if values are equal 0 otherwise.

Example

```
>>> C.equal([41., 42., 43.], [42., 42., 42.]).eval()
array([ 0.,  1.,  0.], dtype=float32)
```

```
>>> C.equal([-1,0,1], [1]).eval()
array([ 0.,  0.,  1.], dtype=float32)
```

Parameters:
- **left** – left side tensor
- **right** – right side tensor
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

exp(*x*, *name=''*) [source]

Computes the element-wise exponential of `x`:

$$\exp(x) = e^x$$

Example

```
>>> C.exp([0., 1.]).eval()
array([ 1.      ,  2.718282], dtype=float32)
```

Parameters:
- **x** – numpy array or any `Function` that outputs a tensor
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

floor(*arg*, *name=''*) [source]

The output of this operation is the element wise value rounded to the largest integer less than or equal to the input.

Example

```
>>> C.floor([0.2, 1.3, 4., 5.5, 0.0]).eval()
array([ 0.,  1.,  4.,  5.,  0.], dtype=float32)
```

```
>>> C.floor([[0.6, 3.3], [1.9, 5.6]]).eval()
array([[ 0.,  3.],
       [ 1.,  5.]], dtype=float32)
```

```
>>> C.floor([-5.5, -4.2, -3., -0.7, 0]).eval()
array([-6., -5., -3., -1.,  0.], dtype=float32)
```

```
>>> C.floor([[-0.6, -4.3], [1.9, -3.2]]).eval()
array([[-1., -5.],
       [ 1., -4.]], dtype=float32)
```

Parameters:
- **arg** – input tensor
- **name** (*str, optional*) – the name of the Function instance in the network (optional)

Returns: `Function`

`forward_backward`(*graph, features, blankTokenId, delayConstraint=-1, name=''*) [source]

Criterion node for training methods that rely on forward-backward Viterbi-like passes, e.g. Connectionist Temporal Classification (CTC) training The node takes as the input the graph of labels, produced by the labels_to_graph operation that determines the exact forward/backward procedure. .. rubric:: Example

graph = cntk.labels_to_graph(labels) networkOut = model(features) fb = C.forward_backward(graph, networkOut, 132)

Parameters:
- **graph** – labels graph
- **features** – network output
- **blankTokenId** – id of the CTC blank label
- **delayConstraint** – label output delay constraint introduced during training that allows to have shorter delay during inference. This is using the original time information to enforce that CTC tokens only get aligned within a time margin. Setting this parameter smaller will result in shorted delay between label output during decoding, yet may hurt accuracy. delayConstraint=-1 means no constraint

Returns: `Function`

`gather`(*reference, indices*) [source]

Retrieves the elements of indices in the tensor reference.

Example

```
>>> c = np.asarray([[[0],[1]],[[4],[5]]]).astype('f')
>>> x = C.input_variable((2,1))
>>> d = np.arange(12).reshape(6,2).astype('f')
>>> y = C.constant(d)
>>> C.gather(y, x).eval({x:c})
array([[[[  0.,   1.]],

        [[  2.,   3.]]],


       [[[  8.,   9.]],

        [[ 10.,  11.]]]], dtype=float32)
```

Parameters:
- **reference** – A tensor
- **indices** – An integer tensor of indices

Returns: `Function`

greater(*left*, *right*, *name=''*) [source]

Elementwise 'greater' comparison of two tensors. Result is 1 if left > right else 0.

Example

```
>>> C.greater([41., 42., 43.], [42., 42., 42.]).eval()
array([ 0.,  0.,  1.], dtype=float32)
```

```
>>> C.greater([-1,0,1], [0]).eval()
array([ 0.,  0.,  1.], dtype=float32)
```

Parameters:
- **left** – left side tensor
- **right** – right side tensor
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

greater_equal(*left*, *right*, *name=''*) [source]

Elementwise 'greater equal' comparison of two tensors. Result is 1 if left >= right else 0.

Example

```
>>> C.greater_equal([41., 42., 43.], [42., 42., 42.]).eval()
array([ 0.,  1.,  1.], dtype=float32)
```

```
>>> C.greater_equal([-1,0,1], [0]).eval()
array([ 0.,  1.,  1.], dtype=float32)
```

Parameters:
- **left** – left side tensor
- **right** – right side tensor
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

hardmax(*x*, *name=''*) [source]

Creates a tensor with the same shape as the input tensor, with zeros everywhere and a 1.0 where the maximum value of the input tensor is located. If the maximum value is repeated, 1.0 is placed in the first location found.

Example

```
>>> C.hardmax([1., 1., 2., 3.]).eval()
array([ 0.,  0.,  0.,  1.], dtype=float32)
```

```
>>> C.hardmax([1., 3., 2., 3.]).eval()
array([ 0.,  1.,  0.,  0.], dtype=float32)
```

Parameters:

- **x** – numpy array or any `Function` that outputs a tensor
- **name** (*str*) – the name of the Function instance in the network

Returns: `Function`

---

input(*shape, dtype=<cntk.default_options.default_override_or object>, needs_gradient=False, is_sparse=False, dynamic_axes=[Axis('defaultBatchAxis')], name=''*) [source]

DEPRECATED.

It creates an input in the network: a place where data, such as features and labels, should be provided.

Parameters:

- **shape** (*tuple or int*) – the shape of the input tensor
- **dtype** (*np.float32 or np.float64*) – data type. Default is np.float32.
- **needs_gradients** (*bool, optional*) – whether to back-propagates to it or not. False by default.
- **is_sparse** (*bool, optional*) – whether the variable is sparse (*False* by default)
- **dynamic_axes** (*list or tuple, default*) – a list of dynamic axis (e.g., batch axis, sequence axis)
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Variable`

---

input_variable(*shape, dtype=np.float32, needs_gradient=False, is_sparse=False, dynamic_axes=[Axis.default_batch_axis()], name=''*) [source]

It creates an input in the network: a place where data, such as features and labels, should be provided.

Parameters:

- **shape** (*tuple or int*) – the shape of the input tensor
- **dtype** (*np.float32 or np.float64*) – data type. Default is np.float32.
- **needs_gradients** (*bool, optional*) – whether to back-propagates to it or not. False by default.
- **is_sparse** (*bool, optional*) – whether the variable is sparse (*False* by default)
- **dynamic_axes** (*list or tuple, default*) – a list of dynamic axis (e.g., batch axis, time axis)
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Variable`

`labels_to_graph(labels, name='')` [source]

Conversion node from labels to graph. Typically used as an input to ForwardBackward node. This node's objective is to transform input labels into a graph representing exact forward-backward criterion.

Example

```
>>> num_classes = 2
>>> labels = C.input_variable((num_classes))
>>> graph = C.labels_to_graph(labels)
```

Parameters: **labels** – input training labels

Returns: `Function`

`leaky_relu(x, name='')` [source]

Leaky Rectified linear operation. Computes the element-wise leaky rectified linear of `x`: `max(x, 0)` for `x >= 0` and `x`: `0.01*x` otherwise.

The output tensor has the same shape as `x`.

