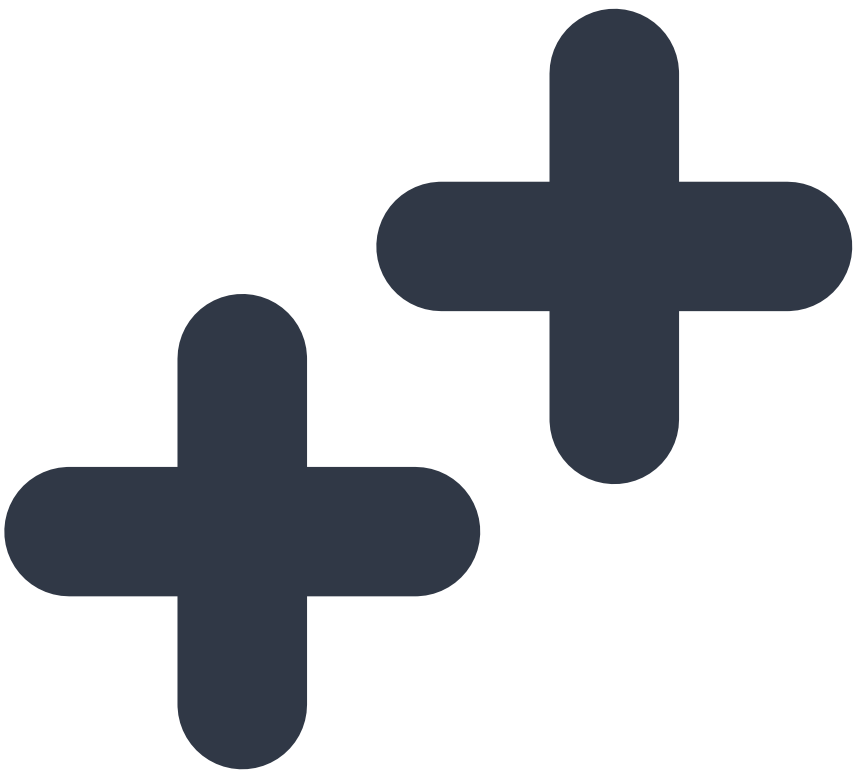


- Docs
- Tutorials
- API
- Blog
- GitHub
- File an Issue
- Contribute
-

- Docs
- Tutorials
- API
- Blog

- GitHub
- File an Issue
- Contribute
-

Docs

Quick Start

- Install
- Learn More
- Upgrading to Caffe2

Learn

- Applications of Deep Learning
- Operators Overview
- Integrating Caffe2 on iOS/Android
- Distributed Training
- Datasets
- Caffe2 Model Zoo

Tutorials

- Caffe2 Tutorials Overview
- Intro Tutorial
- Models and Datasets
- Basics of Caffe2 - Workspaces, Operators, and Nets
- Toy Regression
- Image Pre-Processing
- Loading Pre-Trained Models
- MNIST - Create a CNN from Scratch
- Create Your Own Dataset
- AI Camera Demo and Tutorial
- RNNs and LSTM Networks
- Synchronous SGD

Reference

- Operators Catalogue
- Custom Operators
- Sparse Operations
- CNN Class
- Workspace Class

API

- Caffe2 C++ and Python APIs
- Python
- C++

Operators Catalogue

Accumulate#

Accumulate operator accumulates the input tensor to the output tensor. If the output tensor already has the right size, we add to it; otherwise, we first initialize the output tensor to all zeros, and then do accumulation. Any further calls to the operator, given that no one else fiddles with the output in the interim, will do simple accumulations. Accumulation is done using Axpby operation as shown:

```
1  Y = 1*X + gamma*Y
```

where X is the input tensor, Y is the output tensor and gamma is the multiplier argument.

Interface#

Arguments	
gamma	(float, default 1.0) Accumulation multiplier
Inputs	
input	The input tensor that has to be accumulated to the output tensor. If the output size is not the same as input size, the output tensor is first reshaped and initialized to zero, and only then, accumulation is done.
Outputs	
output	Accumulated output tensor

Code#

caffe2/operators/accumulate_op.cc

Accuracy#

Accuracy takes two inputs- predictions and labels, and returns a float accuracy value for the batch. Predictions are expected in the form of 2-D tensor containing a batch of scores for various classes, and labels are expected in the form of 1-D tensor containing true label indices of samples in the batch. If the score for the label index in the predictions is the highest among all classes, it is considered a correct prediction.

Interface#

Inputs	
predictions	2-D tensor (Tensor) of size (num_batches x num_classes) containing scores
labels	1-D tensor (Tensor) of size (num_batches) having the indices of true labels
Outputs	
accuracy	1-D tensor (Tensor) of size 1 containing accuracy

Code#

caffe2/operators/accuracy_op.cc

Add#

Performs element-wise binary addition (with limited broadcast support). If necessary the right-hand-side argument will be broadcasted to match the shape of left-hand-side argument. When broadcasting is specified, the second tensor can either be of size 1 (a scalar value), or having its shape as a contiguous subset of the first tensor's shape. The starting of the mutually equal shape is specified by the argument "axis", and if it is not set, suffix matching is assumed. 1-dim expansion doesn't work yet. For example, the following tensor shapes are supported (with broadcast=1):

```
1  shape(A) = (2, 3, 4, 5), shape(B) = (,), i.e. B is a scalar
2  shape(A) = (2, 3, 4, 5), shape(B) = (5,)
3  shape(A) = (2, 3, 4, 5), shape(B) = (4, 5)
4  shape(A) = (2, 3, 4, 5), shape(B) = (3, 4), with axis=1
5  shape(A) = (2, 3, 4, 5), shape(B) = (2), with axis=0
6
```

Argument broadcast=1 needs to be passed to enable broadcasting.

Interface#

Arguments	
broadcast	Pass 1 to enable broadcasting

axis	If set, defines the broadcast dimensions. See doc for details.
Inputs	
A	First operand, should share the type with the second operand.
B	Second operand. With broadcasting can be of smaller size than A. If broadcasting is disabled it should be of the same size.
Outputs	
C	Result, has same dimensions and type as A

Code#

caffe2/operators/elementwise_op_schema.cc

AddPadding#

Given a partitioned tensor T<N, D1..., Dn>, where the partitions are defined as ranges on its outer-most (slowest varying) dimension N, with given range lengths, return a tensor T<N + 2*padding_width, D1 ..., Dn> with paddings added to the start and end of each range. Optionally, different paddings can be provided for beginning and end. Paddings provided must be a tensor T<D1..., Dn>. If no padding is provided, add zero padding. If no lengths vector is provided, add padding only once, at the start and end of data.

Interface#

Arguments	
padding_width	Number of copies of padding to add around each range.
end_padding_width	(Optional) Specifies a different end-padding width.
Inputs	
data_in	(T<N, D1..., Dn>) Input data
lengths	(i64) Num of elements in each range. sum(lengths) = N.
start_padding	T<D1..., Dn> Padding data for range start.
end_padding	T<D1..., Dn> (optional) Padding for range end. If not provided, start_padding is used as end_padding as well.
Outputs	
data_out	(T<N + 2*padding_width, D1..., Dn>) Padded data.
lengths_out	(i64, optional) Lengths for each padded range.

Code#

caffe2/operators/sequence_ops.cc

Alias#

Makes the output and the input share the same underlying storage. WARNING: in general, in caffe2's operator interface different tensors should have different underlying storage, which is the assumption made by components such as the dependency engine and memory optimization. Thus, in normal situations you should not use the AliasOp, especially in a normal forward-backward pass. The Alias op is provided so one can achieve true asynchrony, such as Hogwild, in a graph. But make sure you understand all the implications similar to multi-thread computation before you use it explicitly.

Interface#

Inputs	
input	Input tensor whose storage will be shared.
Outputs	
output	Tensor of same shape as input, sharing its storage.

Code#

caffe2/operators/utility_ops.cc

Allgather#

Does an allgather operation among the nodes.

Interface#

Inputs

comm_world The common world.

X A tensor to be allgathered.

Outputs

Y The allgathered tensor, same on all nodes.

Code#

caffe2/operators/communicator_op.cc

Allreduce#

Does an allreduce operation among the nodes. Currently only Sum is supported.

Interface#

Inputs

comm_world The common world.

X A tensor to be allreduced.

Outputs

Y The allreduced tensor, same on all nodes.

Code#

caffe2/operators/communicator_op.cc

And#

Performs element-wise logical operation and (with limited broadcast support). Both input operands should be of type bool . If necessary the right-hand-side argument will be broadcasted to match the shape of left-hand-side argument. When broadcasting is specified, the second tensor can either be of size 1 (a scalar value), or having its shape as a contiguous subset of the first tensor's shape. The starting of the mutually equal shape is specified by the argument "axis", and if it is not set, suffix matching is assumed. 1-dim expansion doesn't work yet. For example, the following tensor shapes are supported (with broadcast=1):

```
1  shape(A) = (2, 3, 4, 5), shape(B) = (,), i.e. B is a scalar
2  shape(A) = (2, 3, 4, 5), shape(B) = (5,)
3  shape(A) = (2, 3, 4, 5), shape(B) = (4, 5)
4  shape(A) = (2, 3, 4, 5), shape(B) = (3, 4), with axis=1
5  shape(A) = (2, 3, 4, 5), shape(B) = (2), with axis=0
6
```

Argument broadcast=1 needs to be passed to enable broadcasting.

Interface#

Arguments

broadcast Pass 1 to enable broadcasting

axis If set, defines the broadcast dimensions. See doc for details.

Inputs

A First operand.

B Second operand. With broadcasting can be of smaller size than A. If broadcasting is disabled it should be of the same size.

Outputs

C Result, has same dimensions and A and type bool

Code#

caffe2/operators/elementwise_op_schema.cc

Append#

Append input 2 to the end of input 1. Input 1 must be the same as output, that is, it is required to be in-place. Input 1 may have to be re-allocated in order for accommodate to the new size. Currently, an exponential growth ratio is used in order to ensure amortized constant time complexity. All except the outer-most dimension must be the same between input 1 and 2.

Interface#

Inputs	
dataset	The tensor to be appended to.
new_data	Tensor to append to the end of dataset.
Outputs	
dataset	Same as input 0, representing the mutated tensor.

Code#

caffe2/operators/dataset_ops.cc

AtomicAppend#

No documentation yet.

Code#

caffe2/operators/dataset_ops.cc

AtomicFetchAdd#

Given a mutex and two int32 scalar tensors, performs an atomic fetch add by mutating the first argument and adding it to the second input argument. Returns the updated integer and the value prior to the update.

Interface#

Inputs	
mutex_ptr	Blob containing to a unique_ptr
mut_value	Value to be mutated after the sum.
increment	Value to add to the first operand.
Outputs	
mut_value	Mutated value after sum. Usually same as input 1.
fetched_value	Value of the first operand before sum.

Code#

caffe2/operators/atomic_ops.cc

AveragePool#

AveragePool consumes an input blob X and applies average pooling across the the blob according to kernel sizes, stride sizes, and pad lengths defined by the ConvPoolOpBase operator. Average pooling consisting of averaging all values of a subset of the input tensor according to the kernel size and downsampling the data into the output blob Y for further processing.

Interface#

Inputs

X	Input data tensor from the previous operator; dimensions depend on whether the NCHW or NHWC operators are being used. For example, in the former, the input 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 the width of the data. The corresponding permutation of dimensions is used in the latter case.

Outputs

Y	Output data tensor from average pooling across the input tensor. Dimensions will vary based on various kernel, stride, and pad sizes.

Code#

caffe2/operators/pool_op.cc

AveragePoolGradient#

No documentation yet.

Code#

caffe2/operators/pool_op.cc

AveragedLoss#

AveragedLoss takes in a 1-D tensor as input and returns a single output float value which represents the average of input data (average of the losses).

Interface#

Inputs

input	The input data as Tensor

Outputs

output	The output tensor of size 1 containing the averaged value.

Code#

caffe2/operators/loss_op.cc

AveragedLossGradient#

No documentation yet.

Code#

caffe2/operators/loss_op.cc

BatchMatMul#

Batch Matrix multiplication Yi = Ai * Bi, where A has size (C x M x K), B has size (C x K x N) where C is the batch size and i ranges from 0 to C-1.

Interface#

Arguments

trans_a	Pass 1 to transpose A before multiplication
trans_b	Pass 1 to transpose B before multiplication

Inputs

A	3D matrix of size (C x M x K)
B	3D matrix of size (C x K x N)

Outputs

Y	3D matrix of size (C x M x N)

Code#

caffe2/operators/batch_matmul_op.cc

BatchToSpace#

BatchToSpace for 4-D tensors of type T. Rearranges (permutes) data from batch into blocks of spatial data, followed by cropping. This is the reverse transformation of SpaceToBatch. More specifically, this op outputs a copy of the input tensor where values from the batch dimension are moved in spatial blocks to the height and width dimensions, followed by cropping along the height and width dimensions.

Code#

caffe2/operators/space_batch_op.cc

BooleanMask#

Given a data 1D tensor and a mask (boolean) tensor of same shape, returns a tensor containing only the elements corresponding to positions where the mask is true.

Interface#

Inputs	
data	The 1D, original data tensor.
mask	A tensor of bools of same shape as data.
Outputs	
masked_data	A tensor of same type as data.

Code#

caffe2/operators/boolean_mask_ops.cc

BooleanMaskLengths#

Given a tensor of int32 segment lengths and a mask (boolean) tensor, return the segment lengths of a corresponding segmented tensor after BooleanMask is applied.

Interface#

Inputs	
lengths	A 1D int32 tensor representing segment lengths.
mask	A 1D bool tensor of values to keep.
Outputs	
masked_lengths	Segment lengths of a masked tensor.

Code#

caffe2/operators/boolean_mask_ops.cc

Broadcast#

Does a broadcast operation from the root node to every other node. The tensor on each node should have been pre-created with the same shape and data type.

Interface#

Arguments

root	(int, default 0) the root to run broadcast from.
Inputs	
comm_world	The common world.
X	A tensor to be broadcasted.
Outputs	
X	In-place as input 1.

Code#

caffe2/operators/communicator_op.cc

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. If the ‘to’ argument is not provided or is not one of the enumerated types in DataType, Caffe2 throws an Enforce error. NOTE: Casting to and from strings is not supported yet.

Interface#

Arguments	
to	The data type to which the elements of the input tensor are cast.Strictly must be one of the types from DataType enum in TensorProto
Inputs	
input	Input tensor to be cast.
Outputs	
output	Output tensor with the same shape as input with type specified by the ‘to’ argument

Code#

caffe2/operators/cast_op.cc

CheckAtomicBool#

Copy the value of a atomic to a bool

Interface#

Inputs	
atomic_bool	Blob containing a unique_ptr<atomic>
Outputs	
value	Copy of the value for the atomic

Code#

caffe2/operators/atomic_ops.cc

CheckCounterDone#

If the internal count value <= 0, outputs true, otherwise outputs false,

Interface#

Inputs	
counter	A blob pointing to an instance of a counter.
Outputs	
done	true if the internal count is zero or negative.

Code#

caffe2/operators/counter_ops.cc

CheckDatasetConsistency#

Checks that the given data fields represents a consistent dataset unther the schema specified by the fields argument. Operator fails if the fields are not consistent. If data is consistent, each field's data can be safely appended to an existing dataset, keeping it consistent.

Interface#

Arguments	
fields	List of strings representing the string names in the formatspecified in the doc for CreateTreeCursor.
Inputs	
field_0	Data for field 0.

Code#

caffe2/operators/dataset_ops.cc

Checkpoint#

The Checkpoint operator is similar to the Save operator, but allows one to save to db every few iterations, with a db name that is appended with the iteration count. It takes [1, infinity) number of inputs and has no output. The first input has to be a TensorCPU of type int and has size 1 (i.e. the iteration counter). This is determined whether we need to do checkpointing.

Interface#

Arguments	
absolute_path	(int, default 0) if set, use the db path directly and do not prepend the current root folder of the workspace.
db	(string) a template string that one can combine with the iteration to create the final db name. For example, "/home /lonestarr/checkpoint_%08d.db"
db_type	(string) the type of the db.
every	(int, default 1) the checkpointing is carried out when (iter mod every) is zero.

Code#

caffe2/operators/load_save_op.cc

Clip#

Clip operator limits the given input within an interval. The interval is specified with arguments 'min' and 'max'. They default to numeric_limits::min() and numeric_limits::max() respectively. The clipping operation can be done in in-place fashion too, where the input and output blobs are the same.

Interface#

Arguments	
min	Minimum value, under which element is replaced by min
max	Maximum value, above which element is replaced by max
Inputs	
input	Input tensor (Tensor) containing elements to beclipped
output	Output tensor (Tensor) containing clippedinput elements

Code#

caffe2/operators/clip_op.cc

ClipGradient#

No documentation yet.

Code#

caffe2/operators/clip_op.cc

Col2Im#

No documentation yet.

Code#

caffe2/operators/im2col_op.cc

CollectTensor#

Collect tensor into tensor vector by reservoir sampling, argument num_to_collect indicates the max number of tensors that will be collcted. The first half of the inputs are tensor vectors, which are also the outputs. The second half of the inputs are the tensors to be collected into each vector (in the same order). The input tensors are collected in all-or-none manner. If they are collected, they will be placed at the same index in the output vectors.

Interface#

Arguments

num_to_collect The max number of tensors to collect

Code#

caffe2/operators/dataset_ops.cc

ComputeOffset#

Compute the offsets matrix given cursor and data blobs. Need to be ran at beginning or after reseting cursor Input(0) is a blob pointing to a TreeCursor, and [Input(1),... Input(num_fields)] a list of tensors containing the data for each field of the dataset. ComputeOffset is thread safe.

Interface#

Inputs

cursor	A blob containing a pointer to the cursor.
dataset_field_0	First dataset field

Outputs

field_0	Tensor containing offset info for this chunk.

Code#

caffe2/operators/dataset_ops.cc

Concat#

Concatenate a list of tensors into a single tensor.

Interface#

Arguments

axis	Which axis to concat on
order	Either NHWC or HCWH, will concat on C axis

Code#

caffe2/operators/concat_split_op.cc

ConcatTensorVector#

Concat Tensors in the std::unique_ptr<std::vector > along the first dimension.

Interface#

Inputs	
vector of Tensor	std::unique_ptr<std::vector >
Outputs	
tensor	tensor after concatenating

Code#

caffe2/operators/dataset_ops.cc

ConditionalSetAtomicBool#

1 Set an atomic<bool> to true if the given condition bool variable is true

Interface#

Inputs	
atomic_bool	Blob containing a unique_ptr<atomic>
condition	Blob containing a bool

Code#

caffe2/operators/atomic_ops.cc

ConstantFill#

The operator fills the elements of the output tensor with a constant value specified by the ‘value’ argument. 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. If the ‘dtype’ argument is not provided, the data type of ‘value’ is used. The output tensor shape is specified by the ‘shape’ argument. If the number of input is 1, the shape will be identical to that of the input at run time with optional additional dimensions appended at the end as specified by ‘extra_shape’ argument. In that case the ‘shape’ argument should not be set. If input_as_shape is set to true, then the input should be a 1D tensor containing the desired output shape (the dimensions specified in extra_shape will also be appended) NOTE: Currently, it supports data type of float, int32, int64, and bool.

Interface#

Arguments	
value	The value for the elements of the output tensor.
dtype	The data type for the elements of the output tensor.Strictly must be one of the types from DataType enum in TensorProto.
shape	The shape of the output tensor.Cannot set the shape argument and pass in an input at the same time.
extra_shape	The additional dimensions appended at the end of the shape indicatedby the input blob.Cannot set the extra_shape argument when there is no input blob.
input_as_shape	1D tensor containing the desired output shape
Inputs	
input	Input tensor (optional) to provide shape information.
Outputs	
output	Output tensor of constant values specified by ‘value’argument and its type is specified by the ‘dtype’ argument

Code#

caffe2/operators/filler_op.cc

Conv#

The convolution operator consumes an input vector, the filter blob and the bias blob and computes the output. Note that other parameters, such as the stride and kernel size, or the pads' sizes in each direction are not necessary for input because they are provided by the ConvPoolOpBase operator. Various dimension checks are done implicitly, and the sizes are specified in the Input docs for this operator. As is expected, the filter is convolved with a subset of the image and the bias is added; this is done throughout the image data and the output is computed. As a side note on the implementation layout: conv_op_impl.h is the templated implementation of the conv_op.h file, which is why they are separate files.

