DaCeML

Scalable Parallel Computing Laboratory, ETH Zurich, and the DaC

May 24, 2021

USER GUIDE

Machine learning powered by data-centric parallel programming.

This project adds PyTorch and ONNX model loading support to DaCe, and supports ONNX operators as library nodes to DaCe stateful dataflow multigraphs.

CHAPTER

ONE

INSTALLATION

DaCeML can be installed by using `pip install git+https://github.com/spcl/daceml`. It is recommended to install the desired version of PyTorch first.

Alternatively, clone the repository and install using:

```
VENV_PATH='' make install
```

See *Development* for more details on the `Makefile`.

1.1 Installing ONNXRuntime

DaCeML executes ONNX operators using ONNXRuntime by default. To enable this, a patched version[1] of ONNXRuntime needs to be installed and setup.

ONNXRuntime can be installed from source or from a prebuilt release.

1.1.1 Prebuilt Release (CPU only)

The prebuilt release only supports executing ONNX operators with ONNXRuntime on the CPU execution provider.

Note: The CPU-only restriction is not relevant for nodes which have pure SDFG implementations. Nodes with pure implementations can be executed on CPUs, GPUs and FPGAs.

To install, download the prebuilt release and extract it somewhere.

```
wget https://github.com/orausch/onnxruntime/releases/download/v2/onnxruntime-daceml-
↪patched.tgz
tar -xzf onnxruntime-daceml-patched.tgz
```

Afterwards, ensure that the environment variable `ORT_RELEASE` points to the location of the extracted folder.

[1] The patched version will be required until #4453 has been merged and released.

1.1.2 Build from Source

Clone the patched onnxruntime repository somewhere and build it using the following commands.

```
git checkout add-session-state
./build.sh --build_shared_lib --parallel --config Release
```

To enable CUDA, add the relevant arguments. For instance:

```
./build.sh --use_cuda --cuda_version=10.1 --cuda_home=/usr/local/cuda --cudnn_home=/
↪usr/local/cuda --build_shared_lib --parallel --config Release
```

See `onnxruntime/BUILD.md` for more details on building ONNXRuntime.

Afterwards, ensure that the environment variable `ORT_ROOT` points to the root of cloned repository.

CHAPTER

TWO

ONNX

2.1 Library Nodes

This package adds ONNX Operators as library nodes to the SDFG IR.

There exists a ONNX library node for each supported ONNX operator. For example, *ONNXConv* is the ONNX library node for the `Conv` operator.

2.1.1 Operator Parameters (Inputs and Outputs)

The parameters for an operator are specified by adding a connector with the name of the parameter. By default, ops already have connectors for the required parameters, but connectors for optional parameters need to be added manually.

2.1.2 Variadic Parameters

Variadic parameters are specified by adding `__` followed by the index of the variadic parameter. For example, the *ONNXSum* operator has a variadic input named `data_0`. If we wanted to add 3 inputs that connect to this variadic parameter, we would add the connectors: `data_0__0`, `data_0__1` and `data_0__2`. The indices after `__` specify the order of the variadic parameters.

Note: For the variadic parameters to parse correctly, the indices should not have leading zeros. Furthermore, the indices should be sequential without gaps. For example, adding connectors `data_0__0`, `data_0__2` to the *ONNXSum* operator is invalid because the parameter with index 1 is missing.

2.1.3 Attributes

Attributes are set by passing them to the constructor (as python types). For example, the following code sets the stride parameter.

```
from daceml.onnx import ONNXConv
conv = ONNXConv("MyConvNode", strides=[2, 2])
```

The following attribute types are supported:

- `INT` – passed as `int`.
- `INTS` – passed as `List [int]`.
- `STRING` – passed as `str`.

- `STRINGS` – passed as `List [str]`.
- `FLOAT` – passed as `float`.
- `FLOATS` – passed as `List [float]`.
- `TENSOR` – passed as `numpy.ndarray`.

2.1.4 Example

The following examples setup and run an SDFG containing an ONNX Conv operator using *`ONNXConv`*.

This can be done using the python frontend:

```
import dace
import daceml.onnx as donnx
import numpy as np

@dace.program
def conv_program(X_arr: dace.float32[5, 3, 10, 10],
                 W_arr: dace.float32[16, 3, 3, 3]):
    output = dace.define_local([5, 16, 4, 4], dace.float32)
    donnx.ONNXConv(X=X_arr, W=W_arr, Y=output, strides=[2, 2])
    return output

X = np.random.rand(5, 3, 10, 10).astype(np.float32)
W = np.random.rand(16, 3, 3, 3).astype(np.float32)

result = conv_program(X_arr=X, W_arr=W)
```

or the SDFG API:

```
import dace
from daceml.onnx import ONNXConv
import numpy as np

sdfg = dace.SDFG("conv_example")
sdfg.add_array("X_arr", (5, 3, 10, 10), dace.float32)
sdfg.add_array("W_arr", (16, 3, 3, 3), dace.float32)
sdfg.add_array("Z_arr", (5, 16, 8, 8), dace.float32)

state = sdfg.add_state()
access_X = state.add_access("X_arr")
access_W = state.add_access("W_arr")
access_Z = state.add_access("Z_arr")

conv = ONNXConv("MyConvNode")

state.add_node(conv)
state.add_edge(access_X, None, conv, "X", sdfg.make_array_memlet("X_arr"))
state.add_edge(access_W, None, conv, "W", sdfg.make_array_memlet("W_arr"))
state.add_edge(conv, "Y", access_Z, None, sdfg.make_array_memlet("Z_arr"))

X = np.random.rand(5, 3, 10, 10).astype(np.float32)
W = np.random.rand(16, 3, 3, 3).astype(np.float32)
Z = np.zeros((5, 16, 8, 8)).astype(np.float32)

sdfg(X_arr=X, W_arr=W, Z_arr=Z)
```

2.2 Node Implementations

The ONNX library nodes work like library nodes in dace: they can have multiple implementations that can be selected prior to compilation. By default, the nodes use the `onnxruntime` implementation which calls the kernels from ONNXRuntime.

The implementation of a node can be chosen either by specifying the default implementation for the whole ONNX library:

```
import daceml.onnx as donnx
donnx.default_implementation = "pure"
```

Or for a specific node:

```
import daceml.onnx as donnx
donnx.ONNXMatMul.default_implementation = "pure"
```

Note that if an implementation doesn't exist, or cannot be applied, the node expansion will fall back to `onnxruntime`.

2.2.1 Implementation Registration

Implementations for an ONNX node can be registered by implementing the abstract *`ONNXForward`* class. The implementation can be registered using the *`op_implementation()`* decorator. For registration, the parameters `op` and `name` must be passed, where `op` is the name of the ONNX op (without the `ONNX` prefix), and `name` is the name of the implementation.

For example:

```
import daceml.onnx as donnx
from daceml.onnx.op_implementations import op_implementation
from daceml.onnx.forward_implementation_abc import ONNXForward

@op_implementation(op="MatMul", name="my_implementation_name")
class MyMatMul(ONNXForward):
    ...

# can then be used with the library nodes
donnx.ONNXMatMul.default_implementation = "my_implementation_name"
```

Implementations should assume that the node has been validated before it was passed to the expansion. This means that:

- input edges are only connected to valid connectors for the ONNX op
- all ONNX required parameters have been connected
- variadic parameters are correctly passed (correctly formed, and without gaps in the indexing)
- the types of parameters solve correctly according to the type constraints given by the onnx specification
- all required attributes have been set

Implementations can choose to reject certain classes of the node the expand, by implementing the (optional) *`forward_can_be_applied()`* method.

2.2.2 Pure Implementations

Several nodes have an SDFG implementation (i.e. not ONNXRuntime based). The list of all implementations can be found *here*.

2.3 Importing ONNX models

ONNX models can be imported using the *ONNXModel* frontend.

```
import onnx
import os
import numpy as np
from daceml.onnx import ONNXModel

# Download an ONNX model. For example:
# https://github.com/onnx/models/raw/master/vision/classification/efficientnet-lite4/
↪model/efficientnet-lite4-11.onnx
model_path = os.path.join("..", "tests", "onnx_files", "efficientnet.onnx")
model = onnx.load(model_path)
dace_model = ONNXModel("efficientnet", model)

test_input = np.random.rand(1, 3, 224, 224).astype(np.float32)
dace_model(test_input)
```

2.4 Schema Representation & Protobuf conversion

ONNX protobufs are imported and converted to python property classes that can be serialized to and from json by dace (for example *ONNXSchema*). ONNX protobuf instances can be converted to these classes using the `from_onnx_proto` class method that is present on these objects.

These objects are created using *onnx_representation()*. Other ONNX protobuf types can likely be supported in this manner as well. For examples, see the source file `daceml/onnx/schema.py`.

CHAPTER

THREE

PYTORCH INTEGRATION

A PyTorch `nn.Module` can be imported using the *DaceModule* wrapper or *dace_module* decorator.

```
import torch
import torch.nn.functional as F
from daceml.pytorch import DaceModule, dace_module

# Input and size definition
B, H, P, SM, SN = 2, 16, 64, 512, 512
N = P * H
Q, K, V = [torch.randn([SN, B, N]), torch.randn([SM, B, N]), torch.randn([SM, B, N])]

# DaCe module used as a wrapper
ptmodel = torch.nn.MultiheadAttention(N, H, bias=False)
dace_model = DaceModule(ptmodel)
outputs_wrapped = dace_model(Q, K, V)

# DaCe module used as a decorator
@dace_module
class Model(nn.Module):
    def __init__(self, kernel_size):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 4, kernel_size)
        self.conv2 = nn.Conv2d(4, 4, kernel_size)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))

dace_model = Model(3)
outputs_dec = dace_model(torch.rand(1, 1, 8, 8))
```

CHAPTER

FOUR

AUTOMATIC DIFFERENTIATION

Warning: The symbolic automatic differentiation feature still experimental.

DaCeML takes a different approach to automatic differentiation than most deep learning frameworks. Instead of hand-writing backward passes for all differentiable operators, DaceML has a symbolic reverse-mode differentation engine.

4.1 Using Autodiff

There are two main ways to generate backward passes in DaCeML.

DaceModule This class includes a `backward` parameter. If `True`, the autodiff engine will be used to add a backward pass to the PyTorch module, and the resulting module can be seamlessly used with other PyTorch code. For example:

```
import torch.nn.functional as F
from daceml.pytorch import dace_module

@dace_module(backward=True)
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(784, 120)
        self.fc2 = nn.Linear(120, 32)
        self.fc3 = nn.Linear(32, 10)
        self.ls = nn.LogSoftmax(dim=-1)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        x = self.ls(x)
        return x


x = torch.randn(8, 784)
y = torch.tensor([0, 1, 2, 3, 4, 5, 6, 7], dtype=torch.long)

model = Net()

criterion = nn.NLLLoss()
```

(continues on next page)

(continued from previous page)

```
prediction = model(x)
loss = criterion(prediction, y)
print(f"gradients before: {model.model.fc3.weight.grad}")

# gradients can flow through model!
loss.backward()

print(f"gradients after: {model.model.fc3.weight.grad}")
```

add_backward_pass()

The autodiff engine can also be run on plain SDFGs. Here, the output `S` of the dace function/sdfg is differentiated w.r.t to `X` and `Y`.

```
from daceml.autodiff import add_backward_pass

@dace.program
def dace_gemm(
    X: dace.float32[5, 4],
    Y: dace.float32[4, 3],
    Z: dace.float32[5, 3],
    S: dace.float32[1],
):

    Z[:] = X @ Y

    @dace.map(_[0:5, 0:3])
    def summap(i, j):
        s >> S(1, lambda x, y: x + y)[0]
        z << Z[i, j]
        s = z

sdfg = dace_gemm.to_sdfg()

add_backward_pass(sdfg=sdfg, state=sdfg.nodes()[0], inputs=["X", "Y"],␣
↪outputs=["S"])
```

4.2 Architecture

At its core, the automatic differentiation engine attempts to *lift* the SymPy scalar differentiation engine to tensor programs. The SDFG IR is especially suitable for this for two reasons:

- In most SDFGs, computation (i.e. Tasklets) operates on scalars, which can often be differentiated symbolically by SymPy.
- The SDFG IR precisely specifies which Tasklets read and write to which memory locations. This information makes it simple to correctly sum the gradient contribution from each tasklet.