Example

```
>>> C.leaky_relu([[-1, -0.5, 0, 1, 2]]).eval()
array([[-0.01 , -0.005,  0.   ,  1.   ,  2.   ]], dtype=float32)
```

Parameters:

- **x** (*numpy.array* or `Function`) – any `Function` that outputs a tensor.
- **name** (*str*, default to '') – the name of the Function instance in the network

Returns: An instance of `Function`

Return type: cntk.ops.functions.Function

less(*left, right, name=''*) [source]

Elementwise 'less' comparison of two tensors. Result is 1 if left < right else 0.

Example

```
>>> C.less([41., 42., 43.], [42., 42., 42.]).eval()
array([ 1.,  0.,  0.], dtype=float32)
```

```
>>> C.less([-1,0,1], [0]).eval()
array([ 1.,  0.,  0.], dtype=float32)
```

Parameters:
- **left** – left side tensor
- **right** – right side tensor
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

less_equal(*left, right, name=''*) [source]

Elementwise 'less equal' comparison of two tensors. Result is 1 if left <= right else 0.

Example

```
>>> C.less_equal([41., 42., 43.], [42., 42., 42.]).eval()
array([ 1.,  1.,  0.], dtype=float32)
```

```
>>> C.less_equal([-1,0,1], [0]).eval()
array([ 1.,  1.,  0.], dtype=float32)
```

Parameters:
- **left** – left side tensor
- **right** – right side tensor
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

log(*x, name=''*) [source]

Computes the element-wise the natural logarithm of `x`:

Example

```
>>> C.log([1., 2.]).eval()
array([ 0.      ,  0.693147], dtype=float32)
```

Parameters:
- **x** – numpy array or any `Function` that outputs a tensor
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

Note

CNTK returns -85.1 for log(x) if `x` is negative or zero. The reason is that it uses 1e-37 (whose natural logarithm is -85.1) as the smallest float number for *log*, because this is the only guaranteed precision across platforms. This will be changed to return *NaN* and *-inf*.

`log_add_exp`(*left, right, name=''*) [source]

Calculates the log of the sum of the exponentials of the two or more input tensors. It supports broadcasting.

Example

```
>>> a = np.arange(3,dtype=np.float32)
>>> np.exp(C.log_add_exp(np.log(1+a), np.log(1+a*a)).eval())
array([ 2.,  4.,  8.], dtype=float32)
>>> np.exp(C.log_add_exp(np.log(1+a), [0.]).eval())
array([ 2.,  3.,  4.], dtype=float32)
```

Parameters:
- **arg1** – left side tensor
- **arg2** – right side tensor
- ***more_args** – additional inputs
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

`minus`(*left, right, name=''*) [source]

The output of this operation is left minus right tensor. It supports broadcasting.

Example

```
>>> C.minus([1, 2, 3], [4, 5, 6]).eval()
array([-3., -3., -3.], dtype=float32)
```

```
>>> C.minus([[1,2],[3,4]], 1).eval()
array([[ 0.,  1.],
       [ 2.,  3.]], dtype=float32)
```

Parameters:
- **left** – left side tensor
- **right** – right side tensor
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

negate(*x, name=''*) [source]

Computes the element-wise negation of `x`:

$$negate(x) = -x$$

Example

```
>>> C.negate([-1, 1, -2, 3]).eval()
array([ 1., -1.,  2., -3.], dtype=float32)
```

Parameters:
- **x** – numpy array or any `Function` that outputs a tensor
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

not_equal(*left, right, name=''*) [source]

Elementwise 'not equal' comparison of two tensors. Result is 1 if left != right else 0.

Example

```
>>> C.not_equal([41., 42., 43.], [42., 42., 42.]).eval()
array([ 1.,  0.,  1.], dtype=float32)
```

```
>>> C.not_equal([-1,0,1], [0]).eval()
array([ 1.,  0.,  1.], dtype=float32)
```

Parameters:

- **left** – left side tensor
- **right** – right side tensor
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

---

one_hot(*x, num_classes, sparse_output=False, axis=-1, name=''*) [source]

Create one hot tensor based on the input tensor

Example

```
>>> data = np.asarray([[1, 2],
...                    [4, 5]], dtype=np.float32)
```

```
>>> x = C.input_variable((2,))
>>> C.one_hot(x, 6, False).eval({x:data})
array([[[ 0.,  1.,  0.,  0.,  0.,  0.],
        [ 0.,  0.,  1.,  0.,  0.,  0.]],

       [[ 0.,  0.,  0.,  0.,  1.,  0.],
        [ 0.,  0.,  0.,  0.,  0.,  1.]]], dtype=float32)
```

Parameters:

- **x** – input tensor, the value must be positive integer and less than num_class
- **num_classes** – the number of class in one hot tensor
- **sparse_output** – if set as True, we will create the one hot tensor as sparse.
- **axis** – The axis to fill (default: -1, a new inner-most axis).
- **name** (*str, optional, keyword only*) – the name of the Function instance in the network

Returns: `Function`

---

optimized_rnnstack(*operand, weights, hidden_size, num_layers, bidirectional=False, recurrent_op='lstm', name=''*) [source]

An RNN implementation that uses the primitives in cuDNN. If cuDNN is not available it fails. You can use `convert_optimized_rnnstack` to convert a model to GEMM-based implementation when no cuDNN.

Parameters:

- **operand** – input of the optimized RNN stack.
- **weights** – parameter tensor that holds the learned weights.
- **hidden_size** (*int*) – number of hidden units in each layer (and in each direction).
- **num_layers** (*int*) – number of layers in the stack.
- **bidirectional** (*bool, default False*) – whether each layer should compute both in forward and separately in backward mode and concatenate the results (if True the output is twice the hidden_size). The default is False which means the recurrence is only computed in the forward direction.
- **recurrent_op** (*str, optional*) – one of 'lstm', 'gru', 'relu', or 'tanh'.
- **name** (*str, optional*) – the name of the Function instance in the network

Example

```
>>> from _cntk_py import constant_initializer
>>> W = C.parameter((C.InferredDimension,4), constant_initializer(0.1))
>>> x = C.input_variable(shape=(4,))
>>> s = np.reshape(np.arange(20.0, dtype=np.float32), (5,4))
>>> t = np.reshape(np.arange(12.0, dtype=np.float32), (3,4))
>>> f = C.optimized_rnnstack(x, W, 8, 2)
>>> r = f.eval({x:[s,t]})
>>> len(r)
2
>>> print(*r[0].shape)
5 8
>>> print(*r[1].shape)
3 8
>>> r[0][:3,:]-r[1]
array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]], dtype=float32)
```

Returns: `Function`

`output_variable`(*shape, dtype, dynamic_axes, needs_gradient=True, name=''*) [source]

It creates an output variable that is used to define a user defined function.

Parameters:

- **shape** (*tuple or int*) – the shape of the input tensor
- **dtype** (*np.float32 or np.float64*) – data type
- **dynamic_axes** (*list or tuple*) – a list of dynamic axis (e.g., batch axis, time axis)
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Variable` that is of output type

`param_relu`(*alpha, x, name=''*) [source]

Parametric rectified linear operation. Computes the element-wise parameteric rectified linear of `x`: `max(x, 0)` for `x >= 0` and `x`: `alpha*x` otherwise.