Interface#

Inputs

X	Input data blob 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 NCHW usage. On the other hand, the NHWC Op has a different set of dimension constraints.
filter	The filter blob that will be used in the convolutions; has size (M x C x kH x kW), where C is the number of channels, and kH and kW are the height and width of the kernel.
bias	The 1D bias blob that is added through the convolution; has size (M).

Outputs

Y	Output data blob that contains the result of the convolution. The output dimensions are functions of the kernel size, stride size, and pad lengths.

Code#

caffe2/operators/conv_op.cc

ConvGradient#

No documentation yet.

Code#

caffe2/operators/conv_op.cc

ConvTranspose#

```
1  The transposed convolution consumes an input vector, the filter blob, and
2  the bias blob, and computes the output. Note that other parameters, such as
3  the stride and kernel size, or the pads' sizes in each direction are not
4  necessary for input because they are provided by the
5  ConvTransposeUnpoolOpBase operator. Various dimension checks are done
6  implicitly, and the sizes are specified in the Input docs for this operator.
7  As is expected, the filter is deconvolved with a subset of the
8  image and the bias is added; this is done throughout the image data and the
9  output is computed. As a side note on the implementation layout:
10 conv_transpose_op_impl.h is the templated implementation of the
11 conv_transpose_op.h file, which is why they are separate files.
```

Interface#

Inputs

X	Input data blob 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 NCHW usage. On the other hand, the NHWC Op has a different set of dimension constraints.

filter	The filter blob that will be used in the transposed convolution; has size (M x C x kH x kW), where C is the number of channels, and kH and kW are the height and width of the kernel.
bias	The 1D bias blob that is added through the convolution;has size (C)
Outputs	
Y	Output data blob that contains the result of the transposed convolution. The output dimensions are functions of the kernel size, stride size, and pad lengths.

Code#

caffe2/operators/conv_transpose_op.cc

ConvTransposeGradient#

No documentation yet.

Code#

caffe2/operators/conv_transpose_op.cc

Copy#

Copy input tensor into output, potentially across devices.

Interface#

Inputs	
input	The input tensor.
Outputs	
output	Tensor that will contain a copy of the input.

Code#

caffe2/operators/utility_ops.cc

CopyCPUToGPU#

Copy tensor for CPU to GPU context. Must be run under GPU device option.

Interface#

Inputs	
input	The input tensor.
Outputs	
output	Tensor that will contain a copy of the input.

Code#

caffe2/operators/utility_ops.cc

CopyFromCPUInput#

Take a CPU input tensor and copy it to an output in the current Context (GPU or CPU). This may involves cross-device MemCpy.

Interface#

Inputs	
input	The input CPU tensor.
Outputs	

output	either a TensorCUDA or a TensorCPU

Code#

caffe2/operators/utility_ops.cc

CopyGPUToCPU#

Copy tensor for GPU to CPU context. Must be run under GPU device option.

Interface#

Inputs	
input	The input tensor.
Outputs	
output	Tensor that will contain a copy of the input.

Code#

caffe2/operators/utility_ops.cc

CosineEmbeddingCriterion#

CosineEmbeddingCriterion takes two inputs: the similarity value and the label, and computes the elementwise criterion output as output = 1 - s,

1 if y == 1

1 max(0, s - margin), if y == -1

Interface#

Inputs	
S	The cosine similarity as a 1-dim TensorCPU.
Y	The label as a 1-dim TensorCPU with int value of 1 or -1.
Outputs	
loss	The output loss with the same dimensionality as S.

Code#

caffe2/operators/cosine_embedding_criterion_op.cc

CosineEmbeddingCriterionGradient#

No documentation yet.

Code#

caffe2/operators/cosine_embedding_criterion_op.cc

CosineSimilarity#

1 Given two input float tensors X, Y, and produces one output float tensor
2 of the cosine similarity between X and Y.

Interface#

Inputs

X 1D input tensor

Outputs

Y 1D input tensor

Code#

caffe2/operators/distance_op.cc

CosineSimilarityGradient#

No documentation yet.

Code#

caffe2/operators/distance_op.cc

CountDown#

If the internal count value > 0, decreases count value by 1 and outputs false, otherwise outputs true.

Interface#

Inputs

counter A blob pointing to an instance of a counter.

Outputs

done false unless the internal count is zero.

Code#

caffe2/operators/counter_ops.cc

CountUp#

Increases count value by 1 and outputs the previous value atomically

Interface#

Inputs

counter A blob pointing to an instance of a counter.

Outputs

previous_count count value BEFORE this operation

Code#

caffe2/operators/counter_ops.cc

CreateAtomicBool#

Create an unique_ptr blob to hold a atomic

Interface#

Outputs

atomic_bool Blob containing a unique_ptr<atomic>

Code#

caffe2/operators/atomic_ops.cc

CreateCommonWorld#

Creates a common world for communication operators.

Interface#

Arguments	
size	(int) size of the common world.
rank	(int) rank of this node in the common world.
Inputs	
kv_handler	Key/value handler for rendezvous (optional).
Outputs	
comm_world	A common world for collective operations.

Code#

caffe2/operators/communicator_op.cc

CreateCounter#

Creates a count-down counter with initial value specified by the 'init_count' argument.

Interface#

Arguments	
init_count	Initial count for the counter, must be >= 0.
Outputs	
counter	A blob pointing to an instance of a new counter.

Code#

caffe2/operators/counter_ops.cc

CreateMutex#

Creates an unlocked mutex and returns it in a unique_ptr blob.

Interface#

Outputs

mutex_ptr Blob containing a std::unique_ptr.

Code#

caffe2/operators/atomic_ops.cc

CreateQPSMetric#

CreateQPSMetric operator create a blob that will store state that is required for computing QPSMetric. The only output of the operator will have blob with QPSMetricState as an output.

Interface#

Outputs	
output	Blob with QPSMetricState

Code#

caffe2/operators/metrics_ops.cc

CreateTensorVector#

Create a std::unique_ptr<std::vector >

Code#

caffe2/operators/dataset_ops.cc

CreateTextFileReader#

Create a text file reader. Fields are delimited by .

Interface#

Arguments	
filename	Path to the file.
num_pases	Number of passes over the file.
field_types	List with type of each field. Type enum is found at core.DataType.
Outputs	
handler	Pointer to the created TextFileReaderInstance.

Code#

caffe2/operators/text_file_reader.cc

CreateTreeCursor#

Creates a cursor to iterate through a list of tensors, where some of those tensors contains the lengths in a nested schema. The schema is determined by the fields arguments. For example, to represent the following schema:

```
Struct(
    a=Int(),
    b=List(List(Int),
    c=List(
        Struct(

            c1=String,
            c2=List(Int),
        ),
    ),
)
```

the field list will be:

```
[
    "a",
    "b:lengths",
    "b:values:lengths",
    "b:values:values",
    "c:lengths",
    "c:c1",
    "c:c2:lengths",
    "c:c2:values",
]
```

```
11
```

And for the following instance of the struct:

```
1  Struct(
2      a=3,
3      b=[[4, 5], [6, 7, 8], [], [9]],
4      c=[
5          Struct(c1='alex', c2=[10, 11]),
6          Struct(c1='bob', c2=[12]),
7      ],
8  )
9
```

The values of the fields will be:

```
 1  {
 2      "a": [3],
 3      "b:lengths": [4],
 4      "b:values:lengths": [2, 3, 0, 1],
 5      "b:values:values": [4, 5, 6, 7, 8, 9],
 6      "c:lengths": [2],
 7      "c:c1": ["alex", "bob"],
 8      "c:c2:lengths": [2, 1],
 9      "c:c2:values", [10, 11, 12],
10  }
11
```

In general, every field name in the format “{prefix}:lengths” defines a domain “{prefix}”, and every subsequent field in the format “{prefx}:{field}” will be in that domain, and the length of the domain is provided for each entry of the parent domain. In the example, “b:lengths” defines a domain of length 4, so every field under domain “b” will have 4 entries. The “lengths” field for a given domain must appear before any reference to that domain. Returns a pointer to an instance of the Cursor, which keeps the current offset on each of the domains defined by fields . Cursor also ensures thread-safety such that ReadNextBatch and ResetCursor can be used safely in parallel. A cursor does not contain data per se, so calls to ReadNextBatch actually need to pass a list of blobs containing the data to read for each one of the fields.

Interface#

Arguments

fields	A list of strings each one representing a field of the dataset.

Outputs

cursor	A blob pointing to an instance of a new TreeCursor.

Code#

caffe2/operators/dataset_ops.cc

CrossEntropy#

Operator computes the cross entropy between the input and the label set. In practice, it is most commonly used at the end of models, after the SoftMax operator and before the AveragedLoss operator. Note that CrossEntropy assumes that the soft labels provided is a 2D array of size N x D (batch size x number of classes). Each entry in the 2D label corresponds to the soft label for the input, where each element represents the correct probability of the class being selected. As such, each element must be between 0 and 1, and all elements in an entry must sum to 1. The formula used is:

```
1          Y[i] = sum_j (label[i][j] * log(X[i][j]))
2
```

where (i, j) is the classifier’s prediction of the jth class (the correct one), and i is the batch size. Each log has a lower limit for numerical

stability.

Interface#

Inputs	
X	Input blob from the previous layer, which is almost always the result of a softmax operation; X is a 2D array of size N x D, where N is the batch size and D is the number of classes
label	Blob containing the labels used to compare the input
Outputs	
Y	Output blob after the cross entropy computation

Code#

caffe2/operators/cross_entropy_op.cc

CrossEntropyGradient#

No documentation yet.

Code#

caffe2/operators/cross_entropy_op.cc

DepthConcat#

Backward compatible operator name for Concat.

Code#

caffe2/operators/concat_split_op.cc

DepthSplit#

Backward compatible operator name for Split.

Code#

caffe2/operators/concat_split_op.cc

Div#

Performs element-wise binary division (with limited broadcast support). If necessary the right-hand-side argument will be broadcasted to match the shape of left-hand-side argument. When broadcasting is specified, the second tensor can either be of size 1 (a scalar value), or having its shape as a contiguous subset of the first tensor's shape. The starting of the mutually equal shape is specified by the argument "axis", and if it is not set, suffix matching is assumed. 1-dim expansion doesn't work yet. For example, the following tensor shapes are supported (with broadcast=1):

```
1  shape(A) = (2, 3, 4, 5), shape(B) = (,), i.e. B is a scalar
2  shape(A) = (2, 3, 4, 5), shape(B) = (5,)
3  shape(A) = (2, 3, 4, 5), shape(B) = (4, 5)
4  shape(A) = (2, 3, 4, 5), shape(B) = (3, 4), with axis=1
5  shape(A) = (2, 3, 4, 5), shape(B) = (2), with axis=0
6
```

Argument broadcast=1 needs to be passed to enable broadcasting.

Interface#

Arguments

broadcast	Pass 1 to enable broadcasting
axis	If set, defines the broadcast dimensions. See doc for details.
Inputs	
A	First operand, should share the type with the second operand.
B	Second operand. With broadcasting can be of smaller size than A. If broadcasting is disabled it should be of the same size.
Outputs	
C	Result, has same dimensions and type as A

Code#

caffe2/operators/elementwise_op_schema.cc

DivGradient#

No documentation yet.

Code#

caffe2/operators/elementwise_op_schema.cc

DotProduct#

1 Given two input float tensors X, Y, and produces one output float tensor
2 of the dot product between X and Y.

Interface#

Inputs	
X	1D input tensor
Outputs	
Y	1D input tensor

Code#

caffe2/operators/distance_op.cc

DotProductGradient#

No documentation yet.

Code#

caffe2/operators/distance_op.cc

Dropout#

Dropout takes one input data (Tensor) and produces two Tensor outputs, output (Tensor) and mask (Tensor). Depending on whether it is in test mode or not, the output Y will either be a random dropout, or a simple copy of the input. Note that our implementation of Dropout does scaling in the training phase, so during testing nothing needs to be done.

Interface#

Arguments	
ratio	(float, default 0.5) the ratio of random dropout
is_test	(int, default 0) if nonzero, run dropout in test mode where the output is simply Y = X.
Inputs	

data	The input data as Tensor.
Outputs	
output	The output.
mask	The output mask. If is_test is nonzero, this output is not filled.

Code#

caffe2/operators/dropout_op.cc

DropoutGrad#

No documentation yet.

Code#

caffe2/operators/dropout_op.cc

EQ#

Performs element-wise comparison == (with limited broadcast support). If necessary the right-hand-side argument will be broadcasted to match the shape of left-hand-side argument. When broadcasting is specified, the second tensor can either be of size 1 (a scalar value), or having its shape as a contiguous subset of the first tensor's shape. The starting of the mutually equal shape is specified by the argument "axis", and if it is not set, suffix matching is assumed. 1-dim expansion doesn't work yet. For example, the following tensor shapes are supported (with broadcast=1):

```
1  shape(A) = (2, 3, 4, 5), shape(B) = (,), i.e. B is a scalar
2  shape(A) = (2, 3, 4, 5), shape(B) = (5,)
3  shape(A) = (2, 3, 4, 5), shape(B) = (4, 5)
4  shape(A) = (2, 3, 4, 5), shape(B) = (3, 4), with axis=1
5  shape(A) = (2, 3, 4, 5), shape(B) = (2), with axis=0
6
```

Argument broadcast=1 needs to be passed to enable broadcasting.

Interface#

Arguments	
broadcast	Pass 1 to enable broadcasting
axis	If set, defines the broadcast dimensions. See doc for details.
Inputs	
A	First operand, should share the type with the second operand.
B	Second operand. With broadcasting can be of smaller size than A. If broadcasting is disabled it should be of the same size.
Outputs	
C	Result, has same dimensions and A and type bool

Code#

caffe2/operators/elementwise_op_schema.cc

Elu#

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

Interface#

Inputs	
X	1D input tensor

Outputs

Y 1D input tensor

Code#

caffe2/operators/elu_op.cc

EluGradient#

EluGradient takes both Y and dY and uses this to update dX according to the chain rule and derivatives of the rectified linear function.

Code#

caffe2/operators/elu_op.cc

EnsureCPUOutput#

Take an input tensor in the current Context (GPU or CPU) and create an output which is always a TensorCPU. This may involves cross-device MemCpy.

Interface#

Inputs

input The input CUDA or CPU tensor.

Outputs

output TensorCPU that is a copy of the input.

Code#

caffe2/operators/utility_ops.cc

Exp#

Calculates the exponential of the given input tensor, element-wise. This operation can be done in an in-place fashion too, by providing the same input and output blobs.

Interface#

Inputs

input Input tensor

Outputs

output The exponential of the input tensor computed element-wise

Code#

caffe2/operators/exp_op.cc

ExpandDims#

Insert single-dimensional entries to the shape of a tensor. Takes one required argument dims , a list of dimensions that will be inserted. Dimension indices in dims are as seen in the output tensor. For example:

```
1 Given a tensor such that tensor.Shape() = [3, 4, 5], then
2 ExpandDims(tensor, dims=[0, 4]).Shape() == [1, 3, 4, 5, 1])
3
```

If the same blob is provided in input and output, the operation is copy-free.

Interface#

Inputs	
data	Original tensor
Outputs	
expanded	Reshaped tensor with same data as input.

Code#

caffe2/operators/utility_ops.cc

ExtendTensor#

Extend input 0 if necessary based on max element in input 1. Input 0 must be the same as output, that is, it is required to be in-place. Input 0 may have to be re-allocated in order for accommodate to the new size. Currently, an exponential growth ratio is used in order to ensure amortized constant time complexity. All except the outer-most dimension must be the same between input 0 and 1.

Interface#

Inputs	
tensor	The tensor to be extended.
new_indices	The size of tensor will be extended based on max element in new_indices.
Outputs	
extended_tensor	Same as input 0, representing the mutated tensor.

Code#

caffe2/operators/extend_tensor_op.cc

FC#

Computes the result of passing an input vector X into a fully connected layer with 2D weight matrix W and 1D bias vector b. The layer computes Y = X * W^T + b, where X has size (M x K), W has size (N x K), b has size (N), and Y has size (M x N), where M is the batch size. Even though b is 1D, it is resized to size (M x N) implicitly and added to each vector in the batch. These dimensions must be matched correctly, or else the operator will throw errors.

Interface#

Arguments	
axis	(int32_t) default to 1; describes the axis of the inputs; defaults to one because the 0th axis most likely describes the batch_size
Inputs	
X	2D input of size (MxK) data
W	2D blob of size (KxN) containing fully connected weight matrix
b	1D blob containing bias vector
Outputs	
Y	2D output tensor

Code#

caffe2/operators/fully_connected_op.cc

FCGradient#

No documentation yet.

Code#

caffe2/operators/fully_connected_op.cc

FeedBlob#

FeedBlobs the content of the blobs. The input and output blobs should be one-to-one inplace.

Interface#

Arguments

value (string) if provided then we will use this string as the value for theprovided output tensor

Code#

caffe2/operators/feed_blob_op.cc

FindDuplicateElements#

Shrink the data tensor by removing data blocks with given zero-based indices in the outermost dimension of the tensor. Indices are not assumed in any order or unique but with the range [0, blocks_size). Indices could be empty.

Interface#

Inputs

data a 1-D tensor.

Outputs

indices indices of duplicate elements in data, excluding first occurrences.

Code#

caffe2/operators/find_duplicate_elements_op.cc

Flatten#

Flattens the input tensor into a 2D matrix, keeping the first dimension unchanged.

Interface#

Inputs

input A tensor of rank >= 2.

Outputs

output A tensor of rank 2 with the contents of the input tensor, with first dimension equal first dimension of input, and remaining input dimensions flatenned into the inner dimension of the output.

Code#

caffe2/operators/utility_ops.cc

FlattenToVec#

Flattens the input tensor into a 1D vector.

Interface#

Inputs

input A tensor of rank >= 1.

Outputs

output A tensor of rank 1 with the contents of the input tensor

Code#

caffe2/operators/utility_ops.cc

FloatToHalf#

No documentation yet.

Code#

caffe2/operators/half_float_ops.cc

Free#

Frees the content of the blobs. The input and output blobs should be one-to-one inplace.

Code#

caffe2/operators/free_op.cc

GE#

Performs element-wise comparison >= (with limited broadcast support). If necessary the right-hand-side argument will be broadcasted to match the shape of left-hand-side argument. When broadcasting is specified, the second tensor can either be of size 1 (a scalar value), or having its shape as a contiguous subset of the first tensor's shape. The starting of the mutually equal shape is specified by the argument "axis", and if it is not set, suffix matching is assumed. 1-dim expansion doesn't work yet. For example, the following tensor shapes are supported (with broadcast=1):

```
1  shape(A) = (2, 3, 4, 5), shape(B) = (,), i.e. B is a scalar
2  shape(A) = (2, 3, 4, 5), shape(B) = (5,)
3  shape(A) = (2, 3, 4, 5), shape(B) = (4, 5)
4  shape(A) = (2, 3, 4, 5), shape(B) = (3, 4), with axis=1
5  shape(A) = (2, 3, 4, 5), shape(B) = (2), with axis=0
6
```

Argument broadcast=1 needs to be passed to enable broadcasting.

Interface#

Arguments	
broadcast	Pass 1 to enable broadcasting
axis	If set, defines the broadcast dimensions. See doc for details.
Inputs	
A	First operand, should share the type with the second operand.
B	Second operand. With broadcasting can be of smaller size than A. If broadcasting is disabled it should be of the same size.
Outputs	
C	Result, has same dimensions and A and type bool

Code#

caffe2/operators/elementwise_op_schema.cc

GT#

Performs element-wise comparison > (with limited broadcast support). If necessary the right-hand-side argument will be broadcasted to match the shape of left-hand-side argument. When broadcasting is specified, the second tensor can either be of size 1 (a scalar value), or having its shape as a contiguous subset of the first tensor's shape. The starting of the mutually equal shape is specified by the argument "axis", and if it is not set, suffix matching is assumed. 1-dim expansion doesn't work yet. For example, the following tensor shapes are supported (with broadcast=1):

```
shape(A) = (2, 3, 4, 5), shape(B) = (,), i.e. B is a scalar
shape(A) = (2, 3, 4, 5), shape(B) = (5,)
shape(A) = (2, 3, 4, 5), shape(B) = (4, 5)
shape(A) = (2, 3, 4, 5), shape(B) = (3, 4), with axis=1
shape(A) = (2, 3, 4, 5), shape(B) = (2), with axis=0

```

Argument broadcast=1 needs to be passed to enable broadcasting.