At a high level, it operates as follows:

1. Find the `AccessNode` for each input and output of the `SDFGState`. Use these to determine the subgraph to differentiate.
2. Traverse the subgraph in reverse topological order. For each node:
    - Call a function that *reverses* the node. To reverse the node, the engine checks the *BackwardImplementation* repository for a registered & applicable backward implementation for

that node. If no such function exists and the node is a `LibraryNode`, attempt to differentiate the *pure* expanded version of the node. Otherwise, call the relevant function on `BackwardGenerator`. Main subtleties here are clarified in *Extending the Engine*. Note that this includes a recursive call for `NestedSDFG` nodes (forwarding intermediate values is a source of complexity here).

- Connect required inputs. This includes gradients of outputs of the node, as well as the values of inputs of the node (which potentially need to be routed through reversed maps, or through the hierarchy of `NestedSDFG` s).

4.3 Extending the Engine

When attempting to differentiate a `LibraryNode`, the engine will recursively expand the node until it is in a form that the engine can differentiate. Usually, this means that the engine will expand the node down to the "pure" implementation consisting of simple tasklets and maps.

However, it is sometimes desirable to "exit" this expansion process at a stage earlier than the lowest level. For instance, consider differentiating the *`ONNXMatMul`* library node. Since no backward implementation exists for this node, it will be expanded to its pure version, an *`ONNXEinsum`*. Fully expanding this node into its pure form would result in a mapped tasklet, which we could differentiate. However, we would like to use BLAS nodes on the forward and backward pass where possible. To achieve this, a custom backward implementation is registered for *`ONNXEinsum`*, which returns a `NestedSDFG` containing other einsums. Since we avoid lowering to the lowest level, we are able to preserve information, and can later potentially expand both the forward and backward pass einsums to more efficient BLAS calls.

Another example is *`ONNXSoftmax`*: a typical implementation includes a maximum operation for numerical stablility. Differentiating this implementation results in several argmax calls, which is not desirable.

In situations like these, it makes sense to provide a custom backward pass implementation.

These implementations are registered using *`BackwardImplementation`*. This requires implementation of `backward()`. Examples of this are `daceml.autodiff.implementations.onnx_ops.DefaultEinsumBackward` and `daceml.autodiff.implementations.onnx_ops.DefaultSoftmaxBackward`.

CHAPTER

FIVE

DEVELOPMENT

The `Makefile` contains a few commands for development tasks such as running tests, checking formatting or installing the package.

For example, the following command would install the package and run tests:

```
VENV_PATH='' make install test
```

If you would like to create a virtual environment and install to it, remove *VENV_PATH=''* from the above command.

5.1 Specific Package Versions

The *DACE_VERSION* and *TORCH_VERSION* variables can be used to install specific versions of those packages over the recommended ones. For example, you can use a local dace repository using:

```
DACE_VERSION='-e /path/to/dace/' make clean install
```

5.2 Makefile Targets

The CI runs several tests using the `Makefile`:

`make test`, `make test-parallel` & `make test-gpu` Run pytest on the `tests/` directory. CPU tests can be run in parallel using the `test-parallel` target.

`make doctest` Run doctests; this executes the code samples in the documentation and docstrings.

`make doc` Build the documentation.

`make check-formatting` This runs the formatting checks. The DaCeML codebase is formatted using `yapf`. Use `check-formatting-names` to only print the names of the misformatted files.

5.3 Testing

DaCeML uses `pytest` to run tests. The pytest runner takes a custom argument `--gpu` to run GPU tests. Tests can be parallelized using `xdist` by passing the arguments `-n auto --dist loadfile`.

If you provide the fixture (i.e. an argument to the test) with name `gpu`, then the test will be parameterized to pass both `True` and `False` to that argument.

5.3.1 Setting the default implementation

Nodes can be expanded to different implementations (See *Node Implementations*). To control the default implementation that is used, tests can be decorated with the following two markers.

`pytest.mark.ort` Use the ONNXRuntime expansion as default

`pytest.mark.pure` Use the pure expansion as default when possible (falling back to ONNXRuntime)

If you provide the fixture (i.e., an argument to the test) with name `default_implementation`, then the test will be parameterized to test both implementations.

5.4 Useful Snippets

5.4.1 GPU Leak checker

Put this code in `tests/conf.py`:

```
import xml.etree.ElementTree as ET
import subprocess
import shlex

import torch
# initialize torch cuda context
a = torch.ones(1, 1).cuda()


def _get_gpu_mem_usage():
    result = subprocess.check_output(shlex.split("nvidia-smi -x -q"))
    usage_str = ET.fromstring(result).find("gpu").find("fb_memory_usage").find("used
↪").text
    if not usage_str.endswith(" MiB"):
        raise RuntimeError("Couldn't parse nvidia-smi output")

    return int(usage_str[:-4])

@pytest.fixture(autouse=True)
def memory_printer():
    before = _get_gpu_mem_usage()
    log.debug(f"Usage before: {before}")
    yield
    after = _get_gpu_mem_usage()
    log.debug(f"Usage after: {after}, delta: {after - before}")
    assert after - before < 200
```

CHAPTER

SIX

EXAMPLES

6.1 Using ONNX Library Nodes

This example demonstrates using ONNX library nodes.

The easiest way to use ONNX library nodes is using the dace python frontend

```
import dace
import daceml.onnx as donnx
import numpy as np


@dace.program
def conv_program(X_arr: dace.float32[5, 3, 10, 10], W_arr: dace.float32[16, 3,
                                                                        3, 3]):
    output = np.ndarray([5, 16, 4, 4], dtype=np.float32)
    donnx.ONNXConv(X=X_arr, W=W_arr, Y=output, strides=[2, 2])
    return output
```

The resulting SDFG contains an instance of the *ONNXConv* library node.

```
conv_program.to_sdfg()
```

We can now execute the program with some example inputs

```
X = np.random.rand(5, 3, 10, 10).astype(np.float32)
W = np.random.rand(16, 3, 3, 3).astype(np.float32)

result = conv_program(X_arr=X, W_arr=W)
```

Let's check the correctness vs. PyTorch

```
import torch
import torch.nn.functional as F

torch_result = F.conv2d(torch.from_numpy(X), torch.from_numpy(W),
                        stride=2).numpy()

assert np.allclose(torch_result, result)
np.linalg.norm(torch_result - result)
```

Out:

```
2.3904955e-05
```

We can also use ONNX nodes using the SDFG Python API.

```
from daceml.onnx import ONNXConv

sdfg = dace.SDFG("conv_example")
sdfg.add_array("X_arr", (5, 3, 10, 10), dace.float32)
sdfg.add_array("W_arr", (16, 3, 3, 3), dace.float32)
sdfg.add_array("Z_arr", (5, 16, 4, 4), dace.float32)

state = sdfg.add_state()
access_X = state.add_access("X_arr")
access_W = state.add_access("W_arr")
access_Z = state.add_access("Z_arr")

conv = ONNXConv("MyConvNode", strides=[2, 2])

state.add_node(conv)
state.add_edge(access_X, None, conv, "X", sdfg.make_array_memlet("X_arr"))
state.add_edge(access_W, None, conv, "W", sdfg.make_array_memlet("W_arr"))
state.add_edge(conv, "Y", access_Z, None, sdfg.make_array_memlet("Z_arr"))

sdfg
```

The SDFG looks the same as the one above. Now let's try running it

```
Z = np.zeros((5, 16, 4, 4)).astype(np.float32)
sdfg(X_arr=X, W_arr=W, Z_arr=Z)
assert np.allclose(torch_result, Z)
np.linalg.norm(torch_result - Z)
```

Out:

```
2.3904955e-05
```

Total running time of the script: (0 minutes 7.763 seconds)

6.2 Optimizing the Mish Operator

DaCeML allows users to optimize DNN modules at all levels of granularity, from operators to full models. In this example, we optimize the Mish operator[1], a relatively novel activation function that, among other uses, has been applied successfully in object detection.[2]

Due to its novelty, it has, at the time of writing, not been implemented in PyTorch, ONNX or ONNX Runtime. We demonstrate how DaCeML can be used to optimize this operator.

We begin with code for the PyTorch Module, and import it into DaCeML by annotating it with the `@dace_module` decorator.

```
import torch
from torch import nn
from torch.nn import functional as F
```

(continues on next page)

[1] Diganta Misra. Mish: A self regularized non-monotonic activation function. In 31st British Machine Vision Conference 2020, BMVC 2020, Virtual Event, UK, September 7-10, 2020. BMVA Press, 2020.

[2] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection. CoRR, abs/2004.10934, 2020.

(continued from previous page)

```
from daceml.pytorch import dace_module


@dace_module(cuda=True, backward=True)
class DaCeMish(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, x):
        x = x * (torch.tanh(F.softplus(x)))
        return x
```

The module works immediately with DaCeML for the forward pass.

The first time we tested this, we found that the automatic differentiation failed due to an missing pure implementation for *ONNXSoftplus*. Fortunately, adding these implementations is easily done using the DaCe python frontend. The following code shows the pure implementation that was added.

```
@python_pure_op_implementation
def Softplus(X, Y):
    Y[:] = np.log(1 + np.exp(X))
```

Let's test the operator and compare with a PyTorch version

```
class Mish(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, x):
        x = x * (torch.tanh(F.softplus(x)))
        return x


# create test inputs (size taken from YOLOv4)
with torch.no_grad():
    dace_input = torch.rand(8, 32, 224, 224).cuda()
    torch_input = torch.clone(dace_input)
    dace_dy = torch.rand(8, 32, 224, 224).cuda()
    torch_dy = torch.clone(dace_dy)

dace_input.requires_grad = True
torch_input.requires_grad = True

torch_mish = Mish().cuda()
dace_mish = DaCeMish()

dace_output = dace_mish(dace_input)
dace_output.backward(dace_dy)
torch_output = torch_mish(torch_input)
torch_output.backward(torch_dy)

assert torch.allclose(dace_output, torch_output)
assert torch.allclose(dace_input.grad, torch_input.grad)
```

Let's profile this implementation

```
from daceml.testing.profiling import time_funcs, print_time_statistics


def run_dace():
    out = dace_mish(dace_input)
    out.backward(dace_dy)


def run_torch():
    out = torch_mish(torch_input)
    out.backward(torch_dy)


times = time_funcs([run_dace, run_torch],
                   func_names=["dace", "torch"],
                   warmups=5,
                   num_iters=100)
print_time_statistics(times, ["dace", "torch"])
```

Out:

```
| Name    |     Min |    Mean |   Median |   Stdev |     Max |
|---------|---------|---------|----------|---------|---------|
| dace    | 11.6968 | 12.6615 |  12.2345 |  1.7286 | 26.7186 |
| torch   |  4.7779 |  4.8643 |   4.8738 |  0.0264 |  4.8814 |
```

6.2.1 Inspection

```
# Let's inspect the forward pass SDFG first.
dace_mish.forward_sdfg
```

We can see that there is a lot of unnecessary data movement on the forward pass. Fusing the different maps would greatly improve runtime.

Now let's look at the backward pass.

```
dace_mish.backward_sdfg
```

We also see another opportunity for optimization: The DaCeML autodiff engine is "forwarding" intermediate values to perform the differentiation. This means that the intermediate values have to be written out in the forward pass, and and read in the backward pass.

6.2.2 Optimization

To improve the runtime, we'll apply three transformations.

Firstly, we'll use `SubgraphFusion` to fuse all the maps into a single kernel. To tackle the issue of forwarding intermediate values in backprop, we'll use the `TaskletFusion` transformation. By fusing the tasklets into a single tasklet before running automatic differentiation, the engine will differentiate the whole expression at once, eliminating the need to access the intermediate values. This is an easy way to tune recomputation vs. storage in automatic differentiation.