The output tensor has the same shape as `x`.

Example

```
>>> alpha = C.constant(value=[[0.5, 0.5, 0.5, 0.5, 0.5]])
>>> C.param_relu(alpha, [[-1, -0.5, 0, 1, 2]]).eval()
array([[-0.5 , -0.25,  0.  ,  1.  ,  2.  ]], dtype=float32)
```

Parameters:

- **alpha** (`Parameter`) – same shape as x
- **x** (*numpy.array* or `Function`) – any `Function` that outputs a tensor.
- **name** (*str*, default to '') – the name of the Function instance in the network

Returns: An instance of `Function`

Return type: cntk.ops.functions.Function

`parameter`(*shape=None, init=None, dtype=None, device=None, name=''*) [source]

It creates a parameter tensor.

Example

```
>>> init_parameter = C.parameter(shape=(3,4), init=2)
>>> np.asarray(init_parameter)
array([[ 2.,  2.,  2.,  2.],
       [ 2.,  2.,  2.,  2.],
       [ 2.,  2.,  2.,  2.]], dtype=float32)
```

Parameters:

- **shape** (*tuple or int, optional*) – the shape of the input tensor. If not provided, it will be inferred from `value` .
- **init** (*scalar or NumPy array or initializer*) – if init is a scalar it will be replicated for every element in the tensor or NumPy array. If it is the output of an initializer form `cntk.initializer` it will be used to initialize the tensor at the first forward pass. If *None*, the tensor will be initialized with 0.
- **dtype** (*optional*) – data type of the constant. If a NumPy array and `dtype` , are given, then data will be converted if needed. If none given, it will default to `np.float32` .
- **device** (`DeviceDescriptor`) – instance of DeviceDescriptor
- **name** (*str, optional*) – the name of the Parameter instance in the network

Returns: `Parameter`

`per_dim_mean_variance_normalize`(*operand, mean, inv_stddev, name=''*) [source]

Computes per dimension mean-variance normalization of the specified input operand.

Parameters:

- **operand** – the variable to be normalized
- **mean** (*NumPy array*) – per dimension mean to use for the normalization
- **inv_stddev** (*NumPy array*) – per dimension standard deviation to use for the normalization
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

---

placeholder(*shape=None, dynamic_axes=None, name=''*) [source]

It creates a placeholder variable that has to be later bound to an actual variable. A common use of this is to serve as a placeholder for a later output variable in a recurrent network, which is replaced with the actual output variable by calling replace_placeholder(s).

Parameters:

- **shape** (*tuple or int*) – the shape of the variable tensor
- **dynamic_axes** (*list*) – the list of dynamic axes that the actual variable uses
- **name** (*str, optional*) – the name of the placeholder variable in the network

Returns: `Variable`

---

plus(*left, right, name=''*) [source]

The output of this operation is the sum of the two or more input tensors. It supports broadcasting.

Example

```
>>> C.plus([1, 2, 3], [4, 5, 6]).eval()
array([ 5.,  7.,  9.], dtype=float32)
```

```
>>> C.plus([-5, -4, -3, -2, -1], [10]).eval()
array([ 5.,  6.,  7.,  8.,  9.], dtype=float32)
```

```
>>> C.plus([-5, -4, -3, -2, -1], [10], [3, 2, 3, 2, 3], [-13], [+42],
'multi_arg_example').eval()
array([ 37.,  37.,  39.,  39.,  41.], dtype=float32)
```

```
>>> C.plus([-5, -4, -3, -2, -1], [10], [3, 2, 3, 2, 3]).eval()
array([  8.,   8.,  10.,  10.,  12.], dtype=float32)
```

Parameters:

- **arg1** – left side tensor
- **arg2** – right side tensor
- ***more_args** – additional inputs
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

---

pooling(*operand, pooling_type, pooling_window_shape, strides=(1,), auto_padding=[False], ceil_out_dim=False, include_pad=False, name=''*) [source]

The pooling operations compute a new tensor by selecting the maximum or average value in the pooling input. In the case of average pooling with padding, the average is only over the valid region.

N-dimensional pooling allows to create max or average pooling of any dimensions, stride or padding.

Example

```
>>> img = np.reshape(np.arange(16, dtype = np.float32), [1, 4, 4])
>>> x = C.input_variable(img.shape)
>>> C.pooling(x, C.AVG_POOLING, (2,2), (2,2)).eval({x : [img]})
array([[[[  2.5,   4.5],
         [ 10.5,  12.5]]]], dtype=float32)
>>> C.pooling(x, C.MAX_POOLING, (2,2), (2,2)).eval({x : [img]})
array([[[[  5.,   7.],
         [ 13.,  15.]]]], dtype=float32)
```

Parameters:

- **operand** – pooling input
- **pooling_type** – one of `MAX_POOLING` or `AVG_POOLING`
- **pooling_window_shape** – dimensions of the pooling window
- **strides** (*default 1*) – strides.
- **auto_padding** (*default [False,]*) – automatic padding flags for each input dimension.
- **ceil_out_dim** (*default False*) – ceiling while computing output size
- **include_pad** (*default False*) – include pad while average pooling
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

---

pow(*base, exponent, name=''*) [source]

Computes *base* raised to the power of *exponent*. It supports broadcasting. This is well defined if *base* is non-negative or *exponent* is an integer. Otherwise the result is NaN. The gradient with respect to the base is well defined if the forward operation is well defined. The gradient with respect to the exponent is well defined if the base is non-negative, and it is set to 0 otherwise.

Example

```
>>> C.pow([1, 2, -2], [3, -2, 3]).eval()
array([ 1.  ,  0.25, -8.  ], dtype=float32)
```

```
>>> C.pow([[0.5, 2],[4, 1]], -2).eval()
array([[ 4.    ,  0.25  ],
       [ 0.0625,  1.    ]], dtype=float32)
```

Parameters:

- **base** – base tensor
- **exponent** – exponent tensor
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

---

random_sample(*weights, num_samples, allow_duplicates, seed=4294967293, name=''*) [source]

Estimates inclusion frequencies for random sampling with or without replacement.

The output value is a set of num_samples random samples represented by a (sparse) matrix of shape [num_samples x len(weights)], where len(weights) is the number of classes (categories) to choose from. The output has no dynamic axis. The samples are drawn according to the weight vector p(i) = weights[i] / sum(weights) We get one set of samples per minibatch. Intended use cases are e.g. sampled softmax, noise contrastive estimation etc.

Parameters:

- **weights** – input vector of sampling weights which should be non-negative numbers.
- **num_samples** (*int*) – number of expected samples
- **allow_duplicates** (*bool*) – If sampling is done with replacement (*True*) or without (*False*).
- **seed** (*int*) – random seed.
- **name** (`str` , optional) – the name of the Function instance in the network.

Returns: `Function`

---

random_sample_inclusion_frequency(*weights, num_samples, allow_duplicates, seed=4294967293, name=''*) [source]

For weighted sampling with the specifed sample size (*num_samples*) this operation computes the expected number of occurrences of each class in the sampled set. In case of sampling without replacement the result is only an estimate which might be quite rough in the case of small sample sizes. Intended uses are e.g. sampled softmax, noise contrastive estimation etc. This operation will be typically used together with `random_sample()` .