Interface#

Arguments	
broadcast	Pass 1 to enable broadcasting
axis	If set, defines the broadcast dimensions. See doc for details.
Inputs	
A	First operand, should share the type with the second operand.
B	Second operand. With broadcasting can be of smaller size than A. If broadcasting is disabled it should be of the same size.
Outputs	
C	Result, has same dimensions and A and type bool

Code#

caffe2/operators/elementwise_op_schema.cc

Gather#

Given DATA tensor of rank r >= 1, and INDICES tensor of rank q, gather entries of the outer-most dimension of DATA indexed by INDICES, and concatenate them in an output tensor of rank q + (r - 1). Example:

```
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],
    ],
]
```

Interface#

Inputs	
DATA	Tensor of rank r >= 1.
INDICES	Tensor of int32/int64 indices, of any rank q.
Outputs	
OUTPUT	Tensor of rank q + (r - 1).

Code#

caffe2/operators/utility_ops.cc

GatherPadding#

Gather the sum of start and end paddings in a padded input sequence. Used in order to compute the gradients of AddPadding w.r.t the padding tensors.

Interface#

Arguments	
padding_width	Outer-size of padding present around each range.
end_padding_width	(Optional) Specifies a different end-padding width.
Inputs	
data_in	T<N, D1..., Dn> Padded input data
lengths	(i64) Num of elements in each range. sum(lengths) = N. If not provided, considers all data as a single segment.
Outputs	
padding_sum	Sum of all start paddings, or of all paddings if end_padding_sum is not provided.
end_padding_sum	T<D1..., Dn> Sum of all end paddings, if provided.

Code#

caffe2/operators/sequence_ops.cc

GatherRanges#

Given DATA tensor of rank 1, and RANGES tensor of rank 3, gather corresponding ranges into a 1-D tensor OUTPUT. RANGES dimentions description: 1: represents list of examples within a batch 2: represents list features 3: two values which are start and length or a range (to be applied on DATA) Another output LENGTHS represents each example length within OUTPUT Example:

```
DATA  = [1, 2, 3, 4, 5, 6]
RANGES = [
  [
    [0, 1],
    [2, 2],
  ],
  [
    [4, 1],
    [5, 1],
  ]
]
OUTPUT = [1, 3, 4, 5, 6]
LENGTHS = [3, 2]
```

Interface#

Inputs	
DATA	Tensor of rank 1.
RANGES	Tensor of int32/int64 ranges, of dims (N, M, 2). Where N is number of examples and M is a size of each example. Last dimention represents a range in the format (start, lengths)
Outputs	
OUTPUT	1-D tensor of size sum of range lengths
LENGTHS	1-D tensor of size N with lengths over gathered data for each row in a batch. sum(LENGTHS) == OUTPUT.size()

Code#

caffe2/operators/utility_ops.cc

GaussianFill#

No documentation yet.

Code#

caffe2/operators/filler_op.cc

GetAllBlobNames#

Return a 1D tensor of strings containing the names of each blob in the active workspace.

Interface#

Arguments
include_shared (bool, default true) Whether to include blobs inherited from parent workspaces.
Outputs
blob_names 1D tensor of strings containing blob names.

Code#

caffe2/operators/workspace_ops.cc

GivenTensorFill#

No documentation yet.

Code#

caffe2/operators/given_tensor_fill_op.cc

GivenTensorInt64Fill#

No documentation yet.

Code#

caffe2/operators/given_tensor_fill_op.cc

GivenTensorIntFill#

No documentation yet.

Code#

caffe2/operators/given_tensor_fill_op.cc

GivenTensorStringFill#

No documentation yet.

Code#

caffe2/operators/given_tensor_fill_op.cc

HSoftmax#

Hierarchical softmax is an operator which approximates the softmax operator while giving significant training speed gains and

reasonably comparable performance. In this operator, instead of calculating the probabilities of all the classes, we calculate the probability of each step in the path from root to the target word in the hierarchy. The operator takes a 2-D tensor (Tensor) containing a batch of layers, a set of parameters represented by the weight matrix and bias terms, and a 1-D tensor (Tensor) holding labels, or the indices of the target class. The hierarchy has to be specified as an argument to the operator. The operator returns a 1-D tensor holding the computed log probability of the target class and a 2-D tensor of intermediate outputs (from the weight matrix and softmax from each step in the path from root to target class) which will be used by the gradient operator to compute gradients for all samples in the batch.

Interface#

Arguments	
hierarchy	Serialized HierarchyProto string containing list of vocabulary words and their paths from root of hierarchy to the leaf
Inputs	
X	Input data from previous layer
W	2D blob containing 'stacked' fully connected weight matrices. Each node in the hierarchy contributes one FC weight matrix if it has children nodes. Dimension is N*D, D is input dimension of data (X), N is sum of all output dimensions, or total number of nodes (excl root)
b	1D blob with N parameters
labels	int word_id of the target word
Outputs	
Y	1-D of log probability outputs, one per sample
intermediate_output	Extra blob to store the intermediate FC and softmax outputs for each node in the hierarchical path of a word. The outputs from samples are stored in consecutive blocks in the forward pass and are used in reverse order in the backward gradientOp pass

Code#

caffe2/operators/h_softmax_op.cc

HSoftmaxGradient#

No documentation yet.

Code#

caffe2/operators/h_softmax_op.cc

HSoftmaxSearch#

```
1 HSoftmaxSearch is an operator to generate the most possible paths given a
2 well-trained model and input vector. Greedy algorithm is used for pruning the
3 search tree.
```

Interface#

Arguments	
tree	Serialized TreeProto string containing a tree including all intermidate nodes and leafs. All nodes must have names for correct outputs
beam	beam used for pruning tree. The pruning algorithm is that only children, whose score is smaller than parent's score puls beam, will be propagated.
topN	Number of nodes in outputs
Inputs	
X	Input data from previous layer
W	The matrix trained from Softmax Ops
b	The bias traiend from Softmax Ops
Outputs	

Y_names	The name of selected nodes and leafs. For nodes, it will be the name defined in the tree. For leafs, it will be the index of the word in the tree.
Y_scores	The corresponding scores of Y_names

Code#

caffe2/operators/h_softmax_op.cc

HalfToFloat#

No documentation yet.

Code#

caffe2/operators/half_float_ops.cc

HasElements#

Returns true iff the input tensor has size > 0

Interface#

Inputs

tensor	Tensor of any type.

Outputs

has_elements	Scalar bool tensor. True if input is not empty.

Code#

caffe2/operators/utility_ops.cc

HuffmanTreeHierarchy#

```
1  HuffmanTreeHierarchy is an operator to generate huffman tree hierarchy given
2  the input labels. It returns the tree as seralized HierarchyProto
```

Interface#

Arguments

num_classes	The number of classes used to build the hierarchy.

Inputs

Labels	The labels vector

Outputs

Hierarch	Huffman coding hierarchy of the labels

Code#

caffe2/operators/h_softmax_op.cc

Im2Col#

The Im2Col operator from Matlab.

Interface#

Inputs

X	4-tensor in NCHW or NHWC.

Outputs

Y	4-tensor. For NCHW: N x (C x kH x kW) x outH x outW.For NHWC: N x outH x outW x (kH x kW x C

Code#

caffe2/operators/im2col_op.cc

IndexFreeze#

Freezes the given index, disallowing creation of new index entries. Should not be called concurrently with IndexGet.

Interface#

Inputs

handle	Pointer to an Index instance.

Outputs

handle	The input handle.

Code#

caffe2/operators/index_ops.cc

IndexGet#

Given an index handle and a tensor of keys, return an Int tensor of same shape containing the indices for each of the keys. If the index is frozen, unknown entries are given index 0. Otherwise, new entries are added into the index. If an insert is necessary but max_elements has been reached, fail.

Interface#

Inputs

handle	Pointer to an Index instance.
keys	Tensor of keys to be looked up.

Outputs

indices	Indices for each of the keys.

Code#

caffe2/operators/index_ops.cc

IndexLoad#

Loads the index from the given 1-D tensor. Elements in the tensor will be given consecutive indexes starting at 1. Fails if tensor contains repeated elements.

Interface#

Arguments

skip_first_entry	If set, skips the first entry of the tensor. This allows to load tensors that are aligned with an embedding, where the first entry corresponds to the default 0 index entry.

Inputs

handle	Pointer to an Index instance.
items	1-D tensor with elements starting with index 1.

Outputs

handle	The input handle.

Code#

caffe2/operators/index_ops.cc

IndexSize#

Returns the number of entries currently present in the index.

Interface#

Inputs	
handle	Pointer to an Index instance.
Outputs	
items	Scalar int64 tensor with number of entries.

Code#

caffe2/operators/index_ops.cc

IndexStore#

Stores the keys of this index in a 1-D tensor. Since element 0 is reserved for unknowns, the first element of the output tensor will be element of index 1.

Interface#

Inputs	
handle	Pointer to an Index instance.
Outputs	
items	1-D tensor with elements starting with index 1.

Code#

caffe2/operators/index_ops.cc

InstanceNorm#

Carries out instance normalization as described in the paper https://arxiv.org/abs/1607.08022. Depending on the mode it is being run, there are multiple cases for the number of outputs, which we list below: Output case #1: output Output case #2: output, saved_mean – don't use, doesn't make sense but won't

1 crash

Output case #3: output, saved_mean, saved_inv_stdev – Makes sense for training

1 only
2

For training mode, type 3 is faster in the sense that for the backward pass, it is able to reuse the saved mean and inv_stdev in the gradient computation.

Interface#

Arguments	
epsilon	The epsilon value to use to avoid division by zero.
order	A StorageOrder string.
Inputs	
input	The input 4-dimensional tensor of shape NCHW or NHWC depending on the order parameter.
scale	The input 1-dimensional scale tensor of size C.
bias	The input 1-dimensional bias tensor of size C.

Outputs	
output	The output 4-dimensional tensor of the same shape as input.
saved_mean	Optional saved mean used during training to speed up gradient computation. Should not be used for testing.
saved_inv_stdev	Optional saved inverse stdev used during training to speed up gradient computation. Should not be used for testing.

Code#

caffe2/operators/instance_norm_op.cc

InstanceNormGradient#

No documentation yet.

Code#

caffe2/operators/instance_norm_op.cc

IntIndexCreate#

Creates a dictionary that maps int32 keys to consecutive integers from 1 to max_elements. Zero is reserved for unknown keys.

Interface#

Arguments	
max_elements	Max number of elements, including the zero entry.
Outputs	
handler	Pointer to an Index instance.

Code#

caffe2/operators/index_ops.cc

IsEmpty#

Returns true iff the input tensor has size == 0

Interface#

Inputs	
tensor	Tensor of any type.
Outputs	
is_empty	Scalar bool tensor. True if input is empty.

Code#

caffe2/operators/utility_ops.cc

LE#

Performs element-wise comparison <= (with limited broadcast support). If necessary the right-hand-side argument will be broadcasted to match the shape of left-hand-side argument. When broadcasting is specified, the second tensor can either be of size 1 (a scalar value), or having its shape as a contiguous subset of the first tensor's shape. The starting of the mutually equal shape is specified by the argument "axis", and if it is not set, suffix matching is assumed. 1-dim expansion doesn't work yet. For example, the following tensor shapes are supported (with broadcast=1):

1. shape(A) = (2, 3, 4, 5), shape(B) = (,), i.e. B is a scalar
2. shape(A) = (2, 3, 4, 5), shape(B) = (5,)
3. shape(A) = (2, 3, 4, 5), shape(B) = (4, 5)

```
4  shape(A) = (2, 3, 4, 5), shape(B) = (3, 4), with axis=1
5  shape(A) = (2, 3, 4, 5), shape(B) = (2), with axis=0
6
```

Argument broadcast=1 needs to be passed to enable broadcasting.

Interface#

Arguments	
broadcast	Pass 1 to enable broadcasting
axis	If set, defines the broadcast dimensions. See doc for details.
Inputs	
A	First operand, should share the type with the second operand.
B	Second operand. With broadcasting can be of smaller size than A. If broadcasting is disabled it should be of the same size.
Outputs	
C	Result, has same dimensions and A and type bool

Code#

caffe2/operators/elementwise_op_schema.cc

LRN#

No documentation yet.

Code#

caffe2/operators/local_response_normalization_op.cc

LRNGradient#

No documentation yet.

Code#

caffe2/operators/local_response_normalization_op.cc

LSTMUnit#

LSTMUnit computes the activations of a standard LSTM (without peephole connections), in a sequence-length aware fashion.

Concretely, given the (fused) inputs X (TxNxD), the previous cell state (NxD), and the sequence lengths (N), computes the LSTM activations, avoiding computation if the input is invalid (as in, the value at X{t}[n] >= seqLengths[n].

Code#

caffe2/operators/lstm_unit_op.cc

LSTMUnitGradient#

No documentation yet.

Code#

caffe2/operators/lstm_unit_op.cc

LT#

Performs element-wise comparison < (with limited broadcast support). If necessary the right-hand-side argument will be broadcasted

to match the shape of left-hand-side argument. When broadcasting is specified, the second tensor can either be of size 1 (a scalar value), or having its shape as a contiguous subset of the first tensor's shape. The starting of the mutually equal shape is specified by the argument "axis", and if it is not set, suffix matching is assumed. 1-dim expansion doesn't work yet. For example, the following tensor shapes are supported (with broadcast=1):

```
1  shape(A) = (2, 3, 4, 5), shape(B) = (,), i.e. B is a scalar
2  shape(A) = (2, 3, 4, 5), shape(B) = (5,)
3  shape(A) = (2, 3, 4, 5), shape(B) = (4, 5)
4  shape(A) = (2, 3, 4, 5), shape(B) = (3, 4), with axis=1
5  shape(A) = (2, 3, 4, 5), shape(B) = (2), with axis=0
6
```

Argument broadcast=1 needs to be passed to enable broadcasting.

Interface#

Arguments	
broadcast	Pass 1 to enable broadcasting
axis	If set, defines the broadcast dimensions. See doc for details.
Inputs	
A	First operand, should share the type with the second operand.
B	Second operand. With broadcasting can be of smaller size than A. If broadcasting is disabled it should be of the same size.
Outputs	
C	Result, has same dimensions and A and type bool

Code#

caffe2/operators/elementwise_op_schema.cc

LabelCrossEntropy#

Operator computes the cross entropy between the input and the label set. In practice, it is most commonly used at the end of models, after the SoftMax operator and before the AveragedLoss operator. Note that LabelCrossEntropy assumes that the label provided is either a 1D array of size N (batch size), or a 2D array of size N x 1 (batch size). Each entry in the label vector indicates which is the correct class; as such, each entry must be between 0 and D - 1, inclusive, where D is the total number of classes. The formula used is:

```
1                 Y[i] = -log(X[i][j])
2
```

where (i, j) is the classifier's prediction of the jth class (the correct one), and i is the batch size. Each log has a lower limit for numerical stability.

Interface#

Inputs	
X	Input blob from the previous layer, which is almost always the result of a softmax operation; X is a 2D array of size N x D, where N is the batch size and D is the number of classes
label	Blob containing the labels used to compare the input
Outputs	
Y	Output blob after the cross entropy computation

Code#

caffe2/operators/cross_entropy_op.cc

LabelCrossEntropyGradient#

No documentation yet.

Code#

caffe2/operators/cross_entropy_op.cc

LastNWindowCollector#

Collect the last N rows from input data. The purpose is to keep track of data accross batches, so for example suppose the LastNWindowCollector is called successively with the following input [1,2,3,4] [5,6,7] [8,9,10,11] And the number of items is set to 6, then the output after the 3rd call will contain the following elements: [6,7,8,9,10,11] No guarantee is made on the ordering of elements in input. So a valid value for output could have been [11,10,9,8,7,6] Also, this method works for any order tensor, treating the first dimension as input rows and keeping the last N rows seen as input. So for instance: [[1,2],[2,3],[3,4],[4,5]] [[5,6],[6,7],[7,8]] [[8,9],[9,10],[10,11],[11,12]] A possible output would be [[6,7],[7,8],[8,9],[9,10],[10,11],[11,12]]

Interface#

Arguments

num_to_collect The number of random samples to append for each positive samples

Inputs

Output data Copy, just to say that the output depends on the previous iterations

Outputs

The last n Data stored in sessions

Code#

caffe2/operators/last_n_window_collector.cc

LengthsMean#

Applies ‘Mean’ to each segment of the input tensor. Segments are defined by their LENGTHS. LENGTHS is a vector that maps each of the first dimension slices of the DATA to a particular group (segment). Values belonging to the same segment are aggregated together. For example LENGTHS = [2, 1] stands for segments DATA[0..1] and DATA[2] The first dimension of the output is equal to the number of input segments, i.e. len(LENGTHS) . Other dimensions are inherited from the input tensor. Mean computes the element-wise mean of the input slices. Operation doesn’t change the shape of the individual blocks.

Interface#

Inputs

DATA Input tensor, slices of which are aggregated.

LENGTHS Vector with the same sum of elements as the first dimension of DATA

Outputs

OUTPUT Aggregated output tensor. Has the first dimension of len(LENGTHS)

Code#

caffe2/operators/segment_reduction_op.cc

LengthsMeanGradient#

No documentation yet.

Code#

caffe2/operators/segment_reduction_op.cc

LengthsPartition#

LengthsPartition splits the input int tensor into multiple ones according to the second tensor. The first dimension is expected to be the tensor that describes lengths of the elements. Takes the second input and partitions it to shards according to the remainder of values modulo the number of partitions. It requires the second tensor to be a 1D-tensor of the integral type. The first tensor should

be 1D-tensor of int32 that would represent the lengths of the elements in the input. The number of partitions is derived as (num_output / num_input). If additional inputs are present they must have the same shape as the first input, optionally with extra trailing dimensions. They will be partitioned accordingly to the first input. Optional arg 'pack_first_input' transforms the first tensor values as X_ij / num_partitions. Outputs are ordered as X_0_part_0, X_1_part_0, ..., X_N-1_part_0, X_0_part_1, ..., X_N-1_part_K-1

Interface#

Arguments

pack_first_input (int, default 0) If set, the operator transforms the first tensor values as floor(X_ij / num_partitions)

Inputs

input	Input tensor containing data to be partitioned. The number of input tensors might be greater than 1 but must have the same shape as the previous tensors.

Outputs

partitions	Output Partitions. The number of output tensors has to be a multiple of the number of input tensors.

Code#

caffe2/operators/partition_ops.cc

LengthsRangeFill#

Convert a length vector to a range sequene. For example, input=[4,3,1], the output would be [0,1,2,3,0,1,2,0].

Interface#

Inputs

lengths	1D tensor of int32 or int64 segment lengths.

Outputs

range_sequence	1D tensor whose size is the sum of lengths

Code#

caffe2/operators/filler_op.cc

LengthsSum#

Applies 'Sum' to each segment of the input tensor. Segments are defined by their LENGTHS. LENGTHS is a vector that maps each of the first dimension slices of the DATA to a particular group (segment). Values belonging to the same segment are aggregated together. For example LENGTHS = [2, 1] stands for segments DATA[0..1] and DATA[2] The first dimension of the output is equal to the number of input segments, i.e. len(LENGTHS) . Other dimensions are inherited from the input tensor. Summation is done element-wise across slices of the input tensor and doesn't change the shape of the individual blocks.

Interface#

Inputs

DATA	Input tensor, slices of which are aggregated.
LENGTHS	Vector with the same sum of elements as the first dimension of DATA

Outputs

OUTPUT	Aggregated output tensor. Has the first dimension of len(LENGTHS)

Code#

caffe2/operators/segment_reduction_op.cc

LengthsSumGradient#

No documentation yet.

Code#

caffe2/operators/segment_reduction_op.cc

LengthsToRanges#

Given a vector of segment lengths, calculates offsets of each segment and packs them next to the lengths. For the input vector of length N the output is a Nx2 matrix with (offset, lengths) packaged for each segment. Output is going to have the same type as input. For long tensors explicit casting from int32 to int64 might be necessary prior to this op. For example, [1, 3, 0, 2] transforms into [[0, 1], [1, 3], [4, 0], [4, 2]] .

Interface#

Inputs

lengths 1D tensor of int32 or int64 segment lengths.

Outputs

ranges 2D tensor of shape len(lengths) X 2 and the same type as lengths

Code#

caffe2/operators/utility_ops.cc

LengthsToSegmentIds#

Given a vector of segment lengths, returns a zero-based, consecutive vector of segment_ids. For example, [1, 3, 0, 2] will produce [0, 1, 1, 1, 3, 3]. In general, the inverse operation is SegmentIdsToLengths. Notice though that trailing empty sequence lengths can't be properly recovered from segment ids.