Finally, we'll apply `Vectorization` to make our kernels operate on more than one element at once.

```
from daceml.transformation import TaskletFusion
from dace.transformation.dataflow import Vectorization, TrivialMapRangeElimination
from dace.transformation.subgraph import SubgraphFusion
from daceml.util import utils
from dace.library import change_default
from daceml import onnx as donnx

# reset the compiled sdfg
dace_mish.reset_sdfg()


# expand the onnx nodes, and apply automatic transformations like inlining
def expand_and_strict_transforms(module):
    # use the pure expansions of operators
    with change_default(donnx, "pure"):
        utils.auto_optimize(module.sdfg, cuda=True, apply_strict=True)


dace_mish.append_post_onnx_hook("auto_optimize", expand_and_strict_transforms)


# apply subgraph fusion
def fuse_sg(module):
    sdfg = module.sdfg
    sdfg.apply_transformations_repeated(TrivialMapRangeElimination)
    SubgraphFusion.apply_to(sdfg, *sdfg.node(0).nodes())


dace_mish.append_post_onnx_hook("subgraph_fusion", fuse_sg)

# apply tasklet fusion
dace_mish.append_post_onnx_hook("fuse_tasklets", lambda x:\
        x.dace_model.sdfg.apply_transformations_repeated(TaskletFusion))


# apply vectorization
def vectorize(fwd, bwd):
    fwd.apply_transformations(Vectorization)
    bwd.apply_transformations(Vectorization)


dace_mish.append_post_autodiff_hook("vectorize", vectorize)
```

Let's check that the new SDFG is still correct.

```
dace_output = dace_mish(dace_input)
dace_output.backward(dace_dy)
torch_output = torch_mish(torch_input)
torch_output.backward(torch_dy)

assert torch.allclose(dace_output, torch_output)
assert torch.allclose(dace_input.grad, torch_input.grad)
```

Out:

```
/opt/actions-runner/_work/daceml/daceml/venv/lib/python3.7/site-packages/dace/sdfg/
↪sdfg.py:1755: UserWarning: SDFG "DaCeMish" is already loaded by another object,␣
↪recompiling under a different name.
```

(continues on next page)

(continued from previous page)

```
  'recompiling under a different name.' % self.name)
/opt/actions-runner/_work/daceml/daceml/venv/lib/python3.7/site-packages/dace/sdfg/
↪sdfg.py:1755: UserWarning: SDFG "DaCeMish_backward" is already loaded by another␣
↪object, recompiling under a different name.
  'recompiling under a different name.' % self.name)
```

After running the module once, we can also inspect the compiled SDFG for the forward and backward pass.

```
dace_mish.forward_sdfg
```

```
dace_mish.backward_sdfg
```

Now we can profile the optimized module.

```
times = time_funcs([run_dace, run_torch],
                   func_names=["dace", "torch"],
                   warmups=5,
                   num_iters=100)
print_time_statistics(times, ["dace", "torch"])
```

Out:

```
| Name    |    Min |   Mean |   Median |   Stdev |    Max |
|---------|--------|--------|----------|---------|--------|
| dace    | 1.5453 | 1.5819 |   1.5854 |  0.0067 | 1.5911 |
| torch   | 4.2580 | 4.3152 |   4.3436 |  0.0398 | 4.3531 |
```

Let's also try PyTorch JIT compilation.

```
import torch.jit

torch_jit = torch.jit.trace(Mish(), torch_input)


def run_torch_jit():
    out = torch_jit(torch_input)
    out.backward(torch_dy)


times = time_funcs([run_dace, run_torch, run_torch_jit],
                   func_names=["dace", "torch", "torch_jit"],
                   warmups=5,
                   num_iters=100)
print_time_statistics(times, ["dace", "torch", "torch_jit"])
```

Out:

```
| Name      |    Min |   Mean |   Median |   Stdev |    Max |
|-----------|--------|--------|----------|---------|--------|
| dace      | 1.5571 | 1.5885 |   1.5888 |  0.0035 | 1.5989 |
| torch     | 4.3418 | 4.3462 |   4.3459 |  0.0015 | 4.3517 |
| torch_jit | 3.8157 | 3.8183 |   3.8183 |  0.0010 | 3.8216 |
```

Total running time of the script: (1 minutes 44.541 seconds)

CHAPTER

SEVEN

DACEML.AUTODIFF

7.1 Generating Backward Passes

add_backward_pass(*sdfg*, *state*, *outputs*, *inputs*)

Experimental: Add a backward pass to *state* using reverse-mode automatic differentiation.

`inputs`, `outputs` and `grads` can be provided either as `AccessNode` nodes, or as `str`, in which case the graph will be searched for exactly one matching `AccessNode` with data matching the `str`.

The SDFG should not contain any inplace operations. It may contain the following nodes:

- Maps
- AccessNodes
- Reductions (Sum, Min, Max)
- ONNXOps
- NestedSDFGs containing a single SDFGState (subject to the same constraints). NestedSDFGs may contain multiple states as long as all other states are only used for zero initialization.

When differentiating an *ONNXOp*, the ONNXBackward registry will be checked for any matching backward pass implementations. If none are found, the ONNXForward registry will be checked for matching pure implementations. If one is found, symbolic differentiation of the pure implementation will be attempted. If this fails, or no pure forward implementation is found, the method will fail.

Parameters

- **sdfg** (`SDFG`) – the parent SDFG of `state`.
- **state** (`SDFGState`) – the state to add the backward pass to. This is also the state of the forward pass.
- **outputs** (`List[Union[AccessNode, str]]`) – the forward pass outputs of the function to differentiate.
- **inputs** (`List[Union[AccessNode, str]]`) – the inputs w.r.t. which the gradient will be returned.

make_backward_function(*model*, *apply_strict=False*)

Convert an ONNXModel to a PyTorch differentiable function. This method should not be used on its own. Instead use the `backward=True` parameter of *daceml.pytorch.DaceModule*.

Parameters

- **model** (`ONNXModel`) – the model to convert.
- **apply_strict** – whether to apply strict transformations before creating the backward pass.

Return type Tuple[SDFG, SDFG, BackwardResult, Dict[str, Data]]

Returns A 4-tuple of forward SDFG, backward SDFG, backward result, and input arrays for backward pass (as mapping of names to DaCe data descriptors).

7.2 Extending Autodiff

class BackwardImplementation

ABC for ONNX op forward implementations.

This registry accepts two types of registrations. The register function expects an argument `node_type=TYPE` where `TYPE` is the type of node that this backward implementation supports. It can also take an argument `op=node_name` where `node_name` is the string of the ONNX op it supports, e.g. `"Conv"`.

It also expects a `name` argument that names the implementation.

abstract static backward(*forward_node*, *context*, *given_gradients*, *required_gradients*)

Add the reverse node for a node from the forward pass to the backward pass, and return it.

For each input connector with name `n` of the forward in required_grads, the returned backward node must add an output connector with name `required_grads[n]` that will output the gradient for that input.

If any input from the forward pass is required, simply add a connector with the same name as the connector on the forward node. The input will later be connected as required.

Parameters

- **forward_node** (Node) – the node for which the backward pass should be generated for.
- **context** (BackwardContext) – the context for this node (see BackwardContext).
- **given_gradients** (List[Optional[str]]) – The names of outputs of the node that gradients will be connected for.
- **required_gradients** (List[Optional[str]]) – The names of connectors that gradients should be generated for.

Return type Tuple[Node, BackwardResult]

Returns the reverse node and gradient names (see BackwardResult).

static backward_can_be_applied(*node*, *state*, *sdfg*)

Return whether this expansion can be applied.

Parameters

- **node** (Node) – the candidate node.
- **state** (SDFGState) – the candidate state.
- **sdfg** (SDFG) – the candidate sdfg.

Return type bool

class BackwardContext(*forward_sdfg*, *forward_state*, *backward_sdfg*, *backward_state*, *backward_generator*)

Bases: object

A tuple holding the graph context required to construct reverse nodes

Parameters

- **forward_sdfg** (*dace.sdfg.sdfg.SDFG*) –
- **forward_state** (*dace.sdfg.state.SDFGState*) –
- **backward_sdfg** (*dace.sdfg.sdfg.SDFG*) –
- **backward_state** (*dace.sdfg.state.SDFGState*) –
- **backward_generator** (*daceml.autodiff.backward_pass_generator.BackwardPassGenerator*) –

Return type None

backward_generator: daceml.autodiff.backward_pass_generator.BackwardPassGenerator
the backward pass generator

backward_sdfg: dace.sdfg.sdfg.SDFG
the backward SDFG

backward_state: dace.sdfg.state.SDFGState
the backward SDFG state

forward_sdfg: dace.sdfg.sdfg.SDFG
the forward SDFG

forward_state: dace.sdfg.state.SDFGState
the forward SDFG state

class BackwardResult(*required_grad_names*, *given_grad_names*)
Bases: `object`

The return type of a differentiated node. It contains the names of the gradients the node calculates and requires.

Parameters

- **required_grad_names** (*Dict[Optional[str], Optional[str]]*) –
- **given_grad_names** (*Dict[Optional[str], Optional[str]]*) –

Return type None

given_grad_names: Dict[Optional[str], Optional[str]]
mapping from names of input connectors to the connector name of the gradient for that connector.

required_grad_names: Dict[Optional[str], Optional[str]]
mapping from names of output connectors to the connector name of the gradient for that connector.

CHAPTER

EIGHT

DACEML.ONNX

has_onnx_node(*name*)
Check if an ONNX operator is supported.

Parameters **name** (str) – the operator name.

Return type bool

get_onnx_node(*name*)
Get the ONNX Operator node for an operator by name.

Parameters **name** (str) – the operator name

Return type *ONNXOp*

class ONNXModel(*name*, *model*, *infer_shapes=True*, *cuda=False*, *apply_strict=False*, *auto_optimize=True*, *fold_constants=True*, *parent_pytorch_module=None*, *storage=None*, *save_transients=None*, *auto_merge=False*)
Loads an ONNX model into an SDFG.

Example First download an ONNX model, such as efficientnet.

```
import onnx
import os
import numpy as np
from daceml.onnx import ONNXModel

model_path = os.path.join("..", "tests", "onnx_files", "efficientnet.
↪onnx")
model = onnx.load(model_path)
dace_model = ONNXModel("efficientnet", model)

test_input = np.random.rand(1, 3, 224, 224).astype(np.float32)
dace_model(test_input)
```

Parameters

- **name** (str) – the name for the SDFG.
- **model** (ModelProto) – the model to import.
- **infer_shapes** (bool) – whether to infer shapes for the model. If this is False, the model must have value infos (with shapes) for all arrays, including intermediate values.
- **cuda** (bool) – if True, the model will be executed on the GPU.
- **apply_strict** (bool) – if True, apply strict transformations after all nodes have been expanded calling (warning: this can be very slow!)
- **auto_optimize** (bool) – if True, apply automatic optimizations before calling.

- **parent_pytorch_module** (Optional[Module]) – when not None, the weight tensors are loaded from the parameters of this model rather than the ONNX graph.
- **storage** (Optional[StorageType]) – the storage type of the parameters, inputs and outputs. If None, will be set according to cuda.
- **save_transients** (Optional[Dict[str, Tensor]]) – if not None, save transients to this dict (for debugging).
- **auto_merge** (bool) – whether to automatically merge symbolic shapes in symbolic shape inference.
- **fold_constants** (*bool*) –

__call__(**args*, ***kwargs*)
Execute the model.

Parameters

- **args** – positional arguments to the model. The i-th argument will be passed as the i-th input of the model.
- **kwargs** – named arguments to the model. The passed names should match the names in the ONNX model.

Return type Union[Tensor, ndarray, Tuple[Union[Tensor, ndarray]]]

Returns the output of the model (or a tuple of outputs if there are multiple).

compile_and_init()
Compile the SDFG and load parameters into GPU memory.

Return type CompiledSDFG

inputs: List[str]
the inputs to the model

outputs: List[str]
the outputs of the model

sdfg: SDFG
the generated SDFG.

state: SDFGState
the state containing the model computation.

weights: Dict[str, torch.Tensor]
mapping from weight name to array

class ONNXOp(**args*, ***kwargs*)
Bases: dace.sdfg.nodes.LibraryNode

Abstract superclass for all ONNX ops. Do not use this class, use the concrete subclasses (e.g. *ONNXConv*) instead.

backward_implementation
Which implementation this library node will expand into in the backward pass.

iter_edges(*state*)
Returns an iterator over tuples of an edge and a boolean that indicates whether that edge is an input, ordered by the order required by the schema. This method assumes that this node has been validated.

Parameters **state** (SDFGState) – the state containing this node.

Return type Iterator[Tuple[MultiConnectorEdge, bool]]

iter_inputs_in_onnx_order(*state*)
Iterate through the output edges in the same order as they would appear in an ONNX node proto. This assumes that the node has been validated!

Parameters **state** (SDFGState) – the state containing this node.

Return type List[MultiConnectorEdge]

Returns the in edges in the order as they would appear in the node proto.

iter_outputs_in_onnx_order(*state*)
Iterate through the input edges in the same order as they would appear in an ONNX node proto. This assumes that the node has been validated!

Parameters **state** (SDFGState) – the state containing this node.

Return type List[MultiConnectorEdge]

Returns the out edges in the order as they would appear in the node proto.

schema
The operator's ONNX OpSchema

validate(*sdfg*, *state*)
Validate this node.

Parameters

- **sdfg** (SDFG) – the parent sdfg.
- **state** (SDFGState) – the parent state.

8.1 Schema Representation

onnx_representation(*represents*, ***mapping*)
Decorator for python representations of ONNX protobufs.