Parameters:

- **weights** – input vector of sampling weights which should be non-negative numbers.
- **num_samples** (*int*) – number of expected samples
- **allow_duplicates** (*bool*) – If sampling is done with replacement (*True*) or without (*False*).
- **seed** (*int*) – random seed.
- **name** (`str` , optional) – the name of the Function instance in the network.

Example

```
>>> import numpy as np
>>> from cntk import *
>>> # weight vector with 100 '1000'-values followed
>>> # by 100 '1' values
>>> w1 = np.full((100),1000, dtype = np.float)
>>> w2 = np.full((100),1, dtype = np.float)
>>> w = np.concatenate((w1, w2))
>>> f = random_sample_inclusion_frequency(w, 150, True).eval()
>>> f[0]
1.4985015
>>> f[1]
1.4985015
>>> f[110]
0.0014985015
>>> # when switching to sampling without duplicates samples are
>>> # forced to pick the low weight classes too
>>> f = random_sample_inclusion_frequency(w, 150, False).eval()
>>> f[0]
1.0
```

Returns: `Function`

reciprocal(*x, name=''*) [source]

Computes the element-wise reciprocal of `x` :

Example

```
>>> C.reciprocal([-1/3, 1/5, -2, 3]).eval()
array([-3.      ,  5.      , -0.5     ,  0.333333], dtype=float32)
```

Parameters:

- **x** – numpy array or any `Function` that outputs a tensor
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

reconcile_dynamic_axes(*x, dynamic_axes_as, name=''*) [source]

Create a new Function instance which reconciles the dynamic axes of the specified tensor operands. The output of the returned Function has the sample layout of the 'x' operand and the dynamic axes of the 'dynamic_axes_as' operand. This operator also performs a runtime check to ensure that the dynamic axes layouts of the 2 operands indeed match.

Parameters:

- **x** – The Function/Variable, whose dynamic axes are to be reconciled
- **dynamic_axes_as** – The Function/Variable, to whose dynamic axes the operand 'x''s dynamic axes are reconciled to.
- **name** (*str, optional*) – the name of the reconcile_dynamic_axes Function in the network

Returns: `Function`

`reduce_log_sum_exp`(*x, axis=None, name=''*) [source]

Computes the log of the sum of the exponentiations of the input tensor's elements across a specified axis or a list of specified axes.

Example

```
>>> # create 3x2x2 matrix in a sequence of length 1 in a batch of one sample
>>> data = np.array([[[5,1], [20,2]],[[30,1], [40,2]],[[55,1], [60,2]]], dtype=np.float32)
```

```
>>> C.reduce_log_sum_exp(data, axis=0).eval().round(4)
array([[[ 55.      ,   2.0986],
        [ 60.      ,   3.0986]]], dtype=float32)
>>> np.log(np.sum(np.exp(data), axis=0)).round(4)
array([[ 55.      ,   2.0986],
       [ 60.      ,   3.0986]], dtype=float32)
>>> C.reduce_log_sum_exp(data, axis=(0,2)).eval().round(4)
array([[[ 55.],
        [ 60.]]], dtype=float32)
>>> np.log(np.sum(np.exp(data), axis=(0,2))).round(4)
array([ 55.,  60.], dtype=float32)
```

```
>>> x = C.input_variable(shape=(2,2))
>>> lse = C.reduce_log_sum_exp(x, axis=[C.axis.Axis.default_batch_axis(), 1])
>>> lse.eval({x:data}).round(4)
array([[ 55.],
       [ 60.]], dtype=float32)
>>> np.log(np.sum(np.exp(data), axis=(0,2))).round(4)
array([ 55.,  60.], dtype=float32)
```

Parameters:

- **x** – input tensor
- **axis** (int or `Axis` or a `list` or `tuple` of int or `Axis`) – axis along which the reduction will be performed
- **name** (*str*) – the name of the Function instance in the network

Returns: `Function`

Note that CNTK keeps the shape of the resulting tensors when reducing over multiple static axes.

reduce_max(*x, axis=None, name=''*) [source]

Computes the max of the input tensor's elements across a specified axis or a list of specified axes.

Example

```
>>> # create 3x2x2 matrix in a sequence of length 1 in a batch of one sample
>>> data = np.array([[[5,1], [20,2]],[[30,1], [40,2]],[[55,1], [60,2]]], dtype=np.float32)
```

```
>>> C.reduce_max(data, 0).eval().round(4)
array([[[ 55.,   1.],
        [ 60.,   2.]]], dtype=float32)
>>> C.reduce_max(data, 1).eval().round(4)
array([[[ 20.,   2.]],

       [[ 40.,   2.]],

       [[ 60.,   2.]]], dtype=float32)
>>> C.reduce_max(data, (0,2)).eval().round(4)
array([[[ 55.],
        [ 60.]]], dtype=float32)
```

```
>>> x = C.input_variable((2,2))
>>> C.reduce_max( x * 1.0, (C.Axis.default_batch_axis(), 1)).eval({x: data}).round(4)
array([[[ 55.],
       [ 60.]]], dtype=float32)
```

Parameters:

- **x** – input tensor
- **axis** (int or `Axis` or a `list` or `tuple` of int or `Axis`) – axis along which the reduction will be performed
- **name** (*str*) – the name of the Function instance in the network

Returns: `Function`

Note that CNTK keeps the shape of the resulting tensors when reducing over multiple static axes.

reduce_mean(*x, axis=None, name=''*) [source]

Computes the mean of the input tensor's elements across a specified axis or a list of specified axes.

Example

```
>>> # create 3x2x2 matrix in a sequence of length 1 in a batch of one sample
>>> data = np.array([[[5,1], [20,2]],[[30,1], [40,2]],[[55,1], [60,2]]], dtype=np.float32)
```

```
>>> C.reduce_mean(data, 0).eval().round(4)
array([[[ 30.,   1.],
        [ 40.,   2.]]], dtype=float32)
>>> np.mean(data, axis=0).round(4)
array([[ 30.,   1.],
       [ 40.,   2.]], dtype=float32)
>>> C.reduce_mean(data, 1).eval().round(4)
array([[[ 12.5,   1.5]],

       [[ 35. ,   1.5]],

       [[ 57.5,   1.5]]], dtype=float32)
>>> np.mean(data, axis=1).round(4)
array([[ 12.5,   1.5],
       [ 35. ,   1.5],
       [ 57.5,   1.5]], dtype=float32)
>>> C.reduce_mean(data, (0,2)).eval().round(4)
array([[[ 15.5],
        [ 21. ]]], dtype=float32)
```

```
>>> x = C.input_variable((2,2))
>>> C.reduce_mean( x * 1.0, (C.Axis.default_batch_axis(), 1)).eval({x: data}).round(4)
array([[ 15.5],
       [ 21.      ]], dtype=float32)
```

Parameters:

- **x** – input tensor
- **axis** (int or `Axis` or a `list` or `tuple` of int or `Axis`) – axis along which the reduction will be performed
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

Note that CNTK keeps the shape of the resulting tensors when reducing over multiple static axes.