Interface#

Inputs

lengths 1D tensor of int32 or int64 segment lengths.

Outputs

segment_ids 1D tensor of length sum(lengths)

Code#

caffe2/operators/utility_ops.cc

LengthsToShape#

No documentation yet.

Code#

caffe2/operators/utility_ops.cc

LengthsToWeights#

Similar as LengthsToSegmentIds but output vector of segment weights derived by lengths. i.e 1/pow(length, power)

Interface#

Arguments

power n of 1/pow(length,n) for normalization

Inputs

lengths 1-D int32_t or int64_t tensor of lengths

Outputs

a vector of weights 1-D float tensor of weights by length

Code#

caffe2/operators/utility_ops.cc

LengthsWeightedSum#

Applies 'WeightedSum' to each segment of the input tensor. Segments are defined by their LENGTHS. LENGTHS is a vector that maps each of the first dimension slices of the DATA to a particular group (segment). Values belonging to the same segment are aggregated together. For example LENGTHS = [2, 1] stands for segments DATA[0..1] and DATA[2] The first dimension of the output is equal to the number of input segments, i.e. len(LENGTHS) . Other dimensions are inherited from the input tensor. Input slices are first scaled by SCALARS and then summed element-wise. It doesn't change the shape of the individual blocks.

Interface#

Arguments	
grad_on_weights	Produce also gradient for weights. For now it's only supported in Lengths-based operators
Inputs	
DATA	Input tensor for the summation
SCALARS	Scalar multipliers for the input slices. Must be a vector with the length matching the first dimension of DATA
LENGTHS	Vector with the same sum of elements as the first dimension of DATA
Outputs	
OUTPUT	Aggregated output tensor. Has the first dimension of len(LENGTHS)

Code#

caffe2/operators/segment_reduction_op.cc

LengthsWeightedSumGradient#

No documentation yet.

Code#

caffe2/operators/segment_reduction_op.cc

LengthsWeightedSumWithMainInputGradient#

No documentation yet.

Code#

caffe2/operators/segment_reduction_op.cc

Load#

The Load operator loads a set of serialized blobs from a db. It takes no input and [0, infinity) number of outputs, using the db keys to match the db entries with the outputs. If an input is passed, then it is assumed that that input blob is a DBReader to load from, and we ignore the db and db_type arguments.

Interface#

Arguments	
absolute_path	(int, default 0) if set, use the db path directly and do not prepend the current root folder of the workspace.
db	(string) the path to the db to load.
db_type	(string) the type of the db.
keep_device	(int, default 0) if nonzero, the blobs are loaded into the device that is specified in the serialized BlobProto. Otherwise, the device will be set as the one that the Load operator is being run under.

load_all	(int, default 0) if nonzero, will load all blobs pointed to by the db to the workspace overwriting/creating blobs as needed.

Code#

caffe2/operators/load_save_op.cc

Log#

Calculates the natural log of the given input tensor, element-wise. This operation can be done in an in-place fashion too, by providing the same input and output blobs.

Interface#

Inputs	
input	Input tensor
Outputs	
output	The natural log of the input tensor computed element-wise

Code#

caffe2/operators/log_op.cc

LongIndexCreate#

Creates a dictionary that maps int64 keys to consecutive integers from 1 to max_elements. Zero is reserved for unknown keys.

Interface#

Arguments	
max_elements	Max number of elements, including the zero entry.
Outputs	
handler	Pointer to an Index instance.

Code#

caffe2/operators/index_ops.cc

LpPool#

LpPool consumes an input blob X and applies L-p pooling across the the blob according to kernel sizes, stride sizes, and pad lengths defined by the ConvPoolOpBase operator. L-p pooling consisting of taking the L-p norm of a subset of the input tensor according to the kernel size and downsampling the data into the output blob Y for further processing.

Interface#

Inputs	
X	Input data tensor from the previous operator; dimensions depend on whether the NCHW or NHWC operators are being used. For example, in the former, the input 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 the width of the data. The corresponding permutation of dimensions is used in the latter case.
Outputs	
Y	Output data tensor from L-p pooling across the input tensor. Dimensions will vary based on various kernel, stride, and pad sizes.

Code#

caffe2/operators/lp_pool_op.cc

LpPoolGradient#

No documentation yet.

Code#

caffe2/operators/lp_pool_op.cc

MSRAFill#

No documentation yet.

Code#

caffe2/operators/filler_op.cc

MakeTwoClass#

Given a vector of probabilities, this operator transforms this into a 2-column matrix with complimentary probabilities for binary classification. In explicit terms, given the vector X, the output Y is vstack(1 - X, X).

Interface#

Inputs	
X	Input vector of probabilities
Outputs	
Y	2-column matrix with complimentary probabilities of X for binary classification

Code#

caffe2/operators/cross_entropy_op.cc

MakeTwoClassGradient#

No documentation yet.

Code#

caffe2/operators/cross_entropy_op.cc

MarginRankingCriterion#

MarginRankingCriterion takes two input data X1 (Tensor), X2 (Tensor), and label Y (Tensor) to produce the loss (Tensor) where the loss function, loss(X1, X2, Y) = max(0, -Y * (X1 - X2) + margin), is applied to the tensor elementwise. If y == 1 then it assumed the first input should be ranked higher (have a larger value) than the second input, and vice-versa for y == -1.

Interface#

Inputs	
X1	The left input vector as a 1-dim TensorCPU.
X2	The right input vector as a 1-dim TensorCPU.
Y	The label as a 1-dim TensorCPU with int value of 1 or -1.
Outputs	
loss	The output loss with the same dimensionality as X1.

Code#

caffe2/operators/margin_ranking_criterion_op.cc

MarginRankingCriterionGradient#

MarginRankingCriterionGradient takes both X1, X2, Y and dY and uses them to update dX1, and dX2 according to the chain rule and derivatives of the loss function.

Code#

caffe2/operators/margin_ranking_criterion_op.cc

MatMul#

Matrix multiplication Y = A * B, where A has size (M x K), B has size (K x N), and Y will have a size (M x N).

Interface#

Arguments

trans_a	Pass 1 to transpose A before multiplication
trans_b	Pass 1 to transpose B before multiplication

Inputs

A	2D matrix of size (M x K)
B	2D matrix of size (K x N)

Outputs

Y	2D matrix of size (M x N)

Code#

caffe2/operators/matmul_op.cc

Max#

Element-wise max of each of the input tensors. The first input tensor can be used in-place as the output tensor, in which case the max will be done in place and results will be accumulated in input0. All inputs and outputs must have the same shape and data type.

Interface#

Inputs

data_0	First of the input tensors. Can be inplace.

Outputs

max	Output tensor. Same dimension as inputs.

Code#

caffe2/operators/utility_ops.cc

MaxGradient#

No documentation yet.

Code#

caffe2/operators/utility_ops.cc

MaxPool#

MaxPool consumes an input blob X and applies max pooling across the the blob according to kernel sizes, stride sizes, and pad lengths defined by the ConvPoolOpBase operator. Max pooling consisting of taking the maximumvalue of a subset of the input tensor according to the kernel size and downsampling the data into the output blob Y for further processing.

Interface#

Inputs

X	Input data tensor from the previous operator; dimensions depend on whether the NCHW or NHWC operators are being used. For example, in the former, the input 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 the width of the data. The corresponding permutation of dimensions is used in the latter case.

Outputs

Y	Output data tensor from max pooling across the input tensor. Dimensions will vary based on various kernel, stride, and pad sizes.

Code#

caffe2/operators/pool_op.cc

MaxPoolGradient#

No documentation yet.

Code#

caffe2/operators/pool_op.cc

Mul#

Performs element-wise binary multiplication (with limited broadcast support). If necessary the right-hand-side argument will be broadcasted to match the shape of left-hand-side argument. When broadcasting is specified, the second tensor can either be of size 1 (a scalar value), or having its shape as a contiguous subset of the first tensor's shape. The starting of the mutually equal shape is specified by the argument "axis", and if it is not set, suffix matching is assumed. 1-dim expansion doesn't work yet. For example, the following tensor shapes are supported (with broadcast=1):

```
1  shape(A) = (2, 3, 4, 5), shape(B) = (,), i.e. B is a scalar
2  shape(A) = (2, 3, 4, 5), shape(B) = (5,)
3  shape(A) = (2, 3, 4, 5), shape(B) = (4, 5)
4  shape(A) = (2, 3, 4, 5), shape(B) = (3, 4), with axis=1
5  shape(A) = (2, 3, 4, 5), shape(B) = (2), with axis=0
6
```

Argument broadcast=1 needs to be passed to enable broadcasting.

Interface#

Arguments

broadcast	Pass 1 to enable broadcasting
axis	If set, defines the broadcast dimensions. See doc for details.

Inputs

A	First operand, should share the type with the second operand.
B	Second operand. With broadcasting can be of smaller size than A. If broadcasting is disabled it should be of the same size.

Outputs

C	Result, has same dimensions and type as A

Code#

caffe2/operators/elementwise_op_schema.cc

MultiClassAccuracy#

Respectively compute accuracy score for each class given a number of instances and predicted scores of each class for each instance.

Interface#

Inputs	
prediction	2-D float tensor (N,D,) of predicted scores of each class for each data. N is the number of instances, i.e., batch size. D is number of possible classes/labels.
labels	1-D int tensor (N,) of labels for each instance.
Outputs	
accuracies	1-D float tensor (D,) of accuracy for each class. If a class has no instance in the batch, its accuracy score is set to zero.
amounts	1-D int tensor (D,) of number of instances for each class in the batch.

Code#

caffe2/operators/multi_class_accuracy_op.cc

NCHW2NHWC#

The operator switches the order of data in a tensor from NCHW- sample index N, channels C, height H and width W, to the NHWC order.

Interface#

Inputs	
data	The input data (Tensor) in the NCHW order.
Outputs	
output	The output tensor (Tensor) in the NHWC order.

Code#

caffe2/operators/order_switch_ops.cc

NHWC2NCHW#

The operator switches the order of data in a tensor from NHWC- sample index N, height H, width H and channels C, to the NCHW order.

Interface#

Inputs	
data	The input data (Tensor) in the NHWC order.
Outputs	
output	The output tensor (Tensor) in the NCHW order.

Code#

caffe2/operators/order_switch_ops.cc

Negative#

Computes the element-wise negative of the input.

Interface#

Inputs	
X	1D input tensor
Outputs	
Y	1D input tensor

Code#

caffe2/operators/negative_op.cc

Normalize#

Given a matrix, apply L2-normalization along the last dimension.

Code#

caffe2/operators/normalize_op.cc

NormalizeGradient#

No documentation yet.

Code#

caffe2/operators/normalize_op.cc

Not#

Performs element-wise negation.

Interface#

Inputs	
X	Input tensor of type bool.
Outputs	
Y	Output tensor of type bool.

Code#

caffe2/operators/elementwise_op_schema.cc

OneHot#

Given a sequence of indices, one for each example in a batch, returns a matrix where each inner dimension has the size of the index and has 1.0 in the index active in the given example, and 0.0 everywhere else.

Interface#

Inputs	
indices	The active index for each example in the batch.
index_size_tensor	Scalar with the size of the index.
Outputs	
one_hots	Matrix of size len(indices) x index_size

Code#

caffe2/operators/one_hot_ops.cc

Or#

Performs element-wise logical operation or (with limited broadcast support). Both input operands should be of type bool . If necessary the right-hand-side argument will be broadcasted to match the shape of left-hand-side argument. When broadcasting is specified, the second tensor can either be of size 1 (a scalar value), or having its shape as a contiguous subset of the first tensor’s shape. The starting of the mutually equal shape is specified by the argument “axis”, and if it is not set, suffix matching is assumed. 1-dim expansion doesn’t work yet. For example, the following tensor shapes are supported (with broadcast=1):

```
1  shape(A) = (2, 3, 4, 5), shape(B) = (,), i.e. B is a scalar
2  shape(A) = (2, 3, 4, 5), shape(B) = (5,)
3  shape(A) = (2, 3, 4, 5), shape(B) = (4, 5)
4  shape(A) = (2, 3, 4, 5), shape(B) = (3, 4), with axis=1
5  shape(A) = (2, 3, 4, 5), shape(B) = (2), with axis=0
6
```

Argument broadcast=1 needs to be passed to enable broadcasting.

Interface#

Arguments	
broadcast	Pass 1 to enable broadcasting
axis	If set, defines the broadcast dimensions. See doc for details.
Inputs	
A	First operand.
B	Second operand. With broadcasting can be of smaller size than A. If broadcasting is disabled it should be of the same size.
Outputs	
C	Result, has same dimensions and A and type bool

Code#

caffe2/operators/elementwise_op_schema.cc

PRelu#

PRelu takes input data (Tensor) and slope tensor as input, and produces one output data (Tensor) where the function `f(x) = slope * x for x < 0` , `f(x) = x for x >= 0` ., is applied to the data tensor elementwise.

Interface#

Inputs	
X	1D input tensor
Slope	1D slope tensor. If Slope is of size 1, the value is sharedacross different channels
Outputs	
Y	1D input tensor

Code#

caffe2/operators/prelu_op.cc

PReluGradient#

PReluGradient takes both Y and dY and uses this to update dX and dW according to the chain rule and derivatives of the rectified linear function.

Code#

caffe2/operators/prelu_op.cc

PackSegments#

Map N dim tensor to N+1 dim based on length blob. Sequences that are shorter than the longest sequence are padded with zeros.

Interface#

Arguments	
pad_minf	Padding number in the packed segments. Use true to pad -infinity, otherwise pad zeros

Inputs

lengths	1-d int/long tensor contains the length in each of the output.
tensor	N dim Tensor.

Outputs

packed_tensor	N + 1 dim Tesorwhere dim(1) is the max length, dim(0) is the batch size.

Code#

caffe2/operators/pack_segments.cc

PadEmptySamples#

Pad empty field given lengths and index features, Input(0) is a blob pointing to the lengths of samples in one batch, [Input(1),... Input(num_fields)] a list of tensors containing the data for each field of the features. PadEmptySamples is thread safe.

Interface#

Inputs

lengths	A blob containing a pointer to the lengths.

Outputs

out_lengths	Tensor containing lengths with empty sample padded.

Code#

caffe2/operators/sequence_ops.cc

PadImage#

PadImage pads values around the boundary of an image according to the pad values and stride sizes defined by the ConvPoolOpBase operator.

Interface#

Inputs

X	Input data tensor from the previous operator; dimensions depend on whether the NCHW or NHWC operators are being used. For example, in the former, the input 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 the width of the data. The corresponding permutation of dimensions is used in the latter case.

Outputs

Y	Output data tensor from padding the H and W dimensions on the tensor. Dimensions will vary based on various pad and stride sizes.

Code#

caffe2/operators/pad_op.cc

PadImageGradient#

No documentation yet.

Code#

caffe2/operators/pad_op.cc

PairWiseLoss#

Operator computes the pair wise loss between all pairs within a batch using the logit loss function on the difference in scores between pairs

Interface#

Inputs

X	Input blob from the previous layer, which is almost always the result of a softmax operation; X is a 2D array of size N x 1where N is the batch size. For more info: D. Sculley, Large Scale Learning to Rank. https://www.eecs.tufts.edu/~dsculley/papers/large-scale-rank.pdf
label	Blob containing the labels used to compare the input

Outputs

Y	Output blob after the cross entropy computation

Code#

caffe2/operators/rank_loss_op.cc

PairWiseLossGradient#

No documentation yet.

Code#

caffe2/operators/rank_loss_op.cc

Partition#

Splits the input int tensor into multiple ones according to the first tensor. Takes the first input and partitions it to shards according to the remainder of values modulo the number of partitions. It requires that the first tensor is of integral type. The number of partitions is derived as (num_output / num_input). If additional inputs are present they must have the same shape as the first input, optionally with extra trailing dimensions. They will be partitioned accordingly to the first input. Optional arg 'pack_first_input' transforms the first tensor values as X_ij / num_partitions. Outputs are ordered as X_0_part_0, X_1_part_0, ..., X_N-1_part_0, X_0_part_1, ..., X_N-1_part_K-1

Interface#

Arguments

pack_first_input (int, default 0) If set, the operator transforms the first tensor values as floor(X_ij / num_partitions)

Inputs

input	Input tensor containing data to be partitioned. The number of input tensors might be greater than 1 but must have the same shape as the previous tensors.

Outputs

partitions	Output Partitions. The number of output tensors has to be a multiple of the number of input tensors.

Code#

caffe2/operators/partition_ops.cc

Perplexity#

Perplexity calculates how well a probability distribution predicts a sample. Perplexity takes a 1-D tensor containing a batch of probabilities. Each value in the tensor belongs to a different sample and represents the probability of the model predicting the true label for that sample. The operator returns a single (float) perplexity value for the batch.

Interface#

Inputs

probabilities	The input data as Tensor. It contains a batch oftrue label or target probabilities

Outputs

output	The output- a single (float) perplexity value for the batch

Code#

caffe2/operators/perplexity_op.cc

PiecewiseLinearTransform#

PiecewiseLinearTransform takes one inputs- predictions, a 2-D tensor (Tensor) of size (batch_size x prediction_dimensions), and three args - upper bounds, slopes and intercepts of piecewise functions. The output tensor has the same shape of input tensor and contains the piecewise linear transformation. Each feature dimension has its own piecewise linear transformation function. Therefore the size of piecewise function parameters are all (pieces x prediction_dimensions). Note that in each piece, low bound is excluded while high bound is included. Also the piecewise linear function must be continuous. If the input is binary predictions (Nx2 tensor), set the binary arg to true (see details below).

Interface#

Arguments

bounds	1-D vector of size (prediction_dimensions x (pieces+1)) contain the upper bounds of each piece of linear function. One special case is the first bound is the lower bound of whole piecewise function and we treat it the same as the left most functions
slopes	1-D vector of size (prediction_dimensions x pieces) containing the slopes of linear function
intercepts	1-D vector of size (prediction_dimensions x pieces) containing the intercepts of linear function
pieces	int value for the number of pieces for the piecewise linear function
binary	If set true, we assume the input is a Nx2 tensor. Its first column is negative predictions and second column is positive and negative + positive = 1. We just need one set of transforms for the positive column.

Inputs

predictions	2-D tensor (Tensor) of size (num_batches x num_classes) containing scores

Outputs

transforms	2-D tensor (Tensor) of size (num_batches x num_classes) containing transformed predictions

Code#

caffe2/operators/piecewise_linear_transform_op.cc

Print#

Logs shape and contents of input tensor to stderr or to a file.

Interface#

Arguments

to_file	(bool) if 1, saves contents to the root folder of the current workspace, appending the tensor contents to a file named after the blob name. Otherwise, logs to stderr.

Inputs

tensor	The tensor to print.

Code#

caffe2/operators/utility_ops.cc

QPSMetric#

QPSMetric operator syncronously updates metric storedcreate a blob that will store state that is required for computing QPSMetric. The only output of the operator will have blob with QPSMetricState as an output.

Interface#

Inputs

QPS_METRIC_STATE	Input Blob QPSMetricState, that needs to be updated
INPUT_BATCH	Input Blob containing a tensor with batch of the examples. First dimension of the batch will be used to get the number of examples in the batch.

Outputs

output	Blob with QPSMetricState

Code#

caffe2/operators/metrics_ops.cc

QPSMetricReport#

QPSMetricReport operator that syncronously consumes the QPSMetricState blob and reports the information about QPS.

Interface#

Outputs

output	Blob with QPSMetricState

Code#

caffe2/operators/metrics_ops.cc

RangeFill#

No documentation yet.

Code#

caffe2/operators/filler_op.cc

ReadNextBatch#

Read the next batch of examples out of the given cursor and data blobs. Input(0) is a blob pointing to a TreeCursor, and [Input(1),... Input(num_fields)] a list of tensors containing the data for each field of the dataset. ReadNextBatch is thread safe.

Interface#

Arguments

batch_size	Number of top-level entries to read.

Inputs

cursor	A blob containing a pointer to the cursor.
dataset_field_0	First dataset field

Outputs

field_0	Tensor containing the next batch for field 0.

Code#

caffe2/operators/dataset_ops.cc

ReadRandomBatch#

Read the next batch of examples out of the given cursor, idx blob, offset matrix and data blobs. Input(0) is a blob pointing to a TreeCursor, Input(1) is a blob pointing to the shuffled idx Input(2) is a blob pointing to the offset matrix and [Input(3),... Input(num_fields)] a list of tensors containing the data for each field of the dataset. ReadRandomBatch is thread safe.