The decorator will monkey patch in the following methods:

- __init__ (a constructor based on the class properties)
- construct_from_onnx_proto
- construct_from_json

Parameters

- **represents** – The onnx protobuf type that the decorated class represents
- **mapping** – a mapping from class property names to either:

  a string s In this case, convert_onnx_attribute will be applied on the protobuf attribute with the name s to get the property value.

  a function f In this case, f will be called with the protobuf, and the property value will be set to the return value of that call.

  If a property name is not present in mapping, the property name itself will be used to access the protobuf attribute.

class ONNXParameterType(*value=<object object>*, *names=None*, *module=None*, *type=None*, *start=1*, *boundary=None*)
An enumeration.

Optional = 2
optional parameters

Single = 1
single/required parameters

Variadic = 3
variadic parameters

class ONNXAttributeType(*value=<object object>*, *names=None*, *module=None*, *type=None*, *start=1*, *boundary=None*)
An enumeration.

Float = 2
Float (python representation is `float`)

Floats = 5
Floats (python representation is `List [float]`)

Int = 1
Integer (python representation is `int`)

Ints = 4
Ints (python representation is `List [int]`)

String = 3
String (python representation is `str`)

Strings = 6
Strings (python representation is `List [str]`)

Tensor = 7
Tensor (python representation is `numpy.ndarray`)

Unsupported = 8
Any unsupported attribute type

class ONNXAttribute(*name=''*, *description=''*, *required=False*, *attribute_type=<ONNXAttributeType.Int: 1>*, *default_value=None*)
Python representation of an ONNX attribute.

Parameters

- **name** (`str`, default `''`) – The attribute name
- **description** (`str`, default `''`) – A description this attribute
- **required** (`bool`, default `False`) – Whether this attribute is required
- **attribute_type** (`ONNXAttributeType`, default `<ONNXAttributeType.Int: 1>`) – The type of this attribute
- **default_value** (`NoneType`, default `None`) – The default value of this attribute

attribute_type
The type of this attribute

default_value
The default value of this attribute

description
A description this attribute

classmethod from_json(*json*, *context=None*)
Construct an object json

classmethod from_onnx_proto(*onnx_proto*)
Construct an object from an ONNX proto of type `<class 'onnx.onnx_cpp2py_export.defs.OpSchema.Attribute'>`.

name
The attribute name

required
Whether this attribute is required

to_json()
Serialize to json

class ONNXTypeConstraint(*type_str=''*, *types=[]*)
Python representation of an ONNX type constraint.

Parameters

- **type_str** (`str`, default `''`) – The type parameter string
- **types** (`list`, default `[]`) – The possible types. Note that only tensor types are currently supported.

classmethod from_json(*json*, *context=None*)
Construct an object json

classmethod from_onnx_proto(*onnx_proto*)
Construct an object from an ONNX proto of type `<class 'onnx.onnx_cpp2py_export.defs.OpSchema.TypeConstraintParam'>`.

to_json()
Serialize to json

type_str
The type parameter string

types
The possible types. Note that only tensor types are currently supported.

class ONNXParameter(*name=''*, *description=''*, *type_str=''*, *param_type=<ONNXParameterType.Single: 1>*, *homogeneous=False*)
Python representation of an ONNX parameter.

Parameters

- **name** (`str`, default `''`) – The parameter name
- **description** (`str`, default `''`) – A description of the parameter
- **type_str** (`str`, default `''`) – The type string of this parameter
- **param_type** (`ONNXParameterType`, default `<ONNXParameterType.Single: 1>`) – The type of the this parameter
- **homogeneous** (`bool`, default `False`) – Whether this parameter is homogeneous

description
A description of the parameter

classmethod from_json(*json*, *context=None*)
Construct an object json

classmethod from_onnx_proto(*onnx_proto*)
Construct an object from an ONNX proto of type `<class 'onnx.onnx_cpp2py_export.defs.OpSchema.FormalParameter'>`.

homogeneous
Whether this parameter is homogeneous

name
The parameter name

param_type
The type of the this parameter

to_json()
Serialize to json

type_str
The type string of this parameter

class ONNXSchema(*name=''*, *domain=''*, *doc=''*, *since_version=0*, *attributes={}*, *type_constraints={}*, *inputs=[]*, *outputs=[]*)
Python representation of an ONNX schema

Parameters

- **name** (`str`, default `''`) – The operator name
- **domain** (`str`, default `''`) – The operator domain
- **doc** (`str`, default `''`) – The operator's docstring
- **since_version** (`int`, default `0`) – The version of the operator
- **attributes** (`dict`, default `{}`) – The operator attributes. Keys should contain the name of the attribute, and values should have type *ONNXAttribute*.
- **type_constraints** (`dict`, default `{}`) – The type constraints for inputs and outputs. Keys should contain the type string of the constraint, values should have type *ONNXTypeConstraint*.
- **inputs** (`list`, default `[]`) – The operator input parameter descriptors. Entries should have type *ONNXParameter*.
- **outputs** (`list`, default `[]`) – The operator output parameter descriptors. Entries should have type *ONNXParameter*.

attributes
The operator attributes. Keys should contain the name of the attribute, and values should have type *ONNXAttribute*.

doc
The operator's docstring

domain
The operator domain

classmethod from_json(*json*, *context=None*)
Construct an object json

classmethod from_onnx_proto(*onnx_proto*)
Construct an object from an ONNX proto of type `<class 'onnx.onnx_cpp2py_export.defs.OpSchema'>`.

inputs
The operator input parameter descriptors. Entries should have type *ONNXParameter*.

name
The operator name

outputs
The operator output parameter descriptors. Entries should have type *ONNXParameter*.

since_version
The version of the operator

to_json()
Serialize to json

type_constraints
The type constraints for inputs and outputs. Keys should contain the type string of the constraint, values should have type *ONNXTypeConstraint*.

8.2 Op Implementation Registration

class ONNXForward
ABC for ONNX op forward implementations.

The register function expects an argument *op* containing the ONNX op name (string).

abstract static forward(*node*, *state*, *sdfg*)
Expand *node* and return its expansion.

Parameters

- **node** (*ONNXOp*) – the candidate node.
- **state** (SDFGState) – the candidate state.
- **sdfg** (SDFG) – the candidate sdfg.

Return type Union[Node, SDFG]

Returns the expanded node.

static forward_can_be_applied(*node*, *state*, *sdfg*)
Return whether this expansion can be applied.

Parameters

- **node** (*ONNXOp*) – the candidate node.
- **state** (SDFGState) – the candidate state.
- **sdfg** (SDFG) – the candidate sdfg.

Return type bool

Returns whether this expansion can be applied.

op_implementation(*op*, *name*)
A decorator that registers an op implementation. It should be used on classes that extend *ONNXForward*.

Parameters

- **op** – the ONNX name of the op to register for.
- **name** – the name of the implementation.

8.3 SDFG-based ONNX Implementations

class PureCast
Bases: *daceml.onnx.forward_implementation_abc.ONNXForward*

Implementation name "pure"

class PureEinsum
Bases: *daceml.onnx.forward_implementation_abc.ONNXForward*

Implementation name "pure"

class PureErf
Bases: *daceml.onnx.forward_implementation_abc.ONNXForward*

Implementation name "pure"

class PureFlatten
Bases: *daceml.onnx.forward_implementation_abc.ONNXForward*

Flatten Expansion, reuses Reshape implementation

Implementation name "pure"

class PureGemm
Bases: *daceml.onnx.forward_implementation_abc.ONNXForward*

Implementation name "pure"

class PureLog
Bases: *daceml.onnx.forward_implementation_abc.ONNXForward*

Implementation name "pure"

class PureLogSoftmax
Bases: *daceml.onnx.forward_implementation_abc.ONNXForward*

Implementation name "pure"

class PureMatMul
Bases: *daceml.onnx.forward_implementation_abc.ONNXForward*

Implementation name "pure"

class PurePow
Bases: *daceml.onnx.forward_implementation_abc.ONNXForward*

Implementation name "pure"

class PureReciprocal
Bases: *daceml.onnx.forward_implementation_abc.ONNXForward*

Implementation name "pure"

class PureReduceMax
Bases: *daceml.onnx.forward_implementation_abc.ONNXForward*

Implementation name "pure"

class PureReduceMean
Bases: *daceml.onnx.forward_implementation_abc.ONNXForward*

Implementation name "pure"

class PureReduceMin
Bases: *daceml.onnx.forward_implementation_abc.ONNXForward*

Implementation name `"pure"`

class PureReduceSum
Bases: *daceml.onnx.forward_implementation_abc.ONNXForward*

Implementation name `"pure"`

class PureRelu
Bases: *daceml.onnx.forward_implementation_abc.ONNXForward*

Implementation name `"pure"`

class PureReshape
Bases: *daceml.onnx.forward_implementation_abc.ONNXForward*

Reshape expansion: this relies on views

Implementation name `"pure"`

class PureSigmoid
Bases: *daceml.onnx.forward_implementation_abc.ONNXForward*

Implementation name `"pure"`

class PureSoftmax
Bases: *daceml.onnx.forward_implementation_abc.ONNXForward*

Implementation name `"pure"`

class PureSqrt
Bases: *daceml.onnx.forward_implementation_abc.ONNXForward*

Implementation name `"pure"`

class PureSum
Bases: *daceml.onnx.forward_implementation_abc.ONNXForward*

Implementation name `"pure"`

class PureTranspose
Bases: *daceml.onnx.forward_implementation_abc.ONNXForward*

Implementation name `"pure"`

8.3.1 Image-related Implementations

class PureBatchNormalization
Bases: *daceml.onnx.forward_implementation_abc.ONNXForward*

Implementation name `"pure"`

class PureConv2D
Bases: *daceml.onnx.forward_implementation_abc.ONNXForward*

The “trivial” convolution implementation, i.e. two nested maps.

Implementation name `"pure"`

class PureMaxPool2D
Bases: *daceml.onnx.forward_implementation_abc.ONNXForward*

Implementation name `"pure"`

8.4 Dace CMake Environments

class ONNXRuntime(**args*, ***kwargs*)
Environment used to run ONNX operator nodes using ONNX Runtime. See *Installing ONNXRuntime* for installation instructions.

class ONNXRuntimeCUDA(**args*, ***kwargs*)
Environment used to run ONNX operator nodes using ONNX Runtime, with the CUDA execution provider. See *Installing ONNXRuntime* for installation instructions.

8.5 Supported ONNX Operators

The following documentation is mostly automatically generated from the ONNX documentation, except for the removal of unsupported attributes and nodes.

class ONNXAbs(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **X** (T, single) – Input tensor

Node Outputs

- **Y** (T, single) – Output tensor

Type Constraints

- **T** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`

Parameters **name** – the name of the node.

class ONNXAcos(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **input** (T, single) – Input tensor

Node Outputs

- **output** (T, single) – The arccosine of the input tensor computed element-wise

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters **name** – the name of the node.

class ONNXAcosh(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **input** (T, single) – Input tensor

Node Outputs

- **output** (T, single) – The hyperbolic arccosine values of the input tensor computed element-wise

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters **`name`** – the name of the node.

`class ONNXAdagrad`(*name*, ***, *decay_factor=0.0*, *epsilon=9.999999974752427e-07*, *norm_coefficient=0.0*)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Compute one iteration of ADAGRAD, a stochastic gradient based optimization algorithm. This operator can conduct the optimization of multiple tensor variables.

Let's define the behavior of this operator. As you can imagine, ADAGRAD requires some parameters:

- The initial learning-rate "R".
- The update count "T". That is, the number of training iterations conducted.
- A L2-norm regularization coefficient "norm_coefficient".
- A learning-rate decay factor "decay_factor".
- A small constant "epsilon" to avoid dividing-by-zero.

At each ADAGRAD iteration, the optimized tensors are moved along a direction computed based on their estimated gradient and accumulated squared gradient. Assume that only a single tensor "X" is updated by this operator. We need the value of "X", its gradient "G", and its accumulated squared gradient "H". Therefore, variables in this operator's input list are sequentially "R", "T", "X", "G", and "H". Other parameters are given as attributes because they are usually constants. Also, the corresponding output tensors are the new value of "X" (called "X_new"), and then the new accumulated squared gradient (called "H_new"). Those outputs are computed from the given inputs following the pseudo code below.