---

reduce_min(*x, axis=None, name=''*) [source]

Computes the min of the input tensor's elements across a specified axis or a list of specified axes.

Example

```
>>> # create 3x2x2 matrix in a sequence of length 1 in a batch of one sample
>>> data = np.array([[[5,1], [20,2]],[[30,1], [40,2]],[[55,1], [60,2]]], dtype=np.float32)
```

```
>>> C.reduce_min(data, 0).eval().round(4)
array([[[  5.,   1.],
        [ 20.,   2.]]], dtype=float32)
>>> C.reduce_min(data, 1).eval().round(4)
array([[[  5.,   1.]],

       [[ 30.,   1.]],

       [[ 55.,   1.]]], dtype=float32)
>>> C.reduce_min(data, (0,2)).eval().round(4)
array([[[ 1.],
        [ 2.]]], dtype=float32)
```

```
>>> x = C.input_variable((2,2))
>>> C.reduce_min( x * 1.0, (C.Axis.default_batch_axis(), 1)).eval({x: data}).round(4)
array([[ 1.],
       [ 2.]], dtype=float32)
```

Parameters:

- **x** – input tensor
- **axis** (int or `Axis` or a list of integers or a list of `Axis`) – axis along which the reduction will be performed
- **name** (*str*) – the name of the Function instance in the network

Returns:

`Function`

Note that CNTK keeps the shape of the resulting tensors when reducing over multiple static axes.

---

reduce_prod(*x, axis=None, name=''*) [source]

Computes the min of the input tensor's elements across the specified axis.

Example

```
>>> # create 3x2x2 matrix in a sequence of length 1 in a batch of one sample
>>> data = np.array([[[5,1], [20,2]],[[30,1], [40,2]],[[55,1], [60,2]]], dtype=np.float32)
```

```
>>> C.reduce_prod(data, 0).eval().round(4)
array([[[  8250.,      1.],
        [ 48000.,      8.]]], dtype=float32)
>>> C.reduce_prod(data, 1).eval().round(4)
array([[[  100.,     2.]],

       [[ 1200.,     2.]],

       [[ 3300.,     2.]]], dtype=float32)
>>> C.reduce_prod(data, (0,2)).eval().round(4)
array([[[   8250.],
        [ 384000.]]], dtype=float32)
```

```
>>> x = C.input_variable((2,2))
>>> C.reduce_prod( x * 1.0, (C.Axis.default_batch_axis(), 1)).eval({x: data}).round(4)
array([[   8250.],
       [ 384000.]], dtype=float32)
```

Parameters:

- **x** – input tensor
- **axis** (int or `Axis` or a `list` or `tuple` of int or `Axis`) – axis along which the reduction will be performed
- **name** (*str*) – the name of the Function instance in the network

Returns: `Function`

Note that CNTK keeps the shape of the resulting tensors when reducing over multiple static axes.

---

reduce_sum(*x, axis=None, name=''*) [source]

Computes the sum of the input tensor's elements across one axis or a list of axes. If the axis parameter is not specified then the sum will be computed over all static axes, which is equivalent with specifying `axis=Axis.all_static_axes()`. If `axis=Axis.all_axes()` is specified, then the output is a scalar which is the sum of all the elements in the minibatch. And if `axis=Axis.default_batch_axis()` is specified, then the reduction will happen across the batch axis (In this case the input must not be a sequence).

Example

```
>>> # create 3x2x2 matrix in a sequence of length 1 in a batch of one sample
>>> data = np.array([[[5,1], [20,2]],[[30,1], [40,2]],[[55,1], [60,2]]], dtype=np.float32)
```

```
>>> C.reduce_sum(data, 0).eval().round(4)
array([[[  90.,    3.],
        [ 120.,    6.]]], dtype=float32)
>>> np.sum(data, axis=0).round(4)
array([[  90.,    3.],
       [ 120.,    6.]], dtype=float32)
>>> C.reduce_sum(data, 1).eval().round(4)
array([[[  25.,    3.]],

       [[  70.,    3.]],

       [[ 115.,    3.]]], dtype=float32)
>>> np.sum(data, axis=1).round(4)
array([[  25.,    3.],
       [  70.,    3.],
       [ 115.,    3.]], dtype=float32)
>>> C.reduce_sum(data, (0,2)).eval().round(4)
array([[[  93.],
        [ 126.]]], dtype=float32)
```

Parameters:

- **x** – input tensor
- **axis** (int or `Axis` or a `list` or `tuple` of int or `Axis`) – axis along which the reduction will be performed
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

Note that CNTK keeps the shape of the resulting tensors when reducing over multiple static axes.

---

relu(*x, name=''*) [source]

Rectified linear operation. Computes the element-wise rectified linear of `x`: `max(x, 0)`

The output tensor has the same shape as `x`.

Example

```
>>> C.relu([[-1, -0.5, 0, 1, 2]]).eval()
array([[ 0.,  0.,  0.,  1.,  2.]], dtype=float32)
```

Parameters:

- **x** – numpy array or any `Function` that outputs a tensor
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

---

reshape(*x, shape, begin_axis=None, end_axis=None, name=''*) [source]

Reinterpret input samples as having different tensor dimensions One dimension may be specified as 0 and will be inferred

The output tensor has the shape specified by 'shape'.

Example

```
>>> i1 = C.input_variable(shape=(3,2))
>>> C.reshape(i1, (2,3)).eval({i1:np.asarray([[[[0., 1.],[2., 3.],[4., 5.]]]],
dtype=np.float32)})
array([[[ 0.,  1.,  2.],
        [ 3.,  4.,  5.]]], dtype=float32)
```

Parameters:

- **x** – tensor to be reshaped
- **shape** (*tuple*) – a tuple defining the resulting shape. The specified shape tuple may contain -1 for at most one axis, which is automatically inferred to the correct dimension size by dividing the total size of the sub-shape being reshaped with the product of the dimensions of all the non-inferred axes of the replacement shape.
- **begin_axis** (*int or None*) – shape replacement begins at this axis. Negative values are counting from the end. *None* is the same as 0. To refer to the end of the shape tuple, pass *Axis.new_leading_axis()*.
- **end_axis** (*int or None*) – shape replacement ends at this axis (excluding this axis). Negative values are counting from the end. *None* refers to the end of the shape tuple.
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

---

roipooling(*operand, rois, pooling_type, roi_output_shape, spatial_scale, name=''*) [source]

The ROI (Region of Interest) pooling operation pools over sub-regions of an input volume and produces a fixed sized output volume regardless of the ROI size. It is used for example for object detection.

Each input image has a fixed number of regions of interest, which are specified as bounding boxes (x, y, w, h) that are relative to the image size [W x H]. This operation can be used as a replacement for the final pooling layer of an image classification network (as presented in Fast R-CNN and others).

Changed in version 2.1: The signature was updated to match the Caffe implementation: the parameters *pooling_type* and *spatial_scale* were added, and the coordinates for the parameters *rois* are now absolute to the original image size.