Interface#

Arguments

batch_size	Number of top-level entries to read.

Inputs

cursor	A blob containing a pointer to the cursor.
idx	idx with a shuffled order.
offsetsmat	offset matrix containing length offset info.
dataset_field_0	First dataset field
Outputs	
field_0	Tensor containing the next batch for field 0.

Code#

caffe2/operators/dataset_ops.cc

ReceiveTensor#

Receives the tensor from another node.

Interface#

Arguments

src	(int) he rank to receive the tensor from.
tag	(int) a tag to receive the tensor with.
raw_buffer	(bool) if set, only send the content and assume that the receiver has already known the tensor's shape and information.
Inputs	
comm_world	The common world.
Y	In-place output. If raw_buffer is specified, Y should have pre-allocated data and type..
src	An int CPUtensor of size 1 specifying the rank. If given, this overrides the 'from' argument of the op.
tag	An int CPUtensor of size 1 specifying the tag to send the tensor with. This overrides the 'tag' argument of the op.
Outputs	
Y	The received tensor.
src	The sender that sent the message as a CPUTensor of size 1 and of type int.
tag	The tag that the message is sent with as a CPUTensor of size 1 and of type int.

Code#

caffe2/operators/communicator_op.cc

RecurrentNetwork#

Run the input network in a recurrent fashion. This can be used to implement fairly general recurrent neural networks (RNNs). The operator proceeds as follows.

- First, initialized the states from the input recurrent states - For each timestep T, apply the links (that map offsets from input/output

1 tensors into the inputs/outputs for the `step` network)

- Finally, alias the recurrent states to the specified output blobs. This is a fairly special-case meta-operator, and so the implementation is somewhat complex. It trades of generality (and frankly usability) against performance and control (compared to e.g. TF dynamic_rnn, Theano scan, etc). See the usage examples for a flavor of how to use it.

Code#

caffe2/operators/recurrent_network_op.cc

RecurrentNetworkGradient#

No documentation yet.

Code#

caffe2/operators/recurrent_network_op.cc

Reduce#

Does a reduce operation from every node to the root node. Currently only Sum is supported.

Interface#

Arguments	
root	(int, default 0) the root to run reduce into.
Inputs	
comm_world	The common world.
X	A tensor to be reduced.
Outputs	
Y	The reduced result on root, not set for other nodes.

Code#

caffe2/operators/communicator_op.cc

ReduceFrontMean#

Reduces the input tensor along the first dimension of the input tensor by applying 'Mean'. This op acts in a similar way to SortedSegmentMean and UnsortedSegmentMean but as if all input slices belong to a single segment. Mean computes the element-wise mean of the input slices. Operation doesn't change the shape of the individual blocks.

Interface#

Inputs	
DATA	Input tensor to be reduced on the first dimension
Outputs	
OUTPUT	Aggregated tensor

Code#

caffe2/operators/segment_reduction_op.cc

ReduceFrontMeanGradient#

No documentation yet.

Code#

caffe2/operators/segment_reduction_op.cc

ReduceFrontSum#

Reduces the input tensor along the first dimension of the input tensor by applying 'Sum'. This op acts in a similar way to SortedSegmentSum and UnsortedSegmentSum but as if all input slices belong to a single segment. Summation is done element-wise across slices of the input tensor and doesn't change the shape of the individual blocks.

Interface#

Inputs	
DATA	Input tensor to be reduced on the first dimension
Outputs	

OUTPUT Aggregated tensor

Code#

caffe2/operators/segment_reduction_op.cc

ReduceFrontSumGradient#

No documentation yet.

Code#

caffe2/operators/segment_reduction_op.cc

ReduceFrontWeightedSum#

Reduces the input tensor along the first dimension of the input tensor by applying 'WeightedSum'. This op acts in a similar way to SortedSegmentWeightedSum and UnsortedSegmentWeightedSum but as if all input slices belong to a single segment. Input slices are first scaled by SCALARS and then summed element-wise. It doesn't change the shape of the individual blocks.

Interface#

Arguments	
grad_on_weights	Produce also gradient for weights. For now it's only supported in Lengths-based operators
Inputs	
DATA	Input tensor for the summation
SCALARS	Scalar multipliers for the input slices. Must be a vector with the length matching the first dimension of DATA
Outputs	
OUTPUT	Aggregated tensor

Code#

caffe2/operators/segment_reduction_op.cc

ReduceFrontWeightedSumGradient#

No documentation yet.

Code#

caffe2/operators/segment_reduction_op.cc

ReduceTailSum#

Reduce the tailing dimensions

Interface#

Inputs	
mat	The matrix
Outputs	
output	Output

Code#

caffe2/operators/rowmul_op.cc

Relu#

Relu takes one input data (Tensor) and produces one output data (Tensor) where the rectified linear function, y = max(0, x), is applied to the tensor elementwise.

Interface#

Inputs	
X	1D input tensor
Outputs	
Y	1D input tensor

Code#

caffe2/operators/relu_op.cc

ReluGradient#

ReluGradient takes both Y and dY and uses this to update dX according to the chain rule and derivatives of the rectified linear function.

Code#

caffe2/operators/relu_op.cc

RemoveDataBlocks#

Shrink the data tensor by removing data blocks with given zero-based indices in the outermost dimension of the tensor. Indices are not assumed in any order or unique but with the range [0, blocks_size). Indices could be empty.

Interface#

Inputs	
data	a N-D data tensor, N >= 1
indices	zero-based indices of blocks to be removed
Outputs	
shrunk data	data after removing data blocks indexed by 'indices'

Code#

caffe2/operators/remove_data_blocks_op.cc

RemovePadding#

Remove padding around the edges of each segment of the input data. This is the reverse opration of AddPadding, and uses the same arguments and conventions for input and output data format.

Interface#

Arguments	
padding_width	Outer-size of padding to remove around each range.
end_padding_width	(Optional) Specifies a different end-padding width.
Inputs	
data_in	T<N, D1..., Dn> Input data
lengths	(i64) Num of elements in each range. sum(lengths) = N. If not provided, considers all data as a single segment.
Outputs	
data_out	(T<N - 2*padding_width, D1..., Dn>) Unpadded data.
lengths_out	(i64, optional) Lengths for each unpadded range.

Code#

caffe2/operators/sequence_ops.cc

ResetCounter#

Resets a count-down counter with initial value specified by the 'init_count' argument.

Interface#

Arguments	
init_count	Resets counter to this value, must be >= 0.
Inputs	
counter	A blob pointing to an instance of a new counter.
Outputs	
previous_value	(optional) Previous value of the counter.

Code#

caffe2/operators/counter_ops.cc

ResetCursor#

Resets the offsets for the given TreeCursor. This operation is thread safe.

Interface#

Inputs	
cursor	A blob containing a pointer to the cursor.

Code#

caffe2/operators/dataset_ops.cc

Reshape#

Reshape the input tensor similar to numpy.reshape. It takes a tensor as input and an optional tensor specifying the new shape. When the second input is absent, an extra argument shape must be specified. It outputs the reshaped tensor as well as the original shape. 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 going to be copied from the input tensor.

Interface#

Arguments	
shape	New shape
Inputs	
data	An input tensor.
new_shape	New shape.
Outputs	
reshaped	Reshaped data.
old_shape	Original shape.

Code#

caffe2/operators/utility_ops.cc

ResizeLike#

Produces tensor condaining data of first input and shape of second input.

Interface#

Inputs	
data	Tensor whose data will be copied into the output.
shape_tensor	Tensor whose shape will be applied to output.
Outputs	
output	Tensor with data of input 0 and shape of input 1.

Code#

caffe2/operators/utility_ops.cc

RetrieveCount#

Retrieve the current value from the counter.

Interface#

Inputs	
counter	A blob pointing to an instance of a counter.
Outputs	
count	current count value.

Code#

caffe2/operators/counter_ops.cc

ReversePackedSegs#

Reverse segments in a 3-D tensor (lengths, segments, embeddings,), leaving paddings unchanged. This operator is used to reverse input of a recurrent neural network to make it a BRNN.

Interface#

Inputs	
data	a 3-D (lengths, segments, embeddings,) tensor.
lengths	length of each segment.
Outputs	
reversed data	a (lengths, segments, embeddings,) tensor with each segment reversedand paddings unchanged.

Code#

caffe2/operators/reverse_packed_segs_op.cc

RoIPool#

Carries out ROI Pooling for Faster-RCNN. Depending on the mode, there are multiple output cases:

1. Output case #1: Y, argmaxes (train mode)
2. Output case #2: Y (test mode)

Interface#

Arguments	
is_test	If set, run in test mode and skip computation of argmaxes (used for gradient computation). Only one output tensor is produced. (Default: false).
order	A StorageOrder string (Default: “NCHW”).

pooled_h	The pooled output height (Default: 1).
pooled_w	The pooled output width (Default: 1).
spatial_scale	Multiplicative spatial scale factor to translate ROI coords from their input scale to the scale used when pooling (Default: 1.0).
Inputs	
X	The input 4-D tensor of data. Only NCHW order is currently supported.
rois	RoIs (Regions of Interest) to pool over. Should be a 2-D tensor of shape (num_rois, 5) given as [[batch_id, x1, y1, x2, y2], ...].
Outputs	
Y	RoI pooled output 4-D tensor of shape (num_rois, channels, pooled_h, pooled_w).
argmaxes	Argmaxes corresponding to indices in X used for gradient computation. Only output if arg "is_test" is false.

Code#

caffe2/operators/roi_pool_op.cc

RoIPoolGradient#

No documentation yet.

Code#

caffe2/operators/roi_pool_op.cc

RowMul#

Given a matrix A and column vector w, the output is the multiplication of row i of A and element i of w, e.g. C[i][j] = A[i][j] * w[i]. This operator should be deprecated when the gradient operator of Mul with broadcast is implemented.

Interface#

Inputs	
mat	The matrix
w	The column vector
Outputs	
output	Output

Code#

caffe2/operators/rowmul_op.cc

Save#

The Save operator saves a set of blobs to a db. It takes [1, infinity) number of inputs and has no output. The contents of the inputs are written into the db specified by the arguments.

Interface#

Arguments	
absolute_path	(int, default 0) if set, use the db path directly and do not prepend the current root folder of the workspace.
strip_regex	(string, default="") if set, characters in the provided blob names that match the regex will be removed prior to saving. Useful for removing device scope from blob names.
db	(string) the path to the db to load.
db_type	(string) the type of the db.

Code#

caffe2/operators/load_save_op.cc

Scale#

Scale takes one input data (Tensor) and produces one output data (Tensor) whose value is the input data tensor scaled element-wise.

Interface#

Arguments	
scale	(float, default 1.0) the scale to apply.

Code#

caffe2/operators/scale_op.cc

ScatterAssign#

Update slices of the tensor in-place by overriding current value. Note: The op pretty much ignores the exact shapes of the input arguments and cares only about sizes. It's done for performance consideration to avoid unnecessary reshapes. Only first dimension of X_0 is important, let's call it N. If M is the total size of X_0 and K is the size of INDICES then X_i is assumed to be of shape K x (M / N) regardless of the real shape. Note: Each update in INDICES is applied independently which means that if duplicated elements are present in INDICES arbitrary one will win. Currently only works on CPU because of access to INDICES.

Interface#

Inputs	
DATA	Tensor to be updated.
INDICES	1-D list of indices on the first dimensionof X_0 that need to be updated
SLICES	Update slices, with shape len(INDICES) + shape(X_0)[1:]
Outputs	
DATA	Has to be exactly the same tensor as the input 0

Code#

caffe2/operators/utility_ops.cc

ScatterWeightedSum#

Similar to WeightedSum, computes the weighted sum of several tensors, with the difference that inputs are sliced tensors. The first tensor has to be in-place and only slices of it on the first dimension as indexed by INDICES will be updated. Note: The op pretty much ignores the exact shapes of the input arguments and cares only about sizes. It's done for performance consideration to avoid unnecessary reshapes. Only first dimension of X_0 is important, let's call it N. If M is the total size of X_0 and K is the size of INDICES then X_i is assumed to be of shape K x (M / N) regardless of the real shape. Note: Each update in INDICES is applied independently which means that if duplicated elements are present in INDICES the corresponding slice of X_0 will be scaled multiple times. Manual collapsing of INDICES is required beforehand if necessary. Note: Updates are applied sequentially by inputs which might have undesired consequences if the input tensor is accessed concurrently by different op (e.g. when doing Hogwild). Other threads might see intermediate results even on individual slice level, e.g. X_0 scaled by weight_0 but without any updates applied. Currently only works on CPU because of access to INDICES.

Interface#

Inputs	
X_0	Tensor to be updated.
Weight_0	Scalar weight for X_0, applied only to slices affected.
INDICES	1-D list of indices on the first dimension of X_0 that need to be updated
X_1	Update slices, with shape len(INDICES) + shape(X_0)[1:]
Weight_1	Scalar weight for X_1 update
Outputs	
X_0	Has to be exactly the same tensor as the input 0

Code#

caffe2/operators/utility_ops.cc

SegmentIdsToLengths#

Transfers a vector of segment ids to a vector of segment lengths. This operation supports non-consecutive segment ids. Segments not appearing in the input vector will have length 0. If the second input is provided, the number of segments = the size of its first dimension. Otherwise, the number of segments = the last index in the first input vector + 1. In general, for consecutive, zero-based segment IDs, this is the inverse operation of LengthsToSegmentIds, except that a vector of segment IDs cannot represent empty segments at the end (if the second input is absent).

Interface#

Inputs	
segment_ids	1-D int32_t or int64_t tensor of segment ids
data (optional)	if provided, number of segments = the size of its first dimension
Outputs	
lengths	1-D int64_t tensor of segment lengths

Code#

caffe2/operators/utility_ops.cc

SegmentIdsToRanges#

Transfers a vector of segment ids to a vector of segment ranges. This operation supports non-consecutive segment ids. Segments not appearing in the input vector will have length 0. If the second input is provided, the number of segments = the size of its first dimension. Otherwise, the number of segments = the last index in the first input vector + 1.

Interface#

Inputs	
segment_ids	1-D int32_t or int64_t tensor of segment ids
data (optional)	if provided, number of segments = the size of its first dimension
Outputs	
lengths	1-D int64_t tensor of segment lengths

Code#

caffe2/operators/utility_ops.cc

SegmentOneHot#

Given a sequence of indices, segmented by the lengths tensor, returns a matrix that has the elements in each sequence set to 1.0, and 0.0 everywhere else.

Interface#

Inputs	
lengths	Size of each segment.
indices	Active indices, of size sum(lengths)
index_size_tensor	Size of the index
Outputs	
one_hots	Matrix of size len(lengths) x index_size

Code#

caffe2/operators/one_hot_ops.cc

SendTensor#

Sends the tensor to another node.

Interface#

Arguments	
dst	The rank to send the tensor to.
tag	(int) a tag to send the tensor with.
raw_buffer	(bool) if set, only send the content and assume that the receiver has already known the tensor's shape and information.
Inputs	
comm_world	The common world.
X	A tensor to be allgathered.
dst	An int CPUtensor of size 1 specifying the rank. If given, this overrides the 'to' argument of the op.
tag	An int CPUtensor of size 1 specifying the tag to send the tensor with. This overrides the 'tag' argument of the op.

Code#

caffe2/operators/communicator_op.cc

Shape#

Produce a 1D int64 tensor with the shape of the input tensor.

Code#

caffe2/operators/utility_ops.cc

Sigmoid#

Sigmoid takes one input data (Tensor) and produces one output data (Tensor) where the sigmoid function, y = 1 / (1 + exp(-x)), is applied to the tensor elementwise.

Interface#

Inputs	
X	1D input tensor
Outputs	
Y	1D output tensor

Code#

caffe2/operators/sigmoid_op.cc

SigmoidCrossEntropyWithLogits#

Given two matrices logits and targets, of same shape, (batch_size, num_classes), computes the sigmoid cross entropy between the two. Returns a tensor of shape (batch_size,) of losses for each example.

Interface#

Inputs	
logits	matrix of logits for each example and class.
targets	matrix of targets, same shape as logits.
Outputs	
xentropy	Vector with the total xentropy for each example.

Code#

caffe2/operators/cross_entropy_op.cc

SigmoidCrossEntropyWithLogitsGradient#

No documentation yet.

Code#

caffe2/operators/cross_entropy_op.cc

SigmoidGradient#

SigmoidGradient takes both Y and dY and uses this to update dX according to the chain rule and derivatives of the sigmoid function.

Code#

caffe2/operators/sigmoid_op.cc

Slice#

Produces a slice of the input tensor. Currently, only slicing in a single dimension is supported. Slices are passed as 2 1D vectors with starting and end indices for each dimension of the input data tensor. End indices are non-inclusive. If a negative value is passed for any of the start or end indices, it represent number of elements before the end of that dimension. Example:

```
data = [
    [1, 2, 3, 4],
    [5, 6, 7, 8],
]
starts = [0, 1]
ends = [-1, 3]

result = [
    [2, 3],
    [6, 7],
]
```

Interface#

Inputs

data	Tensor of data to extract slices from.
starts	1D tensor: start-indices for each dimension of data.
ends	1D tensor: end-indices for each dimension of data.

Outputs

output	Sliced data tensor.

Code#

caffe2/operators/utility_ops.cc

Softmax#

The operator computes the softmax normalized values for each layer in the batch of the given input. The input is a 2-D tensor (Tensor) of size (batch_size x input_feature_dimensions). The output tensor has the same shape and contains the softmax normalized values of the corresponding input.

Interface#

Inputs

input The input data as 2-D Tensor.

Outputs

output The softmax normalized output values with the same shape as input tensor.

Code#

caffe2/operators/softmax_op.cc

SoftmaxGradient#

No documentation yet.

Code#

caffe2/operators/softmax_op.cc

SoftmaxWithLoss#

Combined Softmax and Cross-Entropy loss operator. The operator computes the softmax normalized values for each layer in the batch of the given input, after which cross-entropy loss is computed. This operator is numerically more stable than separate Softmax and CrossEntropy ops. The inputs are a 2-D tensor (Tensor) of size (batch_size x input_feature_dimensions) and tensor of labels (ground truth). Output is tensor with the probability for each label for each example (N x D) and averaged loss (scalar). Use parameter spatial=1 to enable spatial softmax. Spatial softmax also supports special \"don't care\" label (-1) that is ignored when computing the loss. Use parameter label_prob=1 to enable inputting labels as a probability distribution.

1 Currently does not handle spatial=1 case.

Optional third input blob can be used to weight the samples for the loss. For the spatial version, weighting is by x,y position of the input.

Code#

caffe2/operators/softmax_with_loss_op.cc

SoftmaxWithLossGradient#

No documentation yet.

Code#

caffe2/operators/softmax_with_loss_op.cc

Softsign#

Calculates the softsign (x/1+ x) of the given input tensor element-wise. This operation can be done in an in-place fashion too, by providing the same input and output blobs.

Interface#

Inputs

input 1-D input tensor

Outputs

output The softsign (x/1+ x) values of the input tensor computed element-wise

Code#

caffe2/operators/softsign_op.cc

SoftsignGradient#

Calculates the softsign gradient (sgn(x)/(1+ x)^2) of the given input tensor element-wise.

Interface#

Inputs	
input	1-D input tensor
input	1-D input tensor
Outputs	
output	The softsign gradient (sgn(x)/(1+ x)^2) values of the input tensor computed element-wise

Code#

caffe2/operators/softsign_op.cc

SortAndShuffle#

Compute the sorted indices given a field index to sort by and break the sorted indices into chunks of shuffle_size * batch_size and shuffle each chunk, finally we shuffle between batches. If sort_by_field_idx is -1 we skip sort. For example, we have data sorted as 1,2,3,4,5,6,7,8,9,10,11,12 and batchSize = 2 and shuffleSize = 3, when we shuffle we get: [3,1,4,6,5,2] [12,10,11,8,9,7] After this we will shuffle among different batches with size 2 [3,1],[4,6],[5,2],[12,10],[11,8],[9,7] We may end up with something like [9,7],[5,2],[12,10], [4,6],[3,1],[11,8] Input(0) is a blob pointing to a TreeCursor, and [Input(1),... Input(num_fields)] a list of tensors containing the data for each field of the dataset. SortAndShuffle is thread safe.

Interface#

Inputs	
cursor	A blob containing a pointer to the cursor.
dataset_field_0	First dataset field
Outputs	
indices	Tensor containing sorted indices.