Let "+", "-", "*", and "/" are all element-wise arithmetic operations with numpy-style broadcasting support. The pseudo code to compute those outputs is:

// Compute a scalar learning-rate factor. At the first update of X, T is generally // 0 (0-based update index) or 1 (1-based update index). r = R / (1 + T * decay_factor);

// Add gradient of 0.5 * norm_coefficient * ||X||_2^2, where ||X||_2 is the 2-norm. G_regularized = norm_coefficient * X + G;

// Compute new accumulated squared gradient. H_new = H + G_regularized * G_regularized;

// Compute the adaptive part of per-coordinate learning rate. Note that Sqrt(...) // computes element-wise square-root. H_adaptive = Sqrt(H_new) + epsilon

// Compute the new value of "X". X_new = X - r * G_regularized / H_adaptive;

If one assign this operators to optimize multiple inputs, for example, "X_1" and "X_2", the same pseudo code may be extended to handle all tensors jointly. More specifically, we can view "X" as a concatenation of "X_1" and "X_2" (of course, their gradient and accumulate gradient should be concatenated too) and then just reuse the entire pseudo code.

Note that ADAGRAD was first proposed in http://jmlr.org/papers/volume12/duchi11a/duchi11a.pdf. In that reference paper, this operator is a special case of the Figure 1's composite mirror descent update.

Node Inputs

- **R** (T1, single) – The initial learning rate.
- **T** (T2, single) – The update count of "X". It should be a scalar.
- **inputs** (T3, variadic) – The current values of optimized tensors, followed by their respective gradients, followed by their respective accumulated squared gradients.For example, if two tensor "X_1" and "X_2" are optimized, The input list would be ["X_1", "X_2", gradient of "X_1", gradient of "X_2", accumulated squared gradient of "X_1", accumulated squared gradient of "X_2"].

Node Outputs

- **outputs** (T3, variadic) – Updated values of optimized tensors, followed by their updated values of accumulated squared gradients. For example, if two tensor "X_1" and "X_2" are optimized, the output list would be [new value of "X_1," new value of "X_2" new accumulated squared gradient of "X_1", new accumulated squared gradient of "X_2"].

Type Constraints

- **T1** – `float32`, `float64`
- **T2** – `int64`
- **T3** – `float32`, `float64`

Parameters

- **`name`** – the name of the node.
- **`decay_factor`** (`Optional [float]`, default=0.0) – The decay factor of learning rate after one update.The effective learning rate is computed by r = R / (1 + T * decay_factor). Default to 0 so that increasing update counts doesn't reduce the learning rate.
- **`epsilon`** (`Optional [float]`, default=9.999999974752427e-07) – Small scalar to avoid dividing by zero.
- **`norm_coefficient`** (`Optional [float]`, default=0.0) – Regularization coefficient in 0.5 * norm_coefficient * ||X||_2^2. Default to 0, which means no regularization.

`decay_factor`
The decay factor of learning rate after one update.The effective learning rate is computed by r = R / (1 + T * decay_factor). Default to 0 so that increasing update counts doesn't reduce the learning rate.

`epsilon`
Small scalar to avoid dividing by zero.

`norm_coefficient`
Regularization coefficient in 0.5 * norm_coefficient * ||X||_2^2. Default to 0, which means no regularization.

class ONNXAdam(*name*, ***, *alpha=0.8999999761581421*, *beta=0.9990000128746033*, *epsilon=9.999999974752427e-07*, *norm_coefficient=0.0*, *norm_coefficient_post=0.0*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Compute one iteration of Adam, a stochastic gradient based optimization algorithm. This operator can conduct the optimization of multiple tensor variables.

Let's define the behavior of this operator. First of all, Adam requires some parameters:

- The learning-rate "R".
- The update count "T". That is, the number of training iterations conducted.

- A L2-norm regularization coefficient “norm_coefficient”.
- A small constant “epsilon” to avoid dividing-by-zero.
- Two coefficients, “alpha” and “beta”.

At each Adam iteration, the optimized tensors are moved along a direction computed based on their exponentially-averaged historical gradient and exponentially-averaged historical squared gradient. Assume that only a tensor “X” is being optimized. The rest of required information is

- the value of “X”,
- “X“’s gradient (denoted by “G”),
- “X“’s exponentially-averaged historical gradient (denoted by “V”), and
- “X“’s exponentially-averaged historical squared gradient (denoted by “H”).

Some of those parameters are passed into this operator as input tensors and others are stored as this operator’s attributes. Specifically, this operator’s input tensor list is [“R”, “T”, “X”, “G”, “V”, “H”]. That is, “R” is the first input, “T” is the second input, and so on. Other parameters are given as attributes because they are constants. Moreover, the corresponding output tensors are

- the new value of “X” (called “X_new”),
- the new exponentially-averaged historical gradient (denoted by “V_new”), and
- the new exponentially-averaged historical squared gradient (denoted by “H_new”).

Those outputs are computed following the pseudo code below.

Let “+”, “-“, “*”, and “/” are all element-wise arithmetic operations with numpy-style broadcasting support. The pseudo code to compute those outputs is:

> // Add gradient of 0.5 * norm_coefficient * ||X||_2^2, where ||X||_2 is the 2-norm. G_regularized = norm_coefficient * X + G
>
> // Update exponentially-averaged historical gradient. V_new = alpha * V + (1 - alpha) * G_regularized
>
> // Update exponentially-averaged historical squared gradient. H_new = beta * H + (1 - beta) * G_regularized * G_regularized
>
> // Compute the element-wise square-root of H_new. V_new will be element-wisely // divided by H_sqrt for a better update direction. H_sqrt = Sqrt(H_new) + epsilon
>
> // Compute learning-rate. Note that “alpha**T”/”beta**T” is alpha’s/beta’s T-th power. R_adjusted = T > 0 ? R * Sqrt(1 - beta**T) / (1 - alpha**T) : R
>
> // Compute new value of “X”. X_new = X - R_adjusted * V_new / H_sqrt
>
> // Post-update regularization. X_final = (1 - norm_coefficient_post) * X_new

If there are multiple inputs to be optimized, the pseudo code will be applied independently to each of them.

Node Inputs

- **R** (T1, single) – The initial learning rate.
- **T** (T2, single) – The update count of “X”. It should be a scalar.
- **inputs** (T3, variadic) – The tensors to be optimized, followed by their respective gradients, followed by their respective accumulated gradients (aka momentum), followed by their respective accumulated squared gradients. For example, to optimize tensors “X_1” and “X_2,”, the input list would be [“X_1”, “X_2”, gradient of “X_1”, gradient of “X_2”,

accumulated gradient of "X_1", accumulated gradient of "X_2", accumulated squared gradient of "X_1", accumulated squared gradient of "X_2"].

Node Outputs

- **outputs** (T3, variadic) – New values of optimized tensors, followed by their respective new accumulated gradients, followed by their respective new accumulated squared gradients. For example, if two tensors "X_1" and "X_2" are optimized, the outputs list would be [new value of "X_1", new value of "X_2", new accumulated gradient of "X_1", new accumulated gradient of "X_2", new accumulated squared gradient of "X_1", new accumulated squared gradient of "X_2"].

Type Constraints

- **T1** – `float32`, `float64`
- **T2** – `int64`
- **T3** – `float32`, `float64`

Parameters

- **`name`** – the name of the node.
- **`alpha`** (`Optional [float]`, default=0.8999999761581421) – Coefficient of previously accumulated gradient in running average. Default to 0.9.
- **`beta`** (`Optional [float]`, default=0.9990000128746033) – Coefficient of previously accumulated squared-gradient in running average. Default to 0.999.
- **`epsilon`** (`Optional [float]`, default=9.999999974752427e-07) – Small scalar to avoid dividing by zero.
- **`norm_coefficient`** (`Optional [float]`, default=0.0) – Regularization coefficient of 0.5 * norm_coefficient * ||X||_2^2. Default to 0, which means no regularization.
- **`norm_coefficient_post`** (`Optional [float]`, default=0.0) – Regularization coefficient of 0.5 * norm_coefficient * ||X||_2^2. Default to 0, which means no regularization.

`alpha`
Coefficient of previously accumulated gradient in running average. Default to 0.9.

`beta`
Coefficient of previously accumulated squared-gradient in running average. Default to 0.999.

`epsilon`
Small scalar to avoid dividing by zero.

`norm_coefficient`
Regularization coefficient of 0.5 * norm_coefficient * ||X||_2^2. Default to 0, which means no regularization.

`norm_coefficient_post`
Regularization coefficient of 0.5 * norm_coefficient * ||X||_2^2. Default to 0, which means no regularization.

class ONNXAdd(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

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

Node Inputs

- **A** (T, single) – First operand.
- **B** (T, single) – Second operand.

Node Outputs

- **C** (T, single) – Result, has same element type as two inputs

Type Constraints

- **T** – `uint32`, `uint64`, `int32`, `int64`, `float16`, `float32`, `float64`

Parameters **`name`** – the name of the node.

`class ONNXAnd`(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

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

Node Inputs

- **A** (T, single) – First input operand for the logical operator.
- **B** (T, single) – Second input operand for the logical operator.

Node Outputs

- **C** (T1, single) – Result tensor.

Type Constraints

- **T** – `bool_`
- **T1** – `bool_`

Parameters **`name`** – the name of the node.

`class ONNXArgMax`(*name*, *, *axis=0*, *keepdims=1*, *select_last_index=0*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **data** (T, single) – An input tensor.

Node Outputs

- **reduced** (reduced_constraint, single) – Reduced output tensor with integer data type.

Type Constraints

- **T** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`
- **reduced_constraint** – `int64`

Parameters

- **name** – the name of the node.
- **axis** (Optional [int], default=0) – The axis in which to compute the arg indices. Accepted range is [-r, r-1] where r = rank(data).
- **keepdims** (Optional [int], default=1) – Keep the reduced dimension or not, default 1 mean keep reduced dimension.
- **select_last_index** (Optional [int], default=0) – Whether to select the last index or the first index if the {name} appears in multiple indices, default is False (first index).

axis
The axis in which to compute the arg indices. Accepted range is [-r, r-1] where r = rank(data).

keepdims
Keep the reduced dimension or not, default 1 mean keep reduced dimension.

select_last_index
Whether to select the last index or the first index if the {name} appears in multiple indices, default is False (first index).

class ONNXArgMin(*name*, *, *axis=0*, *keepdims=1*, *select_last_index=0*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **data** (T, single) – An input tensor.

Node Outputs

- **reduced** (reduced_constraint, single) – Reduced output tensor with integer data type.

Type Constraints

- **T** – uint8, uint16, uint32, uint64, int8, int16, int32, int64, float16, float32, float64
- **reduced_constraint** – int64

Parameters

- **name** – the name of the node.
- **axis** (Optional [int], default=0) – The axis in which to compute the arg indices. Accepted range is [-r, r-1] where r = rank(data).
- **keepdims** (Optional [int], default=1) – Keep the reduced dimension or not, default 1 mean keep reduced dimension.
- **select_last_index** (Optional [int], default=0) – Whether to select the last index or the first index if the {name} appears in multiple indices, default is False (first index).

axis
The axis in which to compute the arg indices. Accepted range is [-r, r-1] where r = rank(data).

keepdims
Keep the reduced dimension or not, default 1 mean keep reduced dimension.

select_last_index
Whether to select the last index or the first index if the {name} appears in multiple indices, default is False (first index).

class ONNXArrayFeatureExtractor(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Select elements of the input tensor based on the indices passed.
 The indices are applied to the last axes of the tensor.

Node Inputs

- **X** (T, single) – Data to be selected
- **Y** (Y_constraint, single) – The indices, based on 0 as the first index of any dimension.