Parameters:

- **operand** – a convolutional feature map as the input volume ([W x H x C x N]).
- **pooling_type** – only `MAX_POOLING`
- **rois** – the coordinates of the ROIs per image ([4 x roisPerImage x N]), each ROI is (x1, y1, x2, y2) absolute to original image size.
- **roi_output_shape** – dimensions (width x height) of the ROI pooling output shape
- **spatial_scale** – the scale of operand from the original image size.
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

---

round(*arg, name=''*) [source]

The output of this operation is the element wise value rounded to the nearest integer. In case of tie, where element can have exact fractional part of 0.5 this operation follows “round half-up” tie breaking strategy. This is different from the round operation of numpy which follows round half to even.

Example

```
>>> C.round([0.2, 1.3, 4., 5.5, 0.0]).eval()
array([ 0.,  1.,  4.,  6.,  0.], dtype=float32)
```

```
>>> C.round([[0.6, 3.3], [1.9, 5.6]]).eval()
array([[ 1.,  3.],
       [ 2.,  6.]], dtype=float32)
```

```
>>> C.round([-5.5, -4.2, -3., -0.7, 0]).eval()
array([-5., -4., -3., -1.,  0.], dtype=float32)
```

```
>>> C.round([[-0.6, -4.3], [1.9, -3.2]]).eval()
array([[-1., -4.],
       [ 2., -3.]], dtype=float32)
```

Parameters:
- **arg** – input tensor
- **name** (*str, optional*) – the name of the Function instance in the network (optional)

Returns: `Function`

`selu`(*x, scale=1.0507009873554805, alpha=1.6732632423543772, name=''*) [source]

Scaled exponential linear unit operation. Computes the element-wise exponential linear of `x`: `scale * x` for `x >= 0` and `x`: `scale * alpha * (exp(x)-1)` otherwise.

The output tensor has the same shape as `x`.

Example

```
>>> C.selu([[-1, -0.5, 0, 1, 2]]).eval()
array([[-1.111331, -0.691758,  0.      ,  1.050701,  2.101402]], dtype=float32)
```

Parameters:
- **x** (*numpy.array* or `Function`) – any `Function` that outputs a tensor.
- **name** (*str*, default to '') – the name of the Function instance in the network

Returns: An instance of `Function`

Return type: cntk.ops.functions.Function

sigmoid(*x*, *name=''*) [source]

Computes the element-wise sigmoid of `x`:

$$sigmoid(x) = \frac{1}{1+\exp(-x)}$$

The output tensor has the same shape as `x`.

Example

```
>>> C.sigmoid([-2, -1., 0., 1., 2.]).eval()
array([ 0.119203,  0.268941,  0.5     ,  0.731059,  0.880797], dtype=float32)
```

Parameters:
- **x** – numpy array or any `Function` that outputs a tensor
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

sin(*x*, *name=''*) [source]

Computes the element-wise sine of `x`:

The output tensor has the same shape as `x`.

Example

```
>>> np.round(C.sin(np.arcsin([[1,0.5],[-0.25,-0.75]])).eval(),5)
array([[ 1.  ,  0.5 ],
       [-0.25, -0.75]], dtype=float32)
```

Parameters:
- **x** – numpy array or any `Function` that outputs a tensor
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

sinh(*x*, *name=''*) [source]

Computes the element-wise sinh of `x`:

The output tensor has the same shape as `x`.

Example

```
>>> np.round(C.sinh([[1,0.5],[-0.25,-0.75]]).eval(),5)
array([[ 1.1752 ,  0.5211 ],
       [-0.25261, -0.82232]], dtype=float32)
```

Parameters:

- **x** – numpy array or any `Function` that outputs a tensor
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

`slice`(*x, axis, begin_index, end_index, strides=None, name=''*) [source]

Slice the input along one or multiple axes.

Example

```
>>> # slice using input variable
>>> # create 2x3 matrix
>>> x1 = C.input_variable((2,3))
>>> # slice index 1 (second) at first axis
>>> C.slice(x1, 0, 1, 2).eval({x1: np.asarray([[[1,2,-3],
...                                             [4, 5, 6]]],dtype=np.float32)})
array([[[ 4.,  5.,  6.]]], dtype=float32)

>>> # slice index 0 (first) at second axis
>>> C.slice(x1, 1, 0, 1).eval({x1: np.asarray([[[1,2,-3],
...                                             [4, 5, 6]]],dtype=np.float32)})
array([[[ 1.],
        [ 4.]]], dtype=float32)
>>> # slice with strides
>>> C.slice(x1, 0, 0, 2, 2).eval({x1: np.asarray([[[1,2,-3],
...                                               [4, 5, 6]]],dtype=np.float32)})
array([[[ 1.,  2., -3.]]], dtype=float32)

>>> # reverse
>>> C.slice(x1, 0, 0, 2, -1).eval({x1: np.asarray([[[1,2,-3],
...                                                [4, 5, 6]]],dtype=np.float32)})
array([[[ 4.,  5.,  6.],
[ 1.,  2., -3.]]], dtype=float32)

>>> # slice along multiple axes
>>> C.slice(x1, [0,1], [1,0], [2,1]).eval({x1: np.asarray([[[1, 2, -3],
...                                                        [4, 5, 6]]],dtype=np.float32)})
array([[[ 4.]]], dtype=float32)

>>> # slice using constant
>>> data = np.asarray([[1, 2, -3],
...                    [4, 5,  6]], dtype=np.float32)
>>> x = C.constant(value=data)
>>> C.slice(x, 0, 1, 2).eval()
array([[ 4.,  5.,  6.]], dtype=float32)
>>> C.slice(x, 1, 0, 1).eval()
array([[ 1.],
       [ 4.]], dtype=float32)
>>> C.slice(x, [0,1], [1,0], [2,1]).eval()
array([[ 4.]], dtype=float32)

>>> # slice using the index overload
>>> data = np.asarray([[1, 2, -3],
...                    [4, 5,  6]], dtype=np.float32)
>>> x = C.constant(value=data)
>>> x[0].eval()
array([[ 1.,  2.,  -3.]], dtype=float32)
>>> x[0, [1,2]].eval()
array([[ 2.,  -3.]], dtype=float32)

>>> x[1].eval()
array([[ 4.,  5.,  6.]], dtype=float32)
>>> x[:,:2,:].eval()
array([[ 1.,  2.],
       [ 4.,  5.]], dtype=float32)
```

Parameters:

- **x** – input tensor
- **axis** (int or `Axis`) – axis along which `begin_index` and `end_index` will be used. If it is of type int it will be used as a static axis.
- **begin_index** (*int*) – the index along axis where the slicing starts
- **end_index** (*int*) – the index along axis where the slicing ends
- **name** (*str, optional*) – the name of the Function instance in the network
- **strides** (*int*) – step sizes when applying slice, negative value means in reverse order

See also

Indexing in NumPy: https://docs.scipy.org/doc/numpy/reference/arrays.indexing.html

Returns: `Function`

---

softmax(*x, axis=None, name=''*) [source]

Computes the gradient of $f(z) = \log \sum_i \exp(z_i)$ at `z = x`. Concretely,

$$\text{softmax}(x) = \left[\frac{\exp(x_1)}{\sum_i \exp(x_i)} \quad \frac{\exp(x_1)}{\sum_i \exp(x_i)} \quad \cdots \quad \frac{\exp(x_1)}{\sum_i \exp(x_i)} \right]$$

with the understanding that the implementation can use equivalent formulas for efficiency and numerical stability.