Code#

caffe2/operators/dataset_ops.cc

SortedSegmentMean#

Applies ‘Mean’ to each segment of input tensor. Segments need to be sorted and contiguous. See also UnsortedSegmentMean that doesn’t have this requirement. SEGMENT_IDS is a vector that maps each of the first dimension slices of the DATA to a particular group (segment). Values belonging to the same segment are aggregated together. The first dimension of the output is equal to the number of input segments, i.e. SEGMENT_IDS[-1]+1 . Other dimensions are inherited from the input tensor. Mean computes the element-wise mean of the input slices. Operation doesn’t change the shape of the individual blocks.

Interface#

Inputs	
DATA	Input tensor, slices of which are aggregated.
SEGMENT_IDS	Vector with the same length as the first dimension of DATA and values in the range 0..K-1 and in increasing order that maps each slice of DATA to one of the segments
Outputs	
OUTPUT	Aggregated output tensor. Has the first dimension of K (the number of segments).

Code#

caffe2/operators/segment_reduction_op.cc

SortedSegmentMeanGradient#

No documentation yet.

Code#

caffe2/operators/segment_reduction_op.cc

SortedSegmentRangeLogMeanExp#

Applies ‘LogMeanExp’ to each segment of input tensor. In order to allow for more efficient implementation of ‘LogMeanExp’, the input segments have to be contiguous and non-empty. SEGMENT_IDS is a vector that maps each of the first dimension slices of the DATA to a particular group (segment). Values belonging to the same segment are aggregated together. The first dimension of the output is equal to the number of input segments, i.e. SEGMENT_IDS[-1]+1 . Other dimensions are inherited from the input tensor. LogMeanExp computes the element-wise log of the mean of exponentials of input slices. Operation doesn’t change the shape of individual blocks.

Interface#

Inputs	
DATA	Input tensor to be aggregated
SEGMENT_IDS	Vector with the same length as the first dimension of DATA and values in the range 0..K-1 and in increasing order that maps each slice of DATA to one of the segments
Outputs	
OUTPUT	Aggregated tensor with the first dimension of K and the other dimentsions inherited from DATA

Code#

caffe2/operators/segment_reduction_op.cc

SortedSegmentRangeLogMeanExpGradient#

No documentation yet.

Code#

caffe2/operators/segment_reduction_op.cc

SortedSegmentRangeLogSumExp#

Applies ‘LogSumExp’ to each segment of input tensor. In order to allow for more efficient implementation of ‘LogSumExp’, the input segments have to be contiguous and non-empty. SEGMENT_IDS is a vector that maps each of the first dimension slices of the DATA to a particular group (segment). Values belonging to the same segment are aggregated together. The first dimension of the output is equal to the number of input segments, i.e. SEGMENT_IDS[-1]+1 . Other dimensions are inherited from the input tensor. LogSumExp computes the element-wise log of the sum of exponentials of input slices. Operation doesn’t change the shape of individual blocks.

Interface#

Inputs	
DATA	Input tensor to be aggregated
SEGMENT_IDS	Vector with the same length as the first dimension of DATA and values in the range 0..K-1 and in increasing order that maps each slice of DATA to one of the segments
Outputs	
OUTPUT	Aggregated tensor with the first dimension of K and the other dimentsions inherited from DATA

Code#

caffe2/operators/segment_reduction_op.cc

SortedSegmentRangeLogSumExpGradient#

No documentation yet.

Code#

caffe2/operators/segment_reduction_op.cc

SortedSegmentRangeMax#

Applies ‘Max’ to each segment of input tensor. In order to allow for more efficient implementation of ‘Max’, the input segments have to be contiguous and non-empty. SEGMENT_IDS is a vector that maps each of the first dimension slices of the DATA to a particular group (segment). Values belonging to the same segment are aggregated together. The first dimension of the output is equal to the number of input segments, i.e. SEGMENT_IDS[-1]+1 . Other dimensions are inherited from the input tensor. Max computation is done element-wise, so that each element of the output slice corresponds to the max value of the respective elements in the input slices. Operation doesn’t change the shape of individual blocks. This implementation imitates torch nn.Max operator. If the maximum value occurs more than once, the operator will return the first occurence of value. When computing the gradient using the backward propagation, the gradient input corresponding to the first occurence of the maximum value will be used.

Interface#

Inputs	
DATA	Input tensor to be aggregated
SEGMENT_IDS	Vector with the same length as the first dimension of DATA and values in the range 0..K-1 and in increasing order that maps each slice of DATA to one of the segments
Outputs	
OUTPUT	Aggregated tensor with the first dimension of K and the other dimentsions inherited from DATA

Code#

caffe2/operators/segment_reduction_op.cc

SortedSegmentRangeMaxGradient#

No documentation yet.

Code#

caffe2/operators/segment_reduction_op.cc

SortedSegmentRangeMean#

Applies ‘Mean’ to each segment of input tensor. In order to allow for more efficient implementation of ‘Mean’, the input segments have to be contiguous and non-empty. SEGMENT_IDS is a vector that maps each of the first dimension slices of the DATA to a particular group (segment). Values belonging to the same segment are aggregated together. The first dimension of the output is equal to the number of input segments, i.e. SEGMENT_IDS[-1]+1 . Other dimensions are inherited from the input tensor. Mean computation is done element-wise, so that each element of the output slice corresponds to the average value of the respective elements in the input slices. Operation doesn’t change the shape of individual blocks.

Interface#

Inputs	
DATA	Input tensor to be aggregated
SEGMENT_IDS	Vector with the same length as the first dimension of DATA and values in the range 0..K-1 and in increasing order that maps each slice of DATA to one of the segments
Outputs	
OUTPUT	Aggregated tensor with the first dimension of K and the other dimentsions inherited from DATA

Code#

caffe2/operators/segment_reduction_op.cc

SortedSegmentRangeMeanGradient#

No documentation yet.

Code#

caffe2/operators/segment_reduction_op.cc

SortedSegmentRangeSum#

Applies ‘Sum’ to each segment of input tensor. In order to allow for more efficient implementation of ‘Sum’, the input segments have to be contiguous and non-empty. SEGMENT_IDS is a vector that maps each of the first dimension slices of the DATA to a particular group (segment). Values belonging to the same segment are aggregated together. The first dimension of the output is equal to the number of input segments, i.e. SEGMENT_IDS[-1]+1 . Other dimensions are inherited from the input tensor. Summation is done element-wise across slices of the input tensor and doesn’t change the shape of the individual blocks.

Interface#

Inputs	
DATA	Input tensor to be aggregated
SEGMENT_IDS	Vector with the same length as the first dimension of DATA and values in the range 0..K-1 and in increasing order that maps each slice of DATA to one of the segments
Outputs	
OUTPUT	Aggregated tensor with the first dimension of K and the other dimentsions inherited from DATA

Code#

caffe2/operators/segment_reduction_op.cc

SortedSegmentRangeSumGradient#

No documentation yet.

Code#

caffe2/operators/segment_reduction_op.cc

SortedSegmentSum#

Applies ‘Sum’ to each segment of input tensor. Segments need to be sorted and contiguous. See also UnsortedSegmentSum that doesn’t have this requirement. SEGMENT_IDS is a vector that maps each of the first dimension slices of the DATA to a particular group (segment). Values belonging to the same segment are aggregated together. The first dimension of the output is equal to the number of input segments, i.e. SEGMENT_IDS[-1]+1 . Other dimensions are inherited from the input tensor. Summation is done element-wise across slices of the input tensor and doesn’t change the shape of the individual blocks.

Interface#

Inputs	
DATA	Input tensor, slices of which are aggregated.
SEGMENT_IDS	Vector with the same length as the first dimension of DATA and values in the range 0..K-1 and in increasing order that maps each slice of DATA to one of the segments
Outputs	
OUTPUT	Aggregated output tensor. Has the first dimension of K (the number of segments).

Code#

caffe2/operators/segment_reduction_op.cc

SortedSegmentSumGradient#

No documentation yet.

Code#

caffe2/operators/segment_reduction_op.cc

SortedSegmentWeightedSum#

Applies 'WeightedSum' to each segment of input tensor. Segments need to be sorted and contiguous. See also UnsortedSegmentWeightedSum that doesn't have this requirement. SEGMENT_IDS is a vector that maps each of the first dimension slices of the DATA to a particular group (segment). Values belonging to the same segment are aggregated together. The first dimension of the output is equal to the number of input segments, i.e. SEGMENT_IDS[-1]+1 . Other dimensions are inherited from the input tensor. Input slices are first scaled by SCALARS and then summed element-wise. It doesn't change the shape of the individual blocks.

Interface#

Arguments

grad_on_weights Produce also gradient for weights. For now it's only supported in Lengths-based operators

Inputs

DATA	Input tensor for the summation
SCALARS	Scalar multipliers for the input slices. Must be a vector with the length matching the first dimension of DATA
SEGMENT_IDS	Vector with the same length as the first dimension of DATA and values in the range 0..K-1 and in increasing order that maps each slice of DATA to one of the segments

Outputs

OUTPUT	Aggregated output tensor. Has the first dimension of K (the number of segments).

Code#

caffe2/operators/segment_reduction_op.cc

SortedSegmentWeightedSumGradient#

No documentation yet.

Code#

caffe2/operators/segment_reduction_op.cc

SpaceToBatch#

SpaceToBatch for 4-D tensors of type T. Zero-pads and then rearranges (permutes) blocks of spatial data into batch. More specifically, this op outputs a copy of the input tensor where values from the height and width dimensions are moved to the batch dimension. After the zero-padding, both height and width of the input must be divisible by the block size.

Code#

caffe2/operators/space_batch_op.cc

SparseLengthsMean#

Pulls in slices of the input tensor, groups them into segments and applies 'Mean' to each segment. Segments are defined by their LENGTHS. This op is basically Gather and LengthsMean fused together. INDICES should contain integers in range 0..N-1 where N is the first dimension of DATA. INDICES represent which slices of DATA need to be pulled in. LENGTHS is a vector that defines slice sizes by first dimention of DATA. Values belonging to the same segment are aggregated together. sum(LENGTHS) has to match INDICES size. The first dimension of the output is equal to the number of input segment, i.e. len(LENGTHS) . Other dimensions are inherited from the input tensor. Mean computes the element-wise mean of the input slices. Operation doesn't change the shape of the individual blocks.

Interface#

Inputs

DATA	Input tensor, slices of which are aggregated.
INDICES	Integer vector containing indices of the first dimension of DATA for the slices that are being aggregated
LENGTHS	Non negative vector with sum of elements equal to INDICES length

Outputs

OUTPUT	Aggregated output tensor. Has the first dimension of K (the number of segments).

Code#

caffe2/operators/segment_reduction_op.cc

SparseLengthsMeanGradient#

No documentation yet.

Code#

caffe2/operators/segment_reduction_op.cc

SparseLengthsSum#

Pulls in slices of the input tensor, groups them into segments and applies ‘Sum’ to each segment. Segments are defined by their LENGTHS. This op is basically Gather and LengthsSum fused together. INDICES should contain integers in range 0..N-1 where N is the first dimension of DATA. INDICES represent which slices of DATA need to be pulled in. LENGTHS is a vector that defines slice sizes by first dimention of DATA. Values belonging to the same segment are aggregated together. sum(LENGTHS) has to match INDICES size. The first dimension of the output is equal to the number of input segment, i.e. len(LENGTHS) . Other dimensions are inherited from the input tensor. Summation is done element-wise across slices of the input tensor and doesn’t change the shape of the individual blocks.

Interface#

Inputs

DATA	Input tensor, slices of which are aggregated.
INDICES	Integer vector containing indices of the first dimension of DATA for the slices that are being aggregated
LENGTHS	Non negative vector with sum of elements equal to INDICES length

Outputs

OUTPUT	Aggregated output tensor. Has the first dimension of K (the number of segments).

Code#

caffe2/operators/segment_reduction_op.cc

SparseLengthsSumGradient#

No documentation yet.

Code#

caffe2/operators/segment_reduction_op.cc

SparseLengthsWeightedSum#

Pulls in slices of the input tensor, groups them into segments and applies ‘WeightedSum’ to each segment. Segments are defined by their LENGTHS. This op is basically Gather and LengthsWeightedSum fused together. INDICES should contain integers in range 0..N-1 where N is the first dimension of DATA. INDICES represent which slices of DATA need to be pulled in. LENGTHS is a vector that defines slice sizes by first dimention of DATA. Values belonging to the same segment are aggregated together. sum(LENGTHS) has to match INDICES size. The first dimension of the output is equal to the number of input segment, i.e. len(LENGTHS) . Other dimensions are inherited from the input tensor. Input slices are first scaled by SCALARS and then summed element-wise. It doesn’t change the shape

of the individual blocks.

Interface#

Arguments

grad_on_weights Produce also gradient for weights. For now it's only supported in Lengths-based operators

Inputs

DATA	Input tensor for the summation
SCALARS	Scalar multipliers for the input slices. Must be a vector with the length matching the first dimension of DATA
INDICES	Integer vector containing indices of the first dimension of DATA for the slices that are being aggregated
LENGTHS	Non negative vector with sum of elements equal to INDICES length

Outputs

OUTPUT	Aggregated output tensor. Has the first dimension of K (the number of segments).

Code#

caffe2/operators/segment_reduction_op.cc

SparseLengthsWeightedSumGradient#

No documentation yet.

Code#

caffe2/operators/segment_reduction_op.cc

SparseLengthsWeightedSumWithMainInputGradient#

No documentation yet.

Code#

caffe2/operators/segment_reduction_op.cc

SparseSortedSegmentMean#

Pulls in slices of the input tensor, groups them into segments and applies 'Mean' to each segment. Segments need to be sorted and contiguous. See also SparseUnsortedSegmentMean that doesn't have this requirement. This op is basically Gather and SortedSegmentMean fused together. INDICES should contain integers in range 0..N-1 where N is the first dimension of DATA. INDICES represent which slices of DATA need to be pulled in. SEGMENT_IDS is a vector that maps each referenced slice of the DATA to a particular group (segment). Values belonging to the same segment are aggregated together. SEGMENT_IDS should have the same dimension as INDICES. The first dimension of the output is equal to the number of input segments, i.e. SEGMENT_IDS[-1]+1 . Other dimensions are inherited from the input tensor. Mean computes the element-wise mean of the input slices. Operation doesn't change the shape of the individual blocks.

Interface#

Inputs

DATA	Input tensor, slices of which are aggregated.
INDICES	Integer vector containing indices of the first dimension of DATA for the slices that are being aggregated
SEGMENT_IDS	Vector with the same length as INDICES and values in the range 0..K-1 and in increasing order that maps each slice of DATA referenced by INDICES to one of the segments

Outputs

OUTPUT	Aggregated output tensor. Has the first dimension of K (the number of segments).

Code#

caffe2/operators/segment_reduction_op.cc

SparseSortedSegmentMeanGradient#

No documentation yet.

Code#

caffe2/operators/segment_reduction_op.cc

SparseSortedSegmentSum#

Pulls in slices of the input tensor, groups them into segments and applies 'Sum' to each segment. Segments need to be sorted and contiguous. See also SparseUnsortedSegmentSum that doesn't have this requirement. This op is basically Gather and SortedSegmentSum fused together. INDICES should contain integers in range 0..N-1 where N is the first dimension of DATA. INDICES represent which slices of DATA need to be pulled in. SEGMENT_IDS is a vector that maps each referenced slice of the DATA to a particular group (segment). Values belonging to the same segment are aggregated together. SEGMENT_IDS should have the same dimension as INDICES. The first dimension of the output is equal to the number of input segments, i.e. SEGMENT_IDS[-1]+1 . Other dimensions are inherited from the input tensor. Summation is done element-wise across slices of the input tensor and doesn't change the shape of the individual blocks.

Interface#

Inputs	
DATA	Input tensor, slices of which are aggregated.
INDICES	Integer vector containing indices of the first dimension of DATA for the slices that are being aggregated
SEGMENT_IDS	Vector with the same length as INDICES and values in the range 0..K-1 and in increasing order that maps each slice of DATA referenced by INDICES to one of the segments
Outputs	
OUTPUT	Aggregated output tensor. Has the first dimension of K (the number of segments).

Code#

caffe2/operators/segment_reduction_op.cc

SparseSortedSegmentSumGradient#

No documentation yet.

Code#

caffe2/operators/segment_reduction_op.cc

SparseSortedSegmentWeightedSum#

Pulls in slices of the input tensor, groups them into segments and applies 'WeightedSum' to each segment. Segments need to be sorted and contiguous. See also SparseUnsortedSegmentWeightedSum that doesn't have this requirement. This op is basically Gather and SortedSegmentWeightedSum fused together. INDICES should contain integers in range 0..N-1 where N is the first dimension of DATA. INDICES represent which slices of DATA need to be pulled in. SEGMENT_IDS is a vector that maps each referenced slice of the DATA to a particular group (segment). Values belonging to the same segment are aggregated together. SEGMENT_IDS should have the same dimension as INDICES. The first dimension of the output is equal to the number of input segments, i.e. SEGMENT_IDS[-1]+1 . Other dimensions are inherited from the input tensor. Input slices are first scaled by SCALARS and then summed element-wise. It doesn't change the shape of the individual blocks.

Interface#

Arguments	
grad_on_weights	Produce also gradient for weights. For now it's only supported in Lengths-based operators
Inputs	
DATA	Input tensor for the summation
SCALARS	Scalar multipliers for the input slices. Must be a vector with the length matching the first dimension of DATA

INDICES	Integer vector containing indices of the first dimension of DATA for the slices that are being aggregated
SEGMENT_IDS	Vector with the same length as INDICES and values in the range 0..K-1 and in increasing order that maps each slice of DATA referenced by INDICES to one of the segments
Outputs	
OUTPUT	Aggregated output tensor. Has the first dimension of K (the number of segments).

Code#

caffe2/operators/segment_reduction_op.cc

SparseSortedSegmentWeightedSumGradient#

No documentation yet.

Code#

caffe2/operators/segment_reduction_op.cc

SparseToDense#

Convert sparse representations to dense with given indices. Transforms a sparse representation of map<id, value> represented as indices vector and values tensor into a compacted tensor where the first dimension is determined by the first dimension of the 3rd input if it is given or the max index. Missing values are filled with zeros. After running this op: output[indices[i], :] = values[i] # output[j, ...] = 0 if j not in indices

Interface#

Inputs	
indices	1-D int32/int64 tensor of concatenated ids of data
values	Data tensor, first dimension has to match indices
data_to_infer_dim	Optional: if provided, the first dimension of output is the first dimension of this tensor.
Outputs	
output	Output tensor of the same type as values of shape [len(lengths), len(mask)] + shape(default_value) (if lengths is not provided the first dimension is omitted)

Code#

caffe2/operators/sparse_to_dense_op.cc

SparseToDenseMask#

Convert sparse representations to dense with given indices. Transforms a sparse representation of map<id, value> represented as indices vector and values tensor into a compacted tensor where the first dimension corresponds to each id provided in mask argument. Missing values are filled with the value of default_value . After running this op: output[j, :] = values[i] # where mask[j] == indices[i] output[j, ...] = default_value # when mask[j] doesn't appear in indices If lengths is provided and not empty, and extra “batch” dimension is prepended to the output. values and default_value can have additional matching dimensions, operation is performed on the entire subtensor in thise case. For example, if lengths is supplied and values is 1-D vector of floats and default_value is a float scalar, the output is going to be a float matrix of size len(lengths) X len(mask)

Interface#

Arguments	
mask	list(int) argument with desired ids on the ‘dense’ output dimension
Inputs	
indices	1-D int32/int64 tensor of concatenated ids of data
values	Data tensor, first dimension has to match indices
default_value	Default value for the output if the id is not present in indices. Must have the same type as values and the same shape, but without the first dimension

lengths	Optional lengths to represent a batch of indices and values.
Outputs	
output	Output tensor of the same type as values of shape [len(lengths), len(mask)] + shape(default_value) (if lengths is not provided the first dimension is omitted)

Code#

caffe2/operators/sparse_to_dense_mask_op.cc

SparseUnsortedSegmentMean#

Pulls in slices of the input tensor, groups them into segments and applies 'Mean' to each segment. Segments ids can appear in arbitrary order (unlike in SparseSortedSegmentMean). This op is basically Gather and UnsortedSegmentMean fused together. INDICES should contain integers in range 0..N-1 where N is the first dimension of DATA. INDICES represent which slices of DATA need to be pulled in. SEGMENT_IDS is a vector that maps each referenced slice of the DATA to a particular group (segment). Values belonging to the same segment are aggregated together. SEGMENT_IDS should have the same dimension as INDICES. If num_segments argument is passed it would be used as a first dimension for the output. Otherwise, it'd be dynamically calculated from as the max value of SEGMENT_IDS plus one. Other output dimensions are inherited from the input tensor. Mean computes the element-wise mean of the input slices. Operation doesn't change the shape of the individual blocks.