Node Outputs

- **Z** (T, single) – Selected output data as an array

Type Constraints

- **T** – `float32`, `float64`, `int64`, `int32`
- **Y_constraint** – `int64`

Parameters **name** – the name of the node.

class ONNXAsin(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **input** (T, single) – Input tensor

Node Outputs

- **output** (T, single) – The arcsine of the input tensor computed element-wise

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters **name** – the name of the node.

class ONNXAsinh(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **input** (T, single) – Input tensor

Node Outputs

- **output** (T, single) – The hyperbolic arcsine values of the input tensor computed element-wise

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters **name** – the name of the node.

class ONNXAtan(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **input** (T, single) – Input tensor

Node Outputs

- **output** (T, single) – The arctangent of the input tensor computed element-wise

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters **name** – the name of the node.

class ONNXAtanh(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **input** (T, single) – Input tensor

Node Outputs

- **output** (T, single) – The hyperbolic arctangent values of the input tensor computed element-wise

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters **name** – the name of the node.

class ONNXAveragePool(*name*, *, *auto_pad='NOTSET'*, *ceil_mode=0*, *count_include_pad=0*, *kernel_shape*, *pads=None*, *strides=None*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

AveragePool consumes an input tensor X and applies average pooling across the tensor according to kernel sizes, stride sizes, and pad lengths. average pooling consisting of computing the average on all values of a subset of the input tensor according to the kernel size and downsampling the data into the output tensor Y for further processing. The output spatial shape will be following: ` output_spatial_shape[i] = floor((input_spatial_shape[i] + pad_shape[i] - kernel_spatial_shape[i]) / strides_spatial_shape[i] + 1) ` or ` output_spatial_shape[i] = ceil((input_spatial_shape[i] + pad_shape[i] - kernel_spatial_shape[i]) / strides_spatial_shape[i] + 1) ` if ceil_mode is enabled

` * pad_shape[i] is sum of pads along axis i `

auto_pad is a DEPRECATED attribute. If you are using them currently, the output spatial shape will be following: ` VALID: output_spatial_shape[i] = ceil((input_spatial_shape[i] - kernel_spatial_shape[i] + 1) / strides_spatial_shape[i]) SAME_UPPER or SAME_LOWER: output_spatial_shape[i] = ceil(input_spatial_shape[i] / strides_spatial_shape[i]) ` And pad shape will be following if *SAME_UPPER* or *SAME_LOWER*: ` pad_shape[i] = (output_spatial_shape[i] - 1) * strides_spatial_shape[i] + kernel_spatial_shape[i] - input_spatial_shape[i] ` The output of each pooling window is divided by the number of elements (exclude pad when attribute count_include_pad is zero).

Node Inputs

- **X** (T, single) – Input data tensor from the previous operator; dimensions for image case are (N x C x H x W), where N is the batch size, C is the number of channels, and H and W are the height and the width of the data. For non image case, the dimensions are in the form of (N x C x D1 x D2 … Dn), where N is the batch size. Optionally, if dimension denotation is in effect, the operation expects the input data tensor to arrive with the dimension denotation of [DATA_BATCH, DATA_CHANNEL, DATA_FEATURE, DATA_FEATURE …].

Node Outputs

- **Y** (T, single) – Output data tensor from average or max pooling across the input tensor. Dimensions will vary based on various kernel, stride, and pad sizes. Floor value of the dimension is used

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters

- **`name`** – the name of the node.
- **`auto_pad`** (`Optional [str]`, default='NOTSET') – auto_pad must be either NOTSET, SAME_UPPER, SAME_LOWER or VALID. Where default value is NOTSET, which means explicit padding is used. SAME_UPPER or SAME_LOWER mean pad the input so that the output spatial size match the input.In case of odd number add the extra padding at the end for SAME_UPPER and at the beginning for SAME_LOWER. VALID mean no padding.
- **`ceil_mode`** (`Optional [int]`, default=0) – Whether to use ceil or floor (default) to compute the output shape.
- **`count_include_pad`** (`Optional [int]`, default=0) – Whether include pad pixels when calculating values for the edges. Default is 0, doesn't count include pad.
- **`kernel_shape`** (`List [int]`) – The size of the kernel along each axis.
- **`pads`** (`Optional [List [int]]`, default=None) – Padding for the beginning and ending along each spatial axis, it can take any value greater than or equal to 0. The value represent the number of pixels added to the beginning and end part of the corresponding axis. *pads* format should be as follow [x1_begin, x2_begin… x1_end, x2_end,…], where xi_begin the number of pixels added at the beginning of axis *i* and xi_end, the number of pixels added at the end of axis *i*. This attribute cannot be used simultaneously with auto_pad attribute. If not present, the padding defaults to 0 along start and end of each spatial axis.
- **`strides`** (`Optional [List [int]]`, default=None) – Stride along each spatial axis. If not present, the stride defaults to 1 along each spatial axis.

`auto_pad`
auto_pad must be either NOTSET, SAME_UPPER, SAME_LOWER or VALID. Where default value is NOTSET, which means explicit padding is used. SAME_UPPER or SAME_LOWER mean pad the input so that the output spatial size match the input.In case of odd number add the extra padding at the end for SAME_UPPER and at the beginning for SAME_LOWER. VALID mean no padding.

`ceil_mode`
Whether to use ceil or floor (default) to compute the output shape.

`count_include_pad`
Whether include pad pixels when calculating values for the edges. Default is 0, doesn't count include pad.

kernel_shape
The size of the kernel along each axis.

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

strides
Stride along each spatial axis. If not present, the stride defaults to 1 along each spatial axis.

class ONNXBatchNormalization(*name*, *, *epsilon=9.999999747378752e-06*, *momentum=0.8999999761581421*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Carries out batch normalization as described in the paper https://arxiv.org/abs/1502.03167. Depending on the mode it is being run, there are multiple cases for the number of outputs, which we list below:

Output case #1: Y, mean, var, saved_mean, saved_var (training mode) Output case #2: Y (test mode)

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

Node Inputs

- **X** (T, single) – Input data tensor from the previous operator; dimensions are in the form of (N x C x D1 x D2 ... Dn), where N is the batch size, C is the number of channels. Statistics are computed for every channel of C over N and D1 to Dn dimensions. For image data, input dimensions become (N x C x H x W). The op also accepts single dimension input of size N in which case C is assumed to be 1
- **scale** (T, single) – Scale tensor of shape (C).
- **B** (T, single) – Bias tensor of shape (C).
- **in_mean** (T, single) – running (training) or estimated (testing) mean tensor of shape (C).
- **in_var** (T, single) – running (training) or estimated (testing) variance tensor of shape (C).

Node Outputs

- **Y** (T, single) – The output tensor of the same shape as X
- **out_mean** (T, optional) – The running mean after the BatchNormalization operator.
- **out_var** (T, optional) – The running variance after the BatchNormalization operator.
- **saved_mean** (T, optional) – Saved mean used during training to speed up gradient computation.
- **saved_var** (T, optional) – Saved variance used during training to speed up gradient computation.

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters

- **name** – the name of the node.
- **epsilon** (Optional [float], default=9.999999747378752e-06) – The epsilon value to use to avoid division by zero.
- **momentum** (Optional [float], default=0.8999999761581421) – Factor used in computing the running mean and variance.e.g., running_mean = running_mean * momentum + mean * (1 - momentum).

epsilon
The epsilon value to use to avoid division by zero.

momentum
Factor used in computing the running mean and variance.e.g., running_mean = running_mean * momentum + mean * (1 - momentum).

class ONNXBinarizer(*name*, *, *threshold=0.0*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Maps the values of the input tensor to either 0 or 1, element-wise, based on the outcome of a comparison against a threshold value.

Node Inputs

- **X** (T, single) – Data to be binarized

Node Outputs

- **Y** (T, single) – Binarized output data

Type Constraints

- **T** – float32, float64, int64, int32

Parameters

- **name** – the name of the node.
- **threshold** (Optional [float], default=0.0) – Values greater than this are mapped to 1, others to 0.

threshold
Values greater than this are mapped to 1, others to 0.

class ONNXBitShift(*name*, *, *direction*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Bitwise shift operator performs element-wise operation. For each input element, if the attribute "direction" is "RIGHT", this operator moves its binary representation toward the right side so that the input value is effectively decreased. If the attribute "direction" is "LEFT", bits of binary representation moves toward the left side, which results the increase of its actual value. The input X is the tensor to be shifted and another input Y specifies the amounts of shifting. For example, if "direction" is "Right", X is [1, 4], and S is [1, 1], the corresponding output Z would be [0, 2]. If "direction" is "LEFT" with X=[1, 2] and S=[1, 2], the corresponding output Y would be [2, 8].

Because this operator supports Numpy-style broadcasting, X's and Y's shapes are not necessarily identical.

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

Node Inputs

- **X** (T, single) – First operand, input to be shifted.

- **Y** (T, single) – Second operand, amounts of shift.

Node Outputs

- **Z** (T, single) – Output tensor

Type Constraints

- **T** – `uint8`, `uint16`, `uint32`, `uint64`

Parameters

- **`name`** – the name of the node.
- **`direction`** (`str`) – Direction of moving bits. It can be either "RIGHT" (for right shift) or "LEFT" (for left shift).

`direction`
Direction of moving bits. It can be either "RIGHT" (for right shift) or "LEFT" (for left shift).

`class ONNXCast`(*name*, ***, *to*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

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

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

Node Inputs

- **input** (T1, single) – Input tensor to be cast.

Node Outputs

- **output** (T2, single) – Output tensor with the same shape as input with type specified by the 'to' argument

Type Constraints

- **T1** – `float16`, `float32`, `float64`, `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, `uint64`, `bool_`
- **T2** – `float16`, `float32`, `float64`, `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, `uint64`, `bool_`

Parameters

- **`name`** – the name of the node.
- **`to`** (`int`) – 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

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

class ONNXCategoryMapper(*name*, ***, *cats_int64s=None*, *cats_strings=None*, *default_int64=- 1*, *default_string='_Unused'*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Converts strings to integers and vice versa.
 Two sequences of equal length are used to map between integers and strings, with strings and integers at the same index detailing the mapping.
 Each operator converts either integers to strings or strings to integers, depending on which default value attribute is provided. Only one default value attribute should be defined.
 If the string default value is set, it will convert integers to strings. If the int default value is set, it will convert strings to integers.

Node Inputs

- **X** (T1, single) – Input data

Node Outputs

- **Y** (T2, single) – Output data. If strings are input, the output values are integers, and vice versa.

Type Constraints

- **T1** – `int64`
- **T2** – `int64`

Parameters

- **name** – the name of the node.
- **cats_int64s** (`Optional [List [int]]`, default=None) – The integers of the map. This sequence must be the same length as the 'cats_strings' sequence.
- **cats_strings** (`Optional [List [str]]`, default=None) – The strings of the map. This sequence must be the same length as the 'cats_int64s' sequence
- **default_int64** (`Optional [int]`, default=-1) – An integer to use when an input string value is not found in the map.
One and only one of the 'default_*' attributes must be defined.
- **default_string** (`Optional [str]`, default='_Unused') – A string to use when an input integer value is not found in the map.
One and only one of the 'default_*' attributes must be defined.

cats_int64s
The integers of the map. This sequence must be the same length as the 'cats_strings' sequence.

cats_strings
The strings of the map. This sequence must be the same length as the 'cats_int64s' sequence

default_int64
An integer to use when an input string value is not found in the map.
One and only one of the 'default_*' attributes must be defined.

default_string
A string to use when an input integer value is not found in the map.
One and only one of the 'default_*' attributes must be defined.

class ONNXCeil(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Ceil takes one input data (Tensor<T>) and produces one output data (Tensor<T>) where the ceil is, y = ceil(x), is applied to the tensor elementwise.

Node Inputs

- **X** (T, single) – Input tensor

Node Outputs

- **Y** (T, single) – Output tensor

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters **name** – the name of the node.

class ONNXCelu(*name*, *, *alpha=1.0*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

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

Node Inputs

- **X** (T, single) – Input tensor

Node Outputs

- **Y** (T, single) – Output tensor

Type Constraints

- **T** – `float32`

Parameters

- **name** – the name of the node.
- **alpha** (`Optional` [`float`], default=1.0) – The Alpha value in Celu formula which control the shape of the unit. The default value is 1.0.

alpha
The Alpha value in Celu formula which control the shape of the unit. The default value is 1.0.

class ONNXClip(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Clip operator limits the given input within an interval. The interval is specified by the inputs 'min' and 'max'. They default to numeric_limits::lowest() and numeric_limits::max(), respectively.

Node Inputs

- **input** (T, single) – Input tensor whose elements to be clipped
- **min** (T, optional) – Minimum value, under which element is replaced by min. It must be a scalar(tensor of empty shape).
- **max** (T, optional) – Maximum value, above which element is replaced by max. It must be a scalar(tensor of empty shape).

Node Outputs

- **output** (T, single) – Output tensor with clipped input elements

Type Constraints

- **T** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`

Parameters **`name`** – the name of the node.

`class ONNXCompress`(*name*, *, *axis=None*)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **input** (T, single) – Tensor of rank r >= 1.
- **condition** (T1, single) – Rank 1 tensor of booleans to indicate which slices or data elements to be selected. Its length can be less than the input length along the axis or the flattened input size if axis is not specified. In such cases data slices or elements exceeding the condition length are discarded.

Node Outputs

- **output** (T, single) – Tensor of rank r if axis is specified. Otherwise output is a Tensor of rank 1.

Type Constraints

- **T** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`, `bool_`, `complex64`, `complex128`
- **T1** – `bool_`

Parameters

- **`name`** – the name of the node.
- **`axis`** (`Optional [int]`, default=None) – (Optional) Axis along which to take slices. If not specified, input is flattened before elements being selected. Negative value means counting dimensions from the back. Accepted range is [-r, r-1] where r = rank(input).