The output is a vector of non-negative numbers that sum to 1 and can therefore be interpreted as probabilities for mutually exclusive outcomes as in the case of multiclass classification.

If `axis` is given as integer, then the softmax will be computed along that axis. If the provided `axis` is -1, it will be computed along the last axis. Otherwise, softmax will be applied to all axes.

Example

```
>>> C.softmax([[1, 1, 2, 3]]).eval()
array([[ 0.082595,  0.082595,  0.224515,  0.610296]], dtype=float32)
```

```
>>> C.softmax([1, 1]).eval()
array([ 0.5,  0.5], dtype=float32)
```

```
>>> C.softmax([[[1, 1], [3, 5]]], axis=-1).eval()
array([[[ 0.5     ,  0.5     ],
        [ 0.119203,  0.880797]]], dtype=float32)
```

```
>>> C.softmax([[[1, 1], [3, 5]]], axis=1).eval()
array([[[ 0.119203,  0.017986],
        [ 0.880797,  0.982014]]], dtype=float32)
```

Parameters:
- **x** – numpy array or any `Function` that outputs a tensor
- **axis** (int or `Axis`) – axis along which the softmax operation will be performed
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

---

softplus(*x, steepness=1, name=''*) [source]

Softplus operation. Computes the element-wise softplus of `x`:

$$\text{softplus}(x) = \log(1 + \exp(x))$$

The optional `steepness` allows to make the knee sharper (`steepness>1`) or softer, by computing `softplus(x * steepness) / steepness`. (For very large steepness, this approaches a linear rectifier).

The output tensor has the same shape as `x`.

Example

```
>>> C.softplus([[-1, -0.5, 0, 1, 2]]).eval()
array([[ 0.313262,  0.474077,  0.693147,  1.313262,  2.126928]], dtype=float32)
```

```
>>> C.softplus([[-1, -0.5, 0, 1, 2]], steepness=4).eval()
array([[ 0.004537,  0.031732,  0.173287,  1.004537,  2.000084]], dtype=float32)
```

Parameters:
- **x** (*numpy.array* or `Function`) – any `Function` that outputs a tensor.
- **steepness** (*float, optional*) – optional steepness factor
- **name** (*str*, default to '') – the name of the Function instance in the network

Returns: An instance of `Function`

Return type: cntk.ops.functions.Function

---

splice(**inputs, **kw_axis_name*) [source]

Concatenate the input tensors along an axis.

Example

```
>>> # create 2x2 matrix in a sequence of length 1 in a batch of one sample
>>> data1 = np.asarray([[[1, 2],
...                      [4, 5]]], dtype=np.float32)
```

```
>>> x = C.constant(value=data1)
>>> # create 3x2 matrix in a sequence of length 1 in a batch of one sample
>>> data2 = np.asarray([[[10, 20],
...                      [30, 40],
...                      [50, 60]]],dtype=np.float32)
>>> y = C.constant(value=data2)
>>> # splice both inputs on axis=0 returns a 5x2 matrix
>>> C.splice(x, y, axis=1).eval()
array([[[  1.,   2.],
        [  4.,   5.],
        [ 10.,  20.],
        [ 30.,  40.],
        [ 50.,  60.]]], dtype=float32)
```

Parameters:

- **inputs** – one or more input tensors
- **axis** (int or `Axis`, optional, keyword only) – axis along which the concatenation will be performed
- **name** (*str, optional, keyword only*) – the name of the Function instance in the network

Returns: `Function`

sqrt(*x, name=''*) [source]

Computes the element-wise square-root of `x`:

$$sqrt(x) = \sqrt[2]{x}$$

Example

```
>>> C.sqrt([0., 4.]).eval()
array([ 0.,  2.], dtype=float32)
```

Parameters:

- **x** – numpy array or any `Function` that outputs a tensor
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

Note

CNTK returns zero for sqrt of negative nubmers, this will be changed to return NaN

square(*x, name=''*) [source]

Computes the element-wise square of `x`:

Example

```
>>> C.square([1., 10.]).eval()
array([   1.,  100.], dtype=float32)
```

Parameters:
- **x** – numpy array or any `Function` that outputs a tensor
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

stop_gradient(*input, name=''*) [source]

Outputs its input as it is and prevents any gradient contribution from its output to its input.

Parameters:
- **input** – class:*~cntk.ops.functions.Function* that outputs a tensor
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

swapaxes(*x, axis1=0, axis2=1, name=''*) [source]

Swaps two axes of the tensor. The output tensor has the same data but with `axis1` and `axis2` swapped.

Example

```
>>> C.swapaxes([[[0,1],[2,3],[4,5]]], 1, 2).eval()
array([[[ 0.,  2.,  4.],
        [ 1.,  3.,  5.]]], dtype=float32)
```

Parameters:
- **x** – tensor to be transposed
- **axis1** (int or `Axis`) – the axis to swap with `axis2`
- **axis2** (int or `Axis`) – the axis to swap with `axis1`
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

tanh(*x, name=''*) [source]

Computes the element-wise tanh of `x`:

The output tensor has the same shape as `x`.

Example

```
>>> C.tanh([[1,2],[3,4]]).eval()
array([[ 0.761594,  0.964028],
       [ 0.995055,  0.999329]], dtype=float32)
```

Parameters:

- **x** – numpy array or any `Function` that outputs a tensor
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

`times`(*left, right, output_rank=1, infer_input_rank_to_map=-1, name=''*) [source]

The output of this operation is the matrix product of the two input matrices. It supports broadcasting. Sparse is supported in the left operand, if it is a matrix. The operator '@' has been overloaded such that in Python 3.5 and later X @ W equals times(X, W).

For better performance on times operation on sequence which is followed by sequence.reduce_sum, use infer_input_rank_to_map=TIMES_REDUCE_SEQUENCE_AXIS_WITHOUT_INFERRED_INPUT_RANK, i.e. replace following:

```
sequence.reduce_sum(times(seq1, seq2))
```

with:

```
times(seq1, seq2,
infer_input_rank_to_map=TIMES_REDUCE_SEQUENCE_AXIS_WITHOUT_INFERRED_INPUT_RANK)
```

Example

```
>>> C.times([[1,2],[3,4]], [[5],[6]]).eval()
array([[ 17.],
       [ 39.]], dtype=float32)
```

```
>>> C.times(1.*np.reshape(np.arange(8), (2,2,2)),1.*np.reshape(np.arange(8), (2,2,2)),
output_rank=1).eval()
array([[ 28.,  34.],
       [ 76.,  98.]])
```

```
>>> C.times(1.*np.reshape(np.arange(8), (2,2,2)),1.*np.reshape(np.arange(8), (2,2,2)),
output_rank=2).eval()
array([[[[  4.,   5.],
         [  6.,   7.]],

        [[ 12.,  17.],
         [ 22.,  27.]]],


       [[[ 20.,  29.],
         [ 38.,  47.]],

        [[ 28.,  41.],
         [ 54.,  67.]]]])
```

Parameters:

- **left** – left side matrix or tensor
- **right** – right side matrix or tensor
- **output_rank** (*int*) – in case we have tensors as arguments, output_rank represents the number of axes to be collapsed in order to transform the tensors into matrices, perform the operation and then reshape back (explode the axes)
- **infer_input_rank_to_map** (*int*) – meant for internal use only. Always use default value
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

`times_transpose`(*left, right, name=''*) [source]

The output of this operation is the product of the first (`left`) argument with the second (`right`) argument transposed. The second (`right`) argument must have a rank of 1 or 2. This operation is conceptually computing `np.dot(left, right.T)` except when `right` is a vector in which case the output is `np.dot(left,np.reshape(right,(1,-1)).T)` (matching numpy when `left` is a vector).