Interface#

Inputs	
DATA	Input tensor, slices of which are aggregated.
INDICES	Integer vector containing indices of the first dimension of DATA for the slices that are being aggregated
SEGMENT_IDS	Integer vector with the same length as INDICES that maps each slice of DATA referenced by INDICES to one of the segments
Outputs	
OUTPUT	Aggregated output tensor. Has the first dimension of equal to the number of segments.

Code#

caffe2/operators/segment_reduction_op.cc

SparseUnsortedSegmentMeanGradient#

No documentation yet.

Code#

caffe2/operators/segment_reduction_op.cc

SparseUnsortedSegmentSum#

Pulls in slices of the input tensor, groups them into segments and applies 'Sum' to each segment. Segments ids can appear in arbitrary order (unlike in SparseSortedSegmentSum). This op is basically Gather and UnsortedSegmentSum fused together. INDICES should contain integers in range 0..N-1 where N is the first dimension of DATA. INDICES represent which slices of DATA need to be pulled in. SEGMENT_IDS is a vector that maps each referenced slice of the DATA to a particular group (segment). Values belonging to the same segment are aggregated together. SEGMENT_IDS should have the same dimension as INDICES. If num_segments argument is passed it would be used as a first dimension for the output. Otherwise, it'd be dynamically calculated from as the max value of SEGMENT_IDS plus one. Other output dimensions are inherited from the input tensor. Summation is done element-wise across slices of the input tensor and doesn't change the shape of the individual blocks.

Interface#

Inputs	
DATA	Input tensor, slices of which are aggregated.
INDICES	Integer vector containing indices of the first dimension of DATA for the slices that are being aggregated

SEGMENT_IDS	Integer vector with the same length as INDICES that maps each slice of DATA referenced by INDICES to one of the segments
Outputs	
OUTPUT	Aggregated output tensor. Has the first dimension of equal to the number of segments.

Code#

caffe2/operators/segment_reduction_op.cc

SparseUnsortedSegmentSumGradient#

No documentation yet.

Code#

caffe2/operators/segment_reduction_op.cc

SparseUnsortedSegmentWeightedSum#

Pulls in slices of the input tensor, groups them into segments and applies 'WeightedSum' to each segment. Segments ids can appear in arbitrary order (unlike in SparseSortedSegmentWeightedSum). This op is basically Gather and UnsortedSegmentWeightedSum fused together. INDICES should contain integers in range 0..N-1 where N is the first dimension of DATA. INDICES represent which slices of DATA need to be pulled in. SEGMENT_IDS is a vector that maps each referenced slice of the DATA to a particular group (segment). Values belonging to the same segment are aggregated together. SEGMENT_IDS should have the same dimension as INDICES. If num_segments argument is passed it would be used as a first dimension for the output. Otherwise, it'd be dynamically calculated from as the max value of SEGMENT_IDS plus one. Other output dimensions are inherited from the input tensor. Input slices are first scaled by SCALARS and then summed element-wise. It doesn't change the shape of the individual blocks.

Interface#

Arguments	
grad_on_weights	Produce also gradient for weights. For now it's only supported in Lengths-based operators
Inputs	
DATA	Input tensor for the summation
SCALARS	Scalar multipliers for the input slices. Must be a vector with the length matching the first dimension of DATA
INDICES	Integer vector containing indices of the first dimension of DATA for the slices that are being aggregated
SEGMENT_IDS	Integer vector with the same length as INDICES that maps each slice of DATA referenced by INDICES to one of the segments
Outputs	
OUTPUT	Aggregated output tensor. Has the first dimension of equal to the number of segments.

Code#

caffe2/operators/segment_reduction_op.cc

SparseUnsortedSegmentWeightedSumGradient#

No documentation yet.

Code#

caffe2/operators/segment_reduction_op.cc

SpatialBN#

Carries out spatial batch normalization as described in the paper https://arxiv.org/abs/1502.03167. Depending on the mode it is being run, there are multiple cases for the number of outputs, which we list below: Output case #1: Y, mean, var, saved_mean, saved_var

1 (training mode)

Output case #2: Y (test mode)

Interface#

Arguments

is_test	If set to nonzero, run spatial batch normalization in test mode.
epsilon	The epsilon value to use to avoid division by zero.
order	A StorageOrder string.

Inputs

X	The input 4-dimensional tensor of shape NCHW or NHWC depending on the order parameter.
scale	The scale as a 1-dimensional tensor of size C to be applied to the output.
bias	The bias as a 1-dimensional tensor of size C to be applied to the output.
mean	The running mean (training) or the estimated mean (testing) as a 1-dimensional tensor of size C.
var	The running variance (training) or the estimated variance (testing) as a 1-dimensional tensor of size C.

Outputs

Y	The output 4-dimensional tensor of the same shape as X.
mean	The running mean after the spatial BN operator. Must be in-place with the input mean. Should not be used for testing.
var	The running variance after the spatial BN operator. Must be in-place with the input var. Should not be used for testing.
saved_mean	Saved mean used during training to speed up gradient computation. Should not be used for testing.
saved_var	Saved variance used during training to speed up gradient computation. Should not be used for testing.

Code#

caffe2/operators/spatial_batch_norm_op.cc

SpatialBNGradient#

No documentation yet.

Code#

caffe2/operators/spatial_batch_norm_op.cc

Split#

Split a tensor into a list of tensors.

Interface#

Arguments

axis	Which axis to split on
order	Either NHWC or NCWH, will split on C axis

Code#

caffe2/operators/concat_split_op.cc

SquareRootDivide#

Given DATA tensor with first dimention N and SCALE vector of the same size N produces an output tensor with same dimensions as DATA. Which consists of DATA slices. i-th slice is divided by sqrt(SCALE[i]) elementwise. If SCALE[i] == 0 output slice is identical to the input one (no scaling) Example:

```
1 Data = [
```

```
  [1.0, 2.0],
  [3.0, 4.0]
]

SCALE = [4, 9]

OUTPUT = [
  [2.0, 4.0],
  [9.0, 12.0]
]
```

Code#

caffe2/operators/square_root_divide_op.cc

SquaredL2Distance#

```
Given two input float tensors X, Y, and produces one output float tensor
of the L2 difference between X and Y that is computed as ||(X - Y)^2 / 2||.
```

Interface#

Inputs	
X	1D input tensor
Outputs	
Y	1D input tensor

Code#

caffe2/operators/distance_op.cc

SquaredL2DistanceGradient#

No documentation yet.

Code#

caffe2/operators/distance_op.cc

Squeeze#

Remove single-dimensional entries from the shape of a tensor. Takes a

```
parameter `dims` with a list of dimension to squeeze.
```

If the same blob is provided in input and output, the operation is copy-free. This is the exact inverse operation of ExpandDims given the same dims arg.

Interface#

Inputs	
data	Tensors with at least max(dims) dimensions.
Outputs	
squeezed	Reshaped tensor with same data as input.

Code#

caffe2/operators/utility_ops.cc

StopGradient#

StopGradient is a helper operator that does no actual numerical computation, and in the gradient computation phase stops the gradient from being computed through it.

Code#

caffe2/operators/stop_gradient.cc

StringEndsWith#

Performs the ends-with check on each string in the input tensor. Returns tensor of boolean of the same dimension of input.

Interface#

Arguments	
suffix	The suffix to check input strings against.
Inputs	
strings	Tensor of std::string.
Outputs	
bools	Tensor of bools of same shape as input.

Code#

caffe2/operators/string_ops.cc

StringIndexCreate#

Creates a dictionary that maps string keys to consecutive integers from 1 to max_elements. Zero is reserved for unknown keys.

Interface#

Arguments	
max_elements	Max number of elements, including the zero entry.
Outputs	
handle	Pointer to an Index instance.

Code#

caffe2/operators/index_ops.cc

StringJoin#

Takes a 1-D or a 2-D tensor as input and joins elements in each row with the provided delimieter. Output is a 1-D tensor of size equal to the first dimension of the input. Each element in the output tensor is a string of concatenated elements corresponding to each row in the input tensor. For 1-D input, each element is treated as a row.

Interface#

Arguments	
delimiter	Delimiter for join (Default: “,”).
Inputs	
input	1-D or 2-D tensor
Outputs	
strings	1-D tensor of strings created by joining row elements from the input tensor.

Code#

caffe2/operators/string_ops.cc

StringPrefix#

Computes the element-wise string prefix of the string tensor. Input strings that are shorter than prefix length will be returned unchanged. NOTE: Prefix is computed on number of bytes, which may lead to wrong behavior and potentially invalid strings for variable-length encodings such as utf-8.

Interface#

Arguments	
length	Maximum size of the prefix, in bytes.
Inputs	
strings	Tensor of std::string.
Outputs	
prefixes	Tensor of std::string containing prefixes for each input.

Code#

caffe2/operators/string_ops.cc

StringStartsWith#

Performs the starts-with check on each string in the input tensor. Returns tensor of boolean of the same dimension of input.

Interface#

Arguments	
prefix	The prefix to check input strings against.
Inputs	
strings	Tensor of std::string.
Outputs	
bools	Tensor of bools of same shape as input.

Code#

caffe2/operators/string_ops.cc

StringSuffix#

Computes the element-wise string suffix of the string tensor. Input strings that are shorter than suffix length will be returned unchanged. NOTE: Prefix is computed on number of bytes, which may lead to wrong behavior and potentially invalid strings for variable-length encodings such as utf-8.

Interface#

Arguments	
length	Maximum size of the suffix, in bytes.
Inputs	
strings	Tensor of std::string.
Outputs	
suffixes	Tensor of std::string containing suffixes for each output.

Code#

caffe2/operators/string_ops.cc

Sub#

Performs element-wise binary subtraction (with limited broadcast support). If necessary the right-hand-side argument will be broadcasted to match the shape of left-hand-side argument. When broadcasting is specified, the second tensor can either be of size 1 (a scalar value), or having its shape as a contiguous subset of the first tensor's shape. The starting of the mutually equal shape is specified by the argument "axis", and if it is not set, suffix matching is assumed. 1-dim expansion doesn't work yet. For example, the following tensor shapes are supported (with broadcast=1):

```
1  shape(A) = (2, 3, 4, 5), shape(B) = (,), i.e. B is a scalar
2  shape(A) = (2, 3, 4, 5), shape(B) = (5,)
3  shape(A) = (2, 3, 4, 5), shape(B) = (4, 5)
4  shape(A) = (2, 3, 4, 5), shape(B) = (3, 4), with axis=1
5  shape(A) = (2, 3, 4, 5), shape(B) = (2), with axis=0
6
```

Argument broadcast=1 needs to be passed to enable broadcasting.

Interface#

Arguments	
broadcast	Pass 1 to enable broadcasting
axis	If set, defines the broadcast dimensions. See doc for details.
Inputs	
A	First operand, should share the type with the second operand.
B	Second operand. With broadcasting can be of smaller size than A. If broadcasting is disabled it should be of the same size.
Outputs	
C	Result, has same dimensions and type as A

Code#

caffe2/operators/elementwise_op_schema.cc

Sum#

Element-wise sum of each of the input tensors. The first input tensor can be used in-place as the output tensor, in which case the sum will be done in place and results will be accumulated in input0. All inputs and outputs must have the same shape and data type.

Interface#

Inputs	
data_0	First of the input tensors. Can be inplace.
Outputs	
sum	Output tensor. Same dimension as inputs.

Code#

caffe2/operators/utility_ops.cc

SumInt#

No documentation yet.

Code#

caffe2/operators/utility_ops.cc

Summarize#

Summarize computes four statistics of the input tensor (Tensor)- min, max, mean and standard deviation. The output will be written to a 1-D tensor of size 4 if an output tensor is provided. Else, if the argument 'to_file' is greater than 0, the values are written to a log file in the root folder.

Interface#

Arguments	
to_file	(int, default 0) flag to indicate if the summarized statistics have to be written to a log file.
Inputs	
data	The input data as Tensor.
Outputs	
output	1-D tensor (Tensor) of size 4 containing min, max, mean and standard deviation

Code#

caffe2/operators/summarize_op.cc

TT#

The TT-layer serves as a low-rank decomposition of a fully connected layer. The inputs are the same as to a fully connected layer, but the number of parameters are greatly reduced and forward computation time can be drastically reduced especially for layers with large weight matrices. The multiplication is computed as a product of the input vector with each of the cores that make up the TT layer. Given the input sizes (inp_sizes), output sizes(out_sizes), and the ranks of each of the cores (tt_ranks), the ith core will have size:

```
1  inp_sizes[i] * tt_ranks[i] * tt_ranks[i + 1] * out_sizes[i].
2
```

The complexity of the computation is dictated by the sizes of inp_sizes, out_sizes, and tt_ranks, where there is the trade off between accuracy of the low-rank decomposition and the speed of the computation.

Interface#

Arguments	
inp_sizes	(int[]) Input sizes of cores. Indicates the input size of the individual cores; the size of the input vector X must match the product of the inp_sizes array.
out_sizes	(int[]) Output sizes of cores. Indicates the output size of the individual cores; the size of the output vector Y must match the product of the out_sizes array.
tt_ranks	(int[]) Ranks of cores. Indicates the ranks of the individual cores; lower rank means larger compression, faster computation but reduce accuracy.
Inputs	
X	Input tensor from previous layer with size (M x K), where M is the batch size and K is the input size.
b	1D blob containing the bias vector
cores	1D blob containing each individual cores with sizes specified above.
Outputs	
Y	Output tensor from previous layer with size (M x N), where M is the batch size and N is the output size.

Code#

caffe2/operators/tt_linear_op.cc

Tanh#

Calculates the hyperbolic tangent of the given input tensor element-wise. This operation can be done in an in-place fashion too, by providing the same input and output blobs.

Interface#

Inputs

input	1-D input tensor
Outputs	
output	The hyperbolic tangent values of the input tensor computed element-wise

Code#

caffe2/operators/tanh_op.cc

TanhGradient#

No documentation yet.

Code#

caffe2/operators/tanh_op.cc

TensorProtosDBInput#

TensorProtosDBInput is a simple input operator that basically reads things from a db where each key-value pair stores an index as key, and a TensorProtos object as value. These TensorProtos objects should have the same size, and they will be grouped into batches of the given size. The DB Reader is provided as input to the operator and it returns as many output tensors as the size of the TensorProtos object. Each output will simply be a tensor containing a batch of data with size specified by the 'batch_size' argument containing data from the corresponding index in the TensorProtos objects in the DB.

Interface#

Arguments	
batch_size	(int, default 0) the number of samples in a batch. The default value of 0 means that the operator will attempt to insert the entire data in a single output blob.
Inputs	
data	A pre-initialized DB reader. Typically, this is obtained by calling CreateDB operator with a db_name and a db_type. The resulting output blob is a DB Reader tensor
Outputs	
output	The output tensor in which the batches of data are returned. The number of output tensors is equal to the size of (number of TensorProto's in) the TensorProtos objects stored in the DB as values. Each output tensor will be of size specified by the 'batch_size' argument of the operator

Code#

caffe2/operators/tensor_protos_db_input.cc

TextFileReaderRead#

Read a batch of rows from the given text file reader instance. Expects the number of fields to be equal to the number of outputs. Each output is a 1D tensor containing the values for the given field for each row. When end of file is reached, returns empty tensors.

Interface#

Arguments	
batch_size	Maximum number of rows to read.
Inputs	
handler	Pointer to an existing TextFileReaderInstance.

Code#

caffe2/operators/text_file_reader.cc

Transpose#

Transpose the input tensor similar to numpy.transpose. For example, when axes=(1, 0, 2), given an input tensor of shape (1, 2, 3), the output shape will be (2, 1, 3).

Interface#

Arguments	
axes	A list of integers. By default, reverse the dimensions, otherwise permute the axes according to the values given.
Inputs	
data	An input tensor.
Outputs	
transposed	Transposed output.

Code#

caffe2/operators/transpose_op.cc

UniformFill#

No documentation yet.

Code#

caffe2/operators/filler_op.cc

UniformIntFill#

No documentation yet.

Code#

caffe2/operators/filler_op.cc

Unique#

Deduplicates input indices vector and optionally produces reverse remapping. There's no guarantees on the ordering of the output indices.

Interface#

Inputs	
indices	1D tensor of int32 or int64 indices.
Outputs	
unique_indices	1D tensor of deduped entries.

Code#

caffe2/operators/utility_ops.cc

UniqueUniformFill#

Fill the output tensor with uniform samples between min and max (inclusive). If the second input is given, its elements will be excluded from uniform sampling. Using the second input will require you to provide shape via the first input.

Interface#

Arguments	
min	Minimum value, inclusive
max	Maximum value, inclusive

dtype	The data type for the elements of the output tensor.Strictly must be one of the types from DataType enum in TensorProto.This only supports INT32 and INT64 now. If not set, assume INT32
shape	The shape of the output tensor.Cannot set the shape argument and pass in an input at the same time.
extra_shape	The additional dimensions appended at the end of the shape indicatedby the input blob. Cannot set the extra_shape argument when there is no input blob.
input_as_shape	1D tensor containing the desired output shape
Inputs	
input	Input tensor to provide shape information
avoid	(optional) Avoid elements in this tensor. Elements must be unique.
Outputs	
output	Output tensor of unique uniform samples

Code#

caffe2/operators/filler_op.cc

UnpackSegments#

Map N+1 dim tensor to N dim based on length blob

Interface#

Inputs	
lengths	1-d int/long tensor contains the length in each of the input.
tensor	N+1 dim Tensor.
Outputs	
packed_tensor	N dim Tesor

Code#

caffe2/operators/pack_segments.cc

UnsafeCoalesce#

Coalesce the N inputs into N outputs and a single coalesced output blob. This allows operations that operate over multiple small kernels (e.g. biases in a deep CNN) to be coalesced into a single larger operation, amortizing the kernel launch overhead, synchronization costs for distributed computation, etc. The operator: - computes the total size of the coalesced blob by summing the input sizes - allocates the coalesced output blob as the total size - copies the input vectors into the coalesced blob, at the correct offset.

- aliases each Output(i) to- point into the coalesced blob, at the

1 corresponding offset for Input(i).
2

This is ‘unsafe’ as the output vectors are aliased, so use with caution.

Code#

caffe2/operators/utility_ops.cc

UnsortedSegmentMean#

Applies ‘Mean’ to each segment of input tensor. Segments ids can appear in arbitrary order (unlike in SortedSegmentMean). SEGMENT_IDS is a vector that maps each of the first dimension slices of the DATA to a particular group (segment). Values belonging to the same segment are aggregated together. If num_segments argument is passed it would be used as a first dimension for the output. Otherwise, it’d be dynamically calculated from as the max value of SEGMENT_IDS plus one. Other output dimensions are inherited from the input tensor. Mean computes the element-wise mean of the input slices. Operation doesn’t change the shape of the

individual blocks.

Interface#

Arguments

num_segments Optional int argument specifying the number of output segments and thus the first dimension of the output

Inputs

DATA	Input tensor, slices of which are aggregated.
SEGMENT_IDS	Integer vector with the same length as the first dimension of DATA that maps each slice of DATA to one of the segments

Outputs

OUTPUT	Aggregated output tensor. Has the first dimension of equal to the number of segments.

Code#

caffe2/operators/segment_reduction_op.cc

UnsortedSegmentMeanGradient#

No documentation yet.

Code#

caffe2/operators/segment_reduction_op.cc

UnsortedSegmentSum#

Applies 'Sum' to each segment of input tensor. Segments ids can appear in arbitrary order (unlike in SortedSegmentSum). SEGMENT_IDS is a vector that maps each of the first dimension slices of the DATA to a particular group (segment). Values belonging to the same segment are aggregated together. If num_segments argument is passed it would be used as a first dimension for the output. Otherwise, it'd be dynamically calculated from as the max value of SEGMENT_IDS plus one. Other output dimensions are inherited from the input tensor. Summation is done element-wise across slices of the input tensor and doesn't change the shape of the individual blocks.

Interface#

Arguments

num_segments Optional int argument specifying the number of output segments and thus the first dimension of the output

Inputs

DATA	Input tensor, slices of which are aggregated.
SEGMENT_IDS	Integer vector with the same length as the first dimension of DATA that maps each slice of DATA to one of the segments

Outputs

OUTPUT	Aggregated output tensor. Has the first dimension of equal to the number of segments.