`axis`

(Optional) Axis along which to take slices. If not specified, input is flattened before elements being selected. Negative value means counting dimensions from the back. Accepted range is [-r, r-1] where r = rank(input).

`class ONNXConcat`(*name*, *, *axis*)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **inputs** (T, variadic) – List of tensors for concatenation

Node Outputs

- **concat_result** (T, single) – Concatenated tensor

Type Constraints

- **T** – uint8, uint16, uint32, uint64, int8, int16, int32, int64, float16, float32, float64, bool_, complex64, complex128

Parameters

- **name** – the name of the node.
- **axis** (int) – Which axis to concat on. A negative value means counting dimensions from the back. Accepted range is [-r, r-1] where r = rank(inputs)..

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

class ONNXConstant(*name*, ***, *sparse_value=None*, *value=None*, *value_float=None*, *value_floats=None*, *value_int=None*, *value_ints=None*, *value_string=None*, *value_strings=None*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

Node Outputs

- **output** (T, single) – Output tensor containing the same value of the provided tensor.

Type Constraints

- **T** – uint8, uint16, uint32, uint64, int8, int16, int32, int64, float16, float32, float64, bool_, complex64, complex128

Parameters

- **name** – the name of the node.
- **value** (Optional [numpy.ndarray], default=None) – The value for the elements of the output tensor.
- **value_float** (Optional [float], default=None) – The value for the sole element for the scalar, float32, output tensor.
- **value_floats** (Optional [List [float]], default=None) – The values for the elements for the 1D, float32, output tensor.
- **value_int** (Optional [int], default=None) – The value for the sole element for the scalar, int64, output tensor.
- **value_ints** (Optional [List [int]], default=None) – The values for the elements for the 1D, int64, output tensor.
- **value_string** (Optional [str], default=None) – The value for the sole element for the scalar, UTF-8 string, output tensor.
- **value_strings** (Optional [List [str]], default=None) – The values for the elements for the 1D, UTF-8 string, output tensor.

value
The value for the elements of the output tensor.

value_float
The value for the sole element for the scalar, float32, output tensor.

value_floats
The values for the elements for the 1D, float32, output tensor.

value_int
The value for the sole element for the scalar, int64, output tensor.

value_ints
The values for the elements for the 1D, int64, output tensor.

value_string
The value for the sole element for the scalar, UTF-8 string, output tensor.

value_strings
The values for the elements for the 1D, UTF-8 string, output tensor.

class ONNXConstantOfShape(*name*, *, *value=None*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Generate a tensor with given value and shape.

Node Inputs

- **input** (T1, single) – 1D tensor. The shape of the expected output tensor. If empty tensor is given, the output would be a scalar. All values must be >= 0.

Node Outputs

- **output** (T2, single) – Output tensor of shape specified by 'input'.If attribute 'value' is specified, the value and datatype of the output tensor is taken from 'value'.If attribute 'value' is not specified, the value in the output defaults to 0, and the datatype defaults to float32.

Type Constraints

- **T1** – `int64`
- **T2** – `float16`, `float32`, `float64`, `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, `uint64`, `bool_`

Parameters

- **name** – the name of the node.
- **value** (`Optional [numpy.ndarray]`, default=None) – (Optional) The value of the output elements.Should be a one-element tensor. If not specified, it defaults to a tensor of value 0 and datatype float32

value
(Optional) The value of the output elements.Should be a one-element tensor. If not specified, it defaults to a tensor of value 0 and datatype float32

class ONNXConv(*name*, *, *auto_pad='NOTSET'*, *dilations=None*, *group=1*, *kernel_shape=None*, *pads=None*, *strides=None*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **X** (T, single) – Input data tensor from previous layer; has size (N x C x H x W), where N is the batch size, C is the number of channels, and H and W are the height and width. Note that this is for the 2D image. Otherwise the size is (N x C x D1 x D2 … x Dn). Optionally, if dimension denotation is in effect, the operation expects input data tensor to arrive with the dimension denotation of [DATA_BATCH, DATA_CHANNEL, DATA_FEATURE, DATA_FEATURE …].
- **W** (T, single) – The weight tensor that will be used in the convolutions; has size (M x C/group x kH x kW), where C is the number of channels, and kH and kW are the height and width of the kernel, and M is the number of feature maps. For more than

2 dimensions, the kernel shape will be (M x C/group x k1 x k2 x ... x kn), where (k1 x k2 x ... kn) is the dimension of the kernel. Optionally, if dimension denotation is in effect, the operation expects the weight tensor to arrive with the dimension denotation of [FILTER_OUT_CHANNEL, FILTER_IN_CHANNEL, FILTER_SPATIAL, FILTER_SPATIAL ...]. X.shape[1] == (W.shape[1] * group) == C (assuming zero based indices for the shape array). Or in other words FILTER_IN_CHANNEL should be equal to DATA_CHANNEL.

- **B** (T, optional) – Optional 1D bias to be added to the convolution, has size of M.

Node Outputs

- **Y** (T, single) – Output data tensor that contains the result of the convolution. The output dimensions are functions of the kernel size, stride size, and pad lengths.

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters

- **`name`** – the name of the node.
- **`auto_pad`** (`Optional [str]`, default='NOTSET') – auto_pad must be either NOTSET, SAME_UPPER, SAME_LOWER or VALID. Where default value is NOTSET, which means explicit padding is used. SAME_UPPER or SAME_LOWER mean pad the input so that the output spatial size match the input.In case of odd number add the extra padding at the end for SAME_UPPER and at the beginning for SAME_LOWER. VALID mean no padding.
- **`dilations`** (`Optional [List [int]]`, default=None) – dilation value along each spatial axis of the filter. If not present, the dilation defaults is 1 along each spatial axis.
- **`group`** (`Optional [int]`, default=1) – number of groups input channels and output channels are divided into.
- **`kernel_shape`** (`Optional [List [int]]`, default=None) – The shape of the convolution kernel. If not present, should be inferred from input W.
- **`pads`** (`Optional [List [int]]`, default=None) – Padding for the beginning and ending along each spatial axis, it can take any value greater than or equal to 0. The value represent the number of pixels added to the beginning and end part of the corresponding axis. *pads* format should be as follow [x1_begin, x2_begin...x1_end, x2_end,...], where xi_begin the number of pixels added at the beginning of axis *i* and xi_end, the number of pixels added at the end of axis *i*. This attribute cannot be used simultaneously with auto_pad attribute. If not present, the padding defaults to 0 along start and end of each spatial axis.
- **`strides`** (`Optional [List [int]]`, default=None) – Stride along each spatial axis. If not present, the stride defaults is 1 along each spatial axis.

`auto_pad`
auto_pad must be either NOTSET, SAME_UPPER, SAME_LOWER or VALID. Where default value is NOTSET, which means explicit padding is used. SAME_UPPER or SAME_LOWER mean pad the input so that the output spatial size match the input.In case of odd number add the extra padding at the end for SAME_UPPER and at the beginning for SAME_LOWER. VALID mean no padding.

`dilations`
dilation value along each spatial axis of the filter. If not present, the dilation defaults is 1 along each spatial axis.

`group`
number of groups input channels and output channels are divided into.

kernel_shape
The shape of the convolution kernel. If not present, should be inferred from input W.

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

strides
Stride along each spatial axis. If not present, the stride defaults is 1 along each spatial axis.

class ONNXConvInteger(*name*, ***, *auto_pad='NOTSET'*, *dilations=None*, *group=1*, *kernel_shape=None*, *pads=None*, *strides=None*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

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

Node Outputs

- **y** (T3, single) – Output data tensor that contains the result of the convolution. The output dimensions are functions of the kernel size, stride size, and pad lengths.

Type Constraints

- **T1** – `int8`, `uint8`
- **T2** – `int8`, `uint8`
- **T3** – `int32`

Parameters

- **name** – the name of the node.
- **auto_pad** (Optional [str], default='NOTSET') – auto_pad must be either NOTSET, SAME_UPPER, SAME_LOWER or VALID. Where default value is NOTSET, which means explicit padding is used. SAME_UPPER or SAME_LOWER mean pad the input so that the output spatial size match the input.In case of odd number add the extra padding at the end for SAME_UPPER and at the beginning for SAME_LOWER. VALID mean no padding.
- **dilations** (Optional [List [int]], default=None) – dilation value along each spatial axis of the filter. If not present, the dilation defaults to 1 along each axis.
- **group** (Optional [int], default=1) – number of groups input channels and output channels are divided into. default is 1.
- **kernel_shape** (Optional [List [int]], default=None) – The shape of the convolution kernel. If not present, should be inferred from input 'w'.
- **pads** (Optional [List [int]], default=None) – Padding for the beginning and ending along each spatial axis, it can take any value greater than or equal to 0.The value represent the number of pixels added to the beginning and end part of the corresponding axis.`pads` format should be as follow [x1_begin, x2_begin...x1_end, x2_end,...], where xi_begin the number ofpixels added at the beginning of axis *i* and xi_end, the number of pixels added at the end of axis *i*.This attribute cannot be used simultaneously with auto_pad attribute. If not present, the padding defaultsto 0 along start and end of each spatial axis.
- **strides** (Optional [List [int]], default=None) – Stride along each spatial axis. If not present, the stride defaults to 1 along each axis.

auto_pad
: auto_pad must be either NOTSET, SAME_UPPER, SAME_LOWER or VALID. Where default value is NOTSET, which means explicit padding is used. SAME_UPPER or SAME_LOWER mean pad the input so that the output spatial size match the input.In case of odd number add the extra padding at the end for SAME_UPPER and at the beginning for SAME_LOWER. VALID mean no padding.

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

group
: number of groups input channels and output channels are divided into. default is 1.

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

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

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

class ONNXConvTranspose(*name*, ***, *auto_pad='NOTSET'*, *dilations=None*, *group=1*, *kernel_shape=None*, *output_padding=None*, *output_shape=None*, *pads=None*, *strides=None*)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

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

output_shape[i] = stride[i] * (input_size[i] - 1) + output_padding[i] + ((kernel_shape[i] - 1) * dilations[i] + 1) - pads[start_i] - pads[end_i]

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

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

Node Inputs

- **X** (T, single) – Input data tensor from previous layer; has size (N x C x H x W), where N is the batch size, C is the number of channels, and H and W are the height and width. Note that this is for the 2D image. Otherwise the size is (N x C x D1 x D2 … x Dn)
- **W** (T, single) – The weight tensor that will be used in the convolutions; has size (C x M/group x kH x kW), where C is the number of channels, and kH and kW are the height and width of the kernel, and M is the number of feature maps. For more than 2 dimensions, the weight shape will be (C x M/group x k1 x k2 x … x kn), where (k1 x k2 x … x kn) is the dimension of the kernel. The number of channels in the output should be equal to W.shape[1] * group (assuming zero based indices of the shape array)
- **B** (T, optional) – Optional 1D bias to be added to the convolution, has size of M.

Node Outputs

- **Y** (T, single) – Output data tensor that contains the result of the convolution. The output dimensions are functions of the kernel size, stride size, pad lengths and group count. The number of channels in the output should be equal to W.shape[1] * group (assuming zero based indices of the shape array)

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters

- **`name`** – the name of the node.
- **`auto_pad`** (`Optional [str]`, default='NOTSET') – auto_pad must be either NOTSET, SAME_UPPER, SAME_LOWER or VALID. Where default value is NOTSET, which means explicit padding is used. SAME_UPPER or SAME_LOWER mean pad the input so that the output spatial size match the input.In case of odd number add the extra padding at the end for SAME_UPPER and at the beginning for SAME_LOWER. VALID mean no padding.
- **`dilations`** (`Optional [List [int]]`, default=None) – dilation value along each spatial axis of the filter. If not present, the dilation defaults to 1 along each spatial axis.
- **`group`** (`Optional [int]`, default=1) – number of groups input channels and output channels are divided into.
- **`kernel_shape`** (`Optional [List [int]]`, default=None) – The shape of the convolution kernel. If not present, should be inferred from input W.
- **`output_padding`** (`Optional [List [int]]`, default=None) – Additional elements added to the side with higher coordinate indices in the output. Each padding value in "output_padding" must be less than the corresponding stride/dilation dimension. By default,

this attribute is a zero vector. Note that this attribute doesn't directly affect the computed output values. It only controls the selection of the computed values, so changing this attribute only adds or removes output elements. If "output_shape" is explicitly provided, "output_padding" does not contribute additional size to "output_shape" but participates in the computation of the needed padding amount. This is also called adjs or adjustment in some frameworks.