Example

```
>>> a=np.array([[1,2],[3,4]],dtype=np.float32)
>>> b=np.array([2,-1],dtype=np.float32)
>>> c=np.array([[2,-1]],dtype=np.float32)
>>> d=np.reshape(np.arange(24,dtype=np.float32),(4,3,2))
>>> print(C.times_transpose(a, a).eval())
[[  5.  11.]
 [ 11.  25.]]
>>> print(C.times_transpose(a, b).eval())
[[ 0.]
 [ 2.]]
>>> print(C.times_transpose(a, c).eval())
[[ 0.]
 [ 2.]]
>>> print(C.times_transpose(b, a).eval())
[ 0.  2.]
>>> print(C.times_transpose(b, b).eval())
[ 5.]
>>> print(C.times_transpose(b, c).eval())
[ 5.]
>>> print(C.times_transpose(c, a).eval())
[[ 0.  2.]]
>>> print(C.times_transpose(c, b).eval())
[[ 5.]]
>>> print(C.times_transpose(c, c).eval())
[[ 5.]]
>>> print(C.times_transpose(d, a).eval())
[[[   2.    4.]
  [   8.   18.]
  [  14.   32.]]

 [[  20.   46.]
  [  26.   60.]
  [  32.   74.]]

 [[  38.   88.]
  [  44.  102.]
  [  50.  116.]]

 [[  56.  130.]
  [  62.  144.]
  [  68.  158.]]]
>>> print(C.times_transpose(d, b).eval())
[[[ -1.]
  [  1.]
  [  3.]]

 [[  5.]
  [  7.]
  [  9.]]

 [[ 11.]
  [ 13.]
  [ 15.]]

 [[ 17.]
  [ 19.]
  [ 21.]]]
>>> print(C.times_transpose(d, c).eval())
[[[ -1.]
  [  1.]
  [  3.]]

 [[  5.]
  [  7.]
  [  9.]]

 [[ 11.]
  [ 13.]
  [ 15.]]

 [[ 17.]
  [ 19.]
  [ 21.]]]
```

Parameters:
- **left** – left side tensor
- **right** – right side matrix or vector
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

---

to_batch(*x, name=''*) [source]

Concatenate the input tensor's first axis to batch axis.

Example

```
>>> data = np.arange(12).reshape((3,2,2))
>>> x = C.constant(value=data)
>>> y = C.to_batch(x)
>>> y.shape
(2, 2)
```

Parameters:
- **x** – a tensor with dynamic axis
- **name** – (str, optional, keyword only): the name of the Function instance in the network

Returns: `Function`

---

to_sequence(*x, sequence_lengths=None, sequence_axis_name_prefix='toSequence_', name=''*) [source]

This function converts 'x' to a sequence using the most significant static axis [0] as the sequence axis.

The sequenceLengths input is optional; if unspecified, all sequences are assumed to be of the same length; i.e. dimensionality of the most significant static axis

Parameters:
- **x** – the tensor (or its name) which is converted to a sequence
- **sequence_lengths** – Optional tensor operand representing the sequence lengths. if unspecified, all sequences are assumed to be of the same length; i.e. dimensionality of the most significant static axis.
- **sequence_axis_name_prefix** (*str, optional*) – prefix of the new sequence axis name.
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

---

to_sequence_like(*x, dynamic_axes_like, name=''*) [source]

This function converts 'x' to a sequence using the most significant static axis [0] as the sequence axis. The length of the sequences are obtained from the 'dynamic_axes_like' operand.

Parameters:

- **x** – the tensor (or its name) which is converted to a sequence
- **dynamic_axes_like** – Tensor operand used to obtain the lengths of the generated sequences. The dynamic axes of the generated sequence tensor match the dynamic axes of the 'dynamic_axes_like' operand.
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

`transpose`(*x, perm, name=''*) [source]

Permutes the axes of the tensor. The output has the same data but the axes are permuted according to `perm`.

Example

```
>>> a = np.arange(24).reshape(2,3,4).astype('f')
>>> np.array_equal(C.transpose(a, perm=(2, 0, 1)).eval(), np.transpose(a, (2, 0, 1)))
True
```

Parameters:

- **x** – tensor to be transposed
- **perm** (*list*) – the permutation to apply to the axes.
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

`unpack_batch`(*x, name=''*) [source]

Concatenate the input tensor's last dynamic axis to static axis. Only tensors with batch axis are supported now.

Example

```
>>> data = np.arange(12).reshape((3,2,2))
>>> x = C.input((2,2))
>>> C.unpack_batch(x).eval({x:data})
array([[[  0.,   1.],
        [  2.,   3.]],

       [[  4.,   5.],
        [  6.,   7.]],

       [[  8.,   9.],
        [ 10.,  11.]]], dtype=float32)
```

Parameters:
- **x** – a tensor with dynamic axis
- **name** – (str, optional, keyword only): the name of the Function instance in the network

Returns: `Function`

---

unpooling(*operand, pooling_input, unpooling_type, unpooling_window_shape, strides=(1,), auto_padding=[False], name=''*) [source]

Unpools the `operand` using information from `pooling_input`. Unpooling mirrors the operations performed by pooling and depends on the values provided to the corresponding pooling operation. The output should have the same shape as pooling_input. Pooling the result of an unpooling operation should give back the original input.

Example

```
>>> img = np.reshape(np.arange(16, dtype = np.float32), [1, 4, 4])
>>> x = C.input_variable(img.shape)
>>> y = C.pooling(x, C.MAX_POOLING, (2,2), (2,2))
>>> C.unpooling(y, x, C.MAX_UNPOOLING, (2,2), (2,2)).eval({x : [img]})
array([[[[  0.,   0.,   0.,   0.],
         [  0.,   5.,   0.,   7.],
         [  0.,   0.,   0.,   0.],
         [  0.,  13.,   0.,  15.]]]], dtype=float32)
```

Parameters:
- **operand** – unpooling input
- **pooling_input** – input to the corresponding pooling operation
- **unpooling_type** – only `MAX_UNPOOLING` is supported now
- **unpooling_window_shape** – dimensions of the unpooling window
- **strides** (*default 1*) – strides.
- **auto_padding** – automatic padding flags for each input dimension.
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function`

Subpackages

- cntk.ops.sequence package

Submodules

- cntk.ops.functions module