Code#

caffe2/operators/segment_reduction_op.cc

UnsortedSegmentSumGradient#

No documentation yet.

Code#

caffe2/operators/segment_reduction_op.cc

UnsortedSegmentWeightedSum#

Applies ‘WeightedSum’ to each segment of input tensor. Segments ids can appear in arbitrary order (unlike in SortedSegmentWeightedSum). SEGMENT_IDS is a vector that maps each of the first dimension slices of the DATA to a particular group (segment). Values belonging to the same segment are aggregated together. If num_segments argument is passed it would be used as a first dimension for the output. Otherwise, it’d be dynamically calculated from as the max value of SEGMENT_IDS plus one. Other output dimensions are inherited from the input tensor. Input slices are first scaled by SCALARS and then summed element-wise. It doesn’t change the shape of the individual blocks.

Interface#

Arguments	
num_segments	Optional int argument specifying the number of output segments and thus the first dimension of the output
grad_on_weights	Produce also gradient for weights. For now it’s only supported in Lengths-based operators
Inputs	
DATA	Input tensor for the summation
SCALARS	Scalar multipliers for the input slices. Must be a vector with the length matching the first dimension of DATA
SEGMENT_IDS	Integer vector with the same length as the first dimension of DATA that maps each slice of DATA to one of the segments
Outputs	
OUTPUT	Aggregated output tensor. Has the first dimension of equal to the number of segments.

Code#

caffe2/operators/segment_reduction_op.cc

UnsortedSegmentWeightedSumGradient#

No documentation yet.

Code#

caffe2/operators/segment_reduction_op.cc

WallClockTime#

Time since epoch in nanoseconds.

Interface#

Outputs	
time	The time in nanoseconds.

Code#

caffe2/operators/utility_ops.cc

WeightedSum#

Element-wise weighted sum of several data, weight tensor pairs. Input should be in the form X_0, weight_0, X_1, weight_1, ... where X_i all have the same shape, and weight_i are size 1 tensors that specifies the weight of each vector. Note that if one wants to do in-place computation, it could only be done with X_0 also as the output, but not other X_i.

Interface#

Inputs	
weight_0	Weight of the first input in the sum.
Outputs	
output	Result containing weighted elem-wise sum of inputs.

Code#

caffe2/operators/utility_ops.cc

XavierFill#

No documentation yet.

Code#

caffe2/operators/filler_op.cc

Xor#

Performs element-wise logical operation xor (with limited broadcast support). Both input operands should be of type bool . If necessary the right-hand-side argument will be broadcasted to match the shape of left-hand-side argument. When broadcasting is specified, the second tensor can either be of size 1 (a scalar value), or having its shape as a contiguous subset of the first tensor's shape. The starting of the mutually equal shape is specified by the argument "axis", and if it is not set, suffix matching is assumed. 1-dim expansion doesn't work yet. For example, the following tensor shapes are supported (with broadcast=1):

```
1  shape(A) = (2, 3, 4, 5), shape(B) = (,), i.e. B is a scalar
2  shape(A) = (2, 3, 4, 5), shape(B) = (5,)
3  shape(A) = (2, 3, 4, 5), shape(B) = (4, 5)
4  shape(A) = (2, 3, 4, 5), shape(B) = (3, 4), with axis=1
5  shape(A) = (2, 3, 4, 5), shape(B) = (2), with axis=0
6
```

Argument broadcast=1 needs to be passed to enable broadcasting.

Interface#

Arguments	
broadcast	Pass 1 to enable broadcasting
axis	If set, defines the broadcast dimensions. See doc for details.
Inputs	
A	First operand.
B	Second operand. With broadcasting can be of smaller size than A. If broadcasting is disabled it should be of the same size.
Outputs	
C	Result, has same dimensions and A and type bool

Code#

caffe2/operators/elementwise_op_schema.cc

Adagrad#

Computes the AdaGrad update for an input gradient and accumulated history. Concretely, given inputs (param, grad, history, learning_rate), computes

```
1  new_history = history + square(grad)
2  new_grad = learning_rate * grad / (sqrt(new_history) + epsilon)
3  new_param = param + new_grad
```

and returns (new_param, new_history).

Interface#

Arguments	
epsilon	Default 1e-5

Inputs	
param	Parameters to be updated
moment	Moment history
grad	Gradient computed
lr	learning rate
Outputs	
output_param	Updated parameters
output_moment	Updated moment

Code#

caffe2/sgd/adagrad_op.cc

Adam#

Computes the Adam update (https://arxiv.org/abs/1412.6980) for an input gradient and momentum parameters. Concretely, given inputs (param, m1, m2, grad, lr, iters),

```
t = iters + 1
corrected_local_rate = lr * sqrt(1 - power(beta2, t)) /
  (1 - power(beta1, t))
m1_o = (beta1 * m1) + (1 - beta1) * grad
m2_o = (beta2 * m2) + (1 - beta2) * np.square(grad)
grad_o = corrected_local_rate * m1_o / \
    (sqrt(m2_o) + epsilon)
param_o = param + grad_o

```

and returns (param_o, m1_o, m2_o)

Interface#

Arguments	
beta1	Default 0.9
beta2	Default 0.999
epsilon	Default 1e-5
Inputs	
param	Parameters to be updated
moment_1	First moment history
moment_2	Second moment history
grad	Gradient computed
lr	learning rate
iter	iteration number
Outputs	
output_param	Updated parameters
output_moment_1	Updated first moment
output_moment_2	Updated second moment

Code#

caffe2/sgd/adam_op.cc

AtomicIter#

Similar to Iter, but takes a mutex as the first input to make sure that updates are carried out atomically. This can be used in e.g. Hogwild sgd algorithms.

Interface#

Inputs

mutex The mutex used to do atomic increment.

iter The iter counter as an int64_t TensorCPU.

Code#

caffe2/sgd/iter_op.cc

CloseBlobsQueue#

No documentation yet.

Code#

caffe2/queue/queue_ops.cc

CreateBlobsQueue#

No documentation yet.

Code#

caffe2/queue/queue_ops.cc

CreateDB#

No documentation yet.

Code#

caffe2/db/create_db_op.cc

DequeueBlobs#

No documentation yet.

Code#

caffe2/queue/queue_ops.cc

EnqueueBlobs#

No documentation yet.

Code#

caffe2/queue/queue_ops.cc

Ftrl#

No documentation yet.

Code#

caffe2/sgd/ftrl_op.cc

ImageInput#

No documentation yet.

Code#

caffe2/image/image_input_op.cc

Iter#

Stores a singe integer, that gets incremented on each call to Run(). Useful for tracking the iteration count during SGD, for example.

Code#

caffe2/sgd/iter_op.cc

LearningRate#

No documentation yet.

Code#

caffe2/sgd/learning_rate_op.cc

MomentumSGD#

Computes a momentum SGD update for an input gradient and momentum parameters. Concretely, given inputs (grad, m, lr) and parameters (momentum, nesterov), computes:

```
1  if not nesterov:
2      adjusted_gradient = lr * grad + momentum * m
3      return (adjusted_gradient, adjusted_gradient)
4  else:
5      m_new = momentum * m + lr * grad
6      return ((1 + momentum) * m_new - momentum * m, m_new)
7
```

Output is (grad, momentum) Note the difference to MomemtumSGDUpdate, which actually performs the parameter update (and is thus faster).

Code#

caffe2/sgd/momentum_sgd_op.cc

MomentumSGDUpdate#

Performs a momentum SGD update for an input gradient and momentum parameters. Concretely, given inputs (grad, m, lr, param) and parameters (momentum, nesterov), computes:

```
1  if not nesterov:
2      adjusted_gradient = lr * grad + momentum * m
3      param = param - adjusted_gradient
4      return (adjusted_gradient, adjusted_gradient, param)
5  else:
6      m_new = momentum * m + lr * grad
7      param = param - ((1 + momentum) * m_new - momentum * m),
8      return ((1 + momentum) * m_new - momentum * m, m_new, param)
9
```

Output is (grad, momentum, parameter). Note the difference to MomentumSGD, which returns a new gradient but does not perform the parameter update.

Code#

caffe2/sgd/momentum_sgd_op.cc

PackedFC#

Computes the result of passing an input vector X into a fully connected layer with 2D weight matrix W and 1D bias vector b. This is essentially the same as the FC operator but allows one to pack the weight matrix for more efficient inference. See the schema for the FC op for details. Unlike many other operators in Caffe2, this operator is stateful: it assumes that the input weight matrix W never changes, so it is only suitable for inference time when the weight matrix never gets updated by any other ops. Due to performance considerations, this is not checked in non-debug builds.

Code#

caffe2/mkl/operators/packed_fc_op.cc

Python#

No documentation yet.

Code#

caffe2/python/pybind_state.cc

PythonGradient#

No documentation yet.

Code#

caffe2/python/pybind_state.cc

RmsProp#

Computes the RMSProp update (http://www.cs.toronto.edu/ ~tijmen/csc321/slides/lecture_slides_lec6.pdf). Concretely, given inputs (grad, mean_squares, mom, lr), computes:

```
1  mean_squares_o = mean_squares + (1 - decay) * (squaare(grad) - mean_squares)
2  mom_o = momentum * mom + lr * grad / sqrt(epsilon + mean_squares_o)
3  grad_o = mom_o
4
```

returns (grad_o, mean_squares_o, mom_o).

Code#

caffe2/sgd/rmsprop_op.cc

SafeDequeueBlobs#

Dequeue the blobs from queue. When the queue is closed and empty, the output status will be set to true which can be used as exit criteria for execution step. The 1st input is the queue and the last output is the status. The rest are data blobs.

Interface#

Inputs

queue The shared pointer for the BlobsQueue

Code#

caffe2/queue/queue_ops.cc

SafeEnqueueBlobs#

Enqueue the blobs into queue. When the queue is closed and full, the output status will be set to true which can be used as exit criteria for execution step. The 1st input is the queue and the last output is the status. The rest are data blobs.

Interface#

Inputs

queue The shared pointer for the BlobsQueue

Code#

caffe2/queue/queue_ops.cc

SparseAdagrad#

Given inputs (param, history, indices, grad, lr), runs the dense AdaGrad update on (param, grad, history[indices], lr), and returns (new_param, new_history) as in the dense case.

Interface#

Arguments	
epsilon	Default 1e-5
Inputs	
param	Parameters to be updated
moment	Moment history
indices	Sparse indices
grad	Gradient computed
lr	learning rate
Outputs	
output_param	Updated parameters
output_moment_1	Updated moment

Code#

caffe2/sgd/adagrad_op.cc

SparseAdam#

Computes the Adam Update for the sparse case. Given inputs (param, moment1, moment2, indices, grad, lr, iter), runs the dense Adam on on (param, moment1[indices], momemnt2[indices], lr, iter) and returns (new_param, new_moment1, new_moment2) as in dense case

Interface#

Arguments	
beta1	Default 0.9
beta2	Default 0.999
epsilon	Default 1e-5
Inputs	
param	Parameters to be updated
moment_1	First moment history

moment_2	Second moment history
indices	Sparse indices
grad	Gradient computed
lr	learning rate
iter	iteration number
Outputs	
output_param	Updated parameters
output_moment_1	Updated first moment
output_moment_2	Updated second moment

Code#

caffe2/sgd/adam_op.cc

SparseFtrl#

No documentation yet.

Code#

caffe2/sgd/ftrl_op.cc

FCGradient_Decomp#

No documentation yet.

Code#

caffe2/experiments/operators/fully_connected_op_decomposition.cc

---

FCGradient_Prune#

No documentation yet.

Code#

caffe2/experiments/operators/fully_connected_op_prune.cc

FC_Decomp#

No documentation yet.

Code#

caffe2/experiments/operators/fully_connected_op_decomposition.cc

---

FC_Prune#

No documentation yet.

Code#

caffe2/experiments/operators/fully_connected_op_prune.cc

FC_Sparse#

No documentation yet.

Code#

caffe2/experiments/operators/fully_connected_op_sparse.cc

FunHash#

This layer compresses a fully-connected layer for sparse inputs via hashing. It takes four required inputs and an optional fifth input. The first three inputs scalars , indices , and segment_ids are the sparse segmented representation of sparse data, which are the same as the last three inputs of the SparseSortedSegmentWeightedSum operator. If the argument num_segments is specified, it would be used as the first dimension for the output; otherwise it would be derived from the maximum segment ID. The fourth input is a 1D weight vector. Each entry of the fully-connected layer would be randomly mapped from one of the entries in this vector. When the optional fifth input vector is present, each weight of the fully-connected layer would be the linear combination of K entries randomly mapped from the weight vector, provided the input (length-K vector) serves as the coefficients.

Interface#

Arguments	
num_outputs	Number of outputs
num_segments	Number of segments
Inputs	
scalars	Values of the non-zero entries of the sparse data.
indices	Indices to the non-zero valued features.
segment_ids	Segment IDs corresponding to the non-zero entries.
weight	Weight vector
alpha	Optional coefficients for linear combination of hashed weights.
Outputs	
output	Output tensor with the first dimension equal to the number of segments.

Code#

caffe2/experiments/operators/funhash_op.cc

FunHashGradient#

No documentation yet.

Code#

caffe2/experiments/operators/funhash_op.cc

SparseFunHash#

This layer compresses a fully-connected layer for sparse inputs via hashing. It takes four required inputs and an option fifth input. The first three inputs scalars , indices , and segment_ids are the sparse segmented representation of sparse data, which are the same as the last three inputs of the SparseSortedSegmentWeightedSum operator. If the argument num_segments is specified, it would be used as the first dimension for the output; otherwise it would be derived from the maximum segment ID. The fourth input is a 1D weight vector. Each entry of the fully-connected layer would be randomly mapped from one of the entries in this vector. When the optional fifth input vector is present, each weight of the fully-connected layer would be the linear combination of K entries randomly mapped from the weight vector, provided the input (length-K vector) serves as the coefficients.

Interface#

Arguments	
num_outputs	Number of outputs
num_segments	Number of segments
Inputs	
scalars	Values of the non-zero entries of the sparse data.
indices	Indices to the non-zero valued features.

segment_ids	Segment IDs corresponding to the non-zero entries.
weight	Weight vector
alpha	Optional coefficients for linear combination of hashed weights.
Outputs	
output	Output tensor with the first dimension equal to the number of segments.

Code#

caffe2/experiments/operators/sparse_funhash_op.cc

SparseFunHashGradient#

No documentation yet.

Code#

caffe2/experiments/operators/sparse_funhash_op.cc

SparseMatrixReshape#

Compute the indices of the reshaped sparse matrix. It takes two 1D tensors as input: the column indices (in int64) and the row indices (in int), which correspond to INDICES and SEGMENT_IDS in SparseSortedSegment family. It outputs the corresponding reshaped column and row indices. Two arguments are required: an argument old_shape specifies the original shape of the matrix, and new_shape specifies the new shape. One of the dimension in old_shape and new_shape can be -1. The valid combinations are listed below, where p, q, r, s are strictly positive integers. old_shape=(p, q) new_shape=(r, s) old_shape=(p, q) new_shape=(-1, s) old_shape=(p, q) new_shape=(r, -1) old_shape=(-1, q) new_shape=(-1, s) Note that only the first dimension in old_shape can be -1. In that case the second dimension in new_shape must NOT be -1.

Interface#

Arguments	
old_shape	Old shape.
new_shape	New shape.
Inputs	
old_col	Original column indices.
old_row	Original row indices.
Outputs	
new_col	New column indices.
new_row	New row indices.

Code#

caffe2/experiments/operators/sparse_matrix_reshape_op.cc

TTContraction#

Tensor contraction C = A * B

Interface#

Arguments	
K	i_{k-1} * r_k
M	r_{k-1} * o_{k-1}
N	o_k
Inputs	
A	2D matrix of size (K x M)
B	tensor

Outputs	
C	contracted tensor

Code#

caffe2/experiments/operators/tt_contraction_op.cc

TTContractionGradient#

No documentation yet.

Code#

caffe2/experiments/operators/tt_contraction_op.cc

TTPad#

No documentation yet.

Code#

caffe2/experiments/operators/tt_pad_op.cc

TTPadGradient#

No documentation yet.

Code#

caffe2/experiments/operators/tt_pad_op.cc

FC_Dcomp#

No schema documented yet.

ReluFp16#

No schema documented yet.

ReluFp16Gradient#

No schema documented yet.

Snapshot#

No schema documented yet.

SparseLabelToDense#

No schema documented yet.

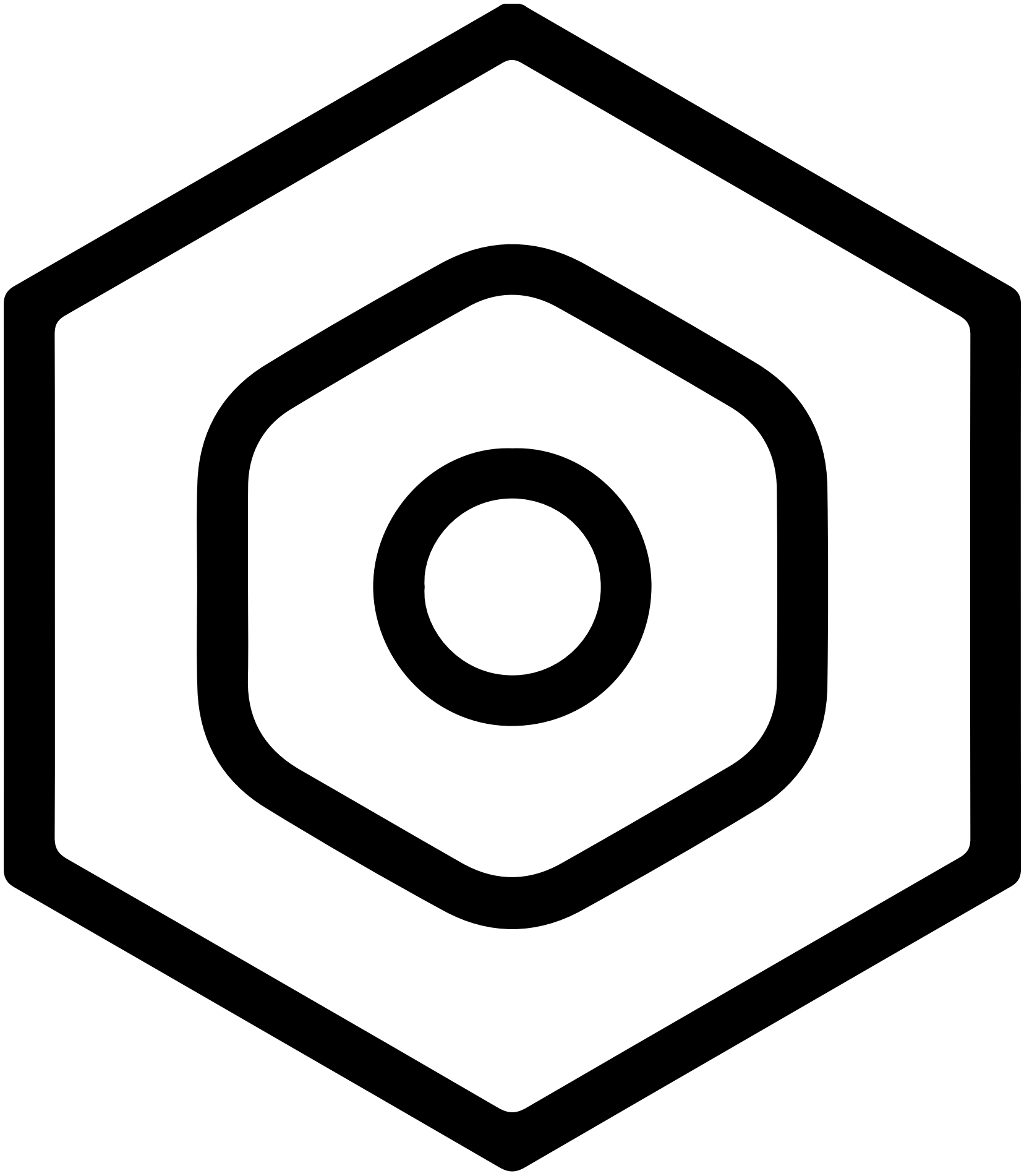
StumpFunc#

No schema documented yet.

TTLinearGradient#

No schema documented yet.

Edit on GitHub



Facebook Open Source

Open Source Projects GitHub Twitter
Contribute to this project on GitHub