- **output_shape** (`Optional [List [int]]`, default=None) – The shape of the output can be explicitly set which will cause pads values to be auto generated. If output_shape is specified pads values are ignored. See doc for details for equations to generate pads
- **pads** (`Optional [List [int]]`, default=None) – Padding for the beginning and ending along each spatial axis, it can take any value greater than or equal to 0. The value represent the number of pixels added to the beginning and end part of the corresponding axis. *pads* format should be as follow [x1_begin, x2_begin…x1_end, x2_end,…], where xi_begin the number of pixels added at the beginning of axis *i* and xi_end, the number of pixels added at the end of axis *i*. This attribute cannot be used simultaneously with auto_pad attribute. If not present, the padding defaults to 0 along start and end of each spatial axis.
- **strides** (`Optional [List [int]]`, default=None) – Stride along each spatial axis. If not present, the stride defaults to 1 along each spatial axis.

auto_pad
: auto_pad must be either NOTSET, SAME_UPPER, SAME_LOWER or VALID. Where default value is NOTSET, which means explicit padding is used. SAME_UPPER or SAME_LOWER mean pad the input so that the output spatial size match the input.In case of odd number add the extra padding at the end for SAME_UPPER and at the beginning for SAME_LOWER. VALID mean no padding.

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

group
: number of groups input channels and output channels are divided into.

kernel_shape
: The shape of the convolution kernel. If not present, should be inferred from input W.

output_padding
: Additional elements added to the side with higher coordinate indices in the output. Each padding value in "output_padding" must be less than the corresponding stride/dilation dimension. By default, this attribute is a zero vector. Note that this attribute doesn't directly affect the computed output values. It only controls the selection of the computed values, so changing this attribute only adds or removes output elements. If "output_shape" is explicitly provided, "output_padding" does not contribute additional size to "output_shape" but participates in the computation of the needed padding amount. This is also called adjs or adjustment in some frameworks.

output_shape
: The shape of the output can be explicitly set which will cause pads values to be auto generated. If output_shape is specified pads values are ignored. See doc for details for equations to generate pads

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

strides
Stride along each spatial axis. If not present, the stride defaults to 1 along each spatial axis.

class ONNXCos(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **input** (T, single) – Input tensor

Node Outputs

- **output** (T, single) – The cosine of the input tensor computed element-wise

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters **name** – the name of the node.

class ONNXCosh(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **input** (T, single) – Input tensor

Node Outputs

- **output** (T, single) – The hyperbolic cosine values of the input tensor computed element-wise

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters **name** – the name of the node.

class ONNXCumSum(*name*, *, *exclusive=0*, *reverse=0*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

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

Node Inputs

- **x** (T, single) – An input tensor that is to be processed.
- **axis** (T2, single) – (Optional) A 0-D tensor. Must be in the range [-rank(x), rank(x)-1]. Negative value means counting dimensions from the back.

Node Outputs

- **y** (T, single) – Output tensor of the same type as 'x' with cumulative sums of the x's elements

Type Constraints

- **T** – `uint32`, `uint64`, `int32`, `int64`, `float32`, `float64`

- **T2** – int32, int64

Parameters

- **name** – the name of the node.
- **exclusive** (Optional [int], default=0) – If set to 1 will return exclusive sum in which the top element is not included. In other terms, if set to 1, the j-th output element would be the sum of the first (j-1) elements. Otherwise, it would be the sum of the first j elements.
- **reverse** (Optional [int], default=0) – If set to 1 will perform the sums in reverse direction.

exclusive
If set to 1 will return exclusive sum in which the top element is not included. In other terms, if set to 1, the j-th output element would be the sum of the first (j-1) elements. Otherwise, it would be the sum of the first j elements.

reverse
If set to 1 will perform the sums in reverse direction.

class ONNXDepthToSpace(*name*, *, *blocksize*, *mode='DCR'*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

DepthToSpace rearranges (permutes) data from depth into blocks of spatial data. This is the reverse transformation of SpaceToDepth. More specifically, this op outputs a copy of the input tensor where values from the depth dimension are moved in spatial blocks to the height and width dimensions. By default, *mode* = *DCR*. In the DCR mode, elements along the depth dimension from the input tensor are rearranged in the following order: depth, column, and then row. The output y is computed from the input x as below:

b, c, h, w = x.shape

tmp = np.reshape(x, [b, blocksize, blocksize, c // (blocksize**2), h, w])

tmp = np.transpose(tmp, [0, 3, 4, 1, 5, 2])

y = np.reshape(tmp, [b, c // (blocksize**2), h * blocksize, w * blocksize])

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

b, c, h, w = x.shape

tmp = np.reshape(x, [b, c // (blocksize ** 2), blocksize, blocksize, h, w])

tmp = np.transpose(tmp, [0, 1, 4, 2, 5, 3])

y = np.reshape(tmp, [b, c // (blocksize ** 2), h * blocksize, w * blocksize])

Node Inputs

- **input** (T, single) – Input tensor of [N,C,H,W], where N is the batch axis, C is the channel or depth, H is the height and W is the width.

Node Outputs

- **output** (T, single) – Output tensor of [N, C/(blocksize * blocksize), H * blocksize, W * blocksize].

Type Constraints

- **T** – uint8, uint16, uint32, uint64, int8, int16, int32, int64, float16, float32, float64, bool_, complex64, complex128

Parameters

- **name** – the name of the node.
- **blocksize** (int) – Blocks of [blocksize, blocksize] are moved.
- **mode** (Optional [str], default='DCR') – DCR (default) for depth-column-row order re-arrangement. Use CRD for column-row-depth order.

blocksize
Blocks of [blocksize, blocksize] are moved.

mode
DCR (default) for depth-column-row order re-arrangement. Use CRD for column-row-depth order.

class ONNXDequantizeLinear(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

The linear dequantization operator. It consumes a quantized tensor, a scale, a zero point to compute the full precision tensor. The dequantization formula is y = (x - x_zero_point) * x_scale. 'x_scale' and 'x_zero_point' must have same shape. 'x_zero_point' and 'x' must have same type. 'x' and 'y' must have same shape. In the case of dequantizing int32, there's no zero point (zero point is supposed to be 0).

Node Inputs

- **x** (T, single) – N-D quantized input tensor to be de-quantized.
- **x_scale** (x_scale_constraint, single) – Scale for input 'x'. It's a scalar, which means a per-tensor/layer quantization.
- **x_zero_point** (T, optional) – Zero point for input 'x'. It's a scalar, which means a per-tensor/layer quantization. It's optional. 0 is the default value when it's not specified.

Node Outputs

- **y** (y_constraint, single) – N-D full precision output tensor. It has same shape as input 'x'.

Type Constraints

- **T** – int8, uint8, int32
- **x_scale_constraint** – float32
- **y_constraint** – float32

Parameters **name** – the name of the node.

class ONNXDet(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **X** (T, single) – Input tensor

Node Outputs

- **Y** (T, single) – Output tensor

Type Constraints

- **T** – float16, float32, float64

Parameters **name** – the name of the node.

class ONNXDiv(*name*, *)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

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

Node Inputs

- **A** (T, single) – First operand.
- **B** (T, single) – Second operand.

Node Outputs

- **C** (T, single) – Result, has same element type as two inputs

Type Constraints

- **T** – `uint32`, `uint64`, `int32`, `int64`, `float16`, `float32`, `float64`

Parameters **name** – the name of the node.

class ONNXDropout(*name*, *, *seed=None*)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Dropout takes an input floating-point tensor, an optional input ratio (floating-point scalar) and an optional input training_mode (boolean scalar). It produces two tensor outputs, output (floating-point tensor) and mask (optional *Tensor<bool>*). If *training_mode* is true then the output Y will be a random dropout; Note that this Dropout scales the masked input data by the following equation, so to convert the trained model into inference mode, the user can simply not pass *training_mode* input or set it to false. ` `output = scale * data * mask,` ` where ` `scale = 1. / (1. - ratio).` ` This operator has **optional** inputs/outputs. See [the doc](IR.md) for more details about the representation of optional arguments. An empty string may be used in the place of an actual argument's name to indicate a missing argument. Trailing optional arguments (those not followed by an argument that is present) may also be simply omitted.

Node Inputs

- **data** (T, single) – The input data as Tensor.
- **ratio** (T1, optional) – The ratio of random dropout, with value in [0, 1). If this input was not set, or if it was set to 0, the output would be a simple copy of the input. If it's non-zero, output will be a random dropout of the scaled input, which is typically the case during training. It is an optional value, if not specified it will default to 0.5.
- **training_mode** (T2, optional) – If set to true then it indicates dropout is being used for training. It is an optional value hence unless specified explicitly, it is false. If it is false, ratio is ignored and the operation mimics inference mode where nothing will be dropped from the input data and if mask is requested as output it will contain all ones.

Node Outputs

- **output** (T, single) – The output.
- **mask** (T2, optional) – The output mask.

Type Constraints

- **T** – `float16`, `float32`, `float64`
- **T1** – `float16`, `float32`, `float64`
- **T2** – `bool_`

Parameters

- **name** – the name of the node.
- **seed** (Optional [int], default=None) – (Optional) Seed to the random generator, if not specified we will auto generate one.

seed
(Optional) Seed to the random generator, if not specified we will auto generate one.

class ONNXDynamicQuantizeLinear(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

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

` Zero point is calculated as: ` intermediate_zero_point = qmin - min(x)/y_scale y_zero_point = cast(round(saturate(itermediate_zero_point))) * where qmax and qmin are max and min values for quantization range .i.e [0, 255] in case of uint8 * for saturation, it saturates to [0, 255] if it's uint8, or [-127, 127] if it's int8. Right now only uint8 is supported. * rounding to nearest ties to even. ` Data quantization formula is: ` y = saturate (round (x / y_scale) + y_zero_point) * for saturation, it saturates to [0, 255] if it's uint8, or [-127, 127] if it's int8. Right now only uint8 is supported. * rounding to nearest ties to even. ```

Node Inputs

- **x** (T1, single) – Input tensor

Node Outputs

- **y** (T2, single) – Quantized output tensor
- **y_scale** (y_scale_constraint, single) – Output scale. It's a scalar, which means a per-tensor/layer quantization.
- **y_zero_point** (T2, single) – Output zero point. It's a scalar, which means a per-tensor/layer quantization.

Type Constraints

- **T1** – float32
- **T2** – uint8
- **y_scale_constraint** – float32

Parameters **name** – the name of the node.

class ONNXEinsum(*name*, *, *equation*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

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

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

The Einsum operator evaluates algebraic tensor operations on a sequence of tensors, using the Einstein summation convention. The equation string contains a comma-separated sequence of lower case letters. Each term corresponds to an operand tensor, and the characters within the terms correspond to operands dimensions.

This sequence may be followed by "->" to separate the left and right hand side of the equation. If the equation contains "->" followed by the right-hand side, the explicit (not classical) form of the Einstein summation is performed, and the right-hand side indices indicate output tensor dimensions. In other cases, output indices are (implicitly) set to the alphabetically sorted sequence of indices appearing exactly once in the equation.

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

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

Node Inputs

- **Inputs** (T, variadic) – Operands

Node Outputs

- **Output** (T, single) – Output tensor

Type Constraints

- **T** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`

Parameters

- **`name`** – the name of the node.
- **`equation`** (`str`) – Einsum expression string.

`equation`
Einsum expression string.

`class ONNXElu`(*name*, *, *alpha=1.0*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **X** (T, single) – 1D input tensor

Node Outputs

- **Y** (T, single) – 1D input tensor

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters

- **`name`** – the name of the node.
- **`alpha`** (`Optional` [`float`], default=1.0) – Coefficient of ELU.

`alpha`
Coefficient of ELU.

`class ONNXEqual`(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

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

Node Inputs

- **A** (T, single) – First input operand for the logical operator.
- **B** (T, single) – Second input operand for the logical operator.

Node Outputs

- **C** (T1, single) – Result tensor.

Type Constraints

- **T** – `bool_`, `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`
- **T1** – `bool_`

Parameters **`name`** – the name of the node.

`class ONNXErf`(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **input** (T, single) – Input tensor

Node Outputs

- **output** (T, single) – The error function of the input tensor computed element-wise. It has the same shape and type of the input.

Type Constraints

- **T** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`

Parameters **`name`** – the name of the node.

`class ONNXExp`(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **input** (T, single) – Input tensor

Node Outputs

- **output** (T, single) – The exponential of the input tensor computed element-wise

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters **`name`** – the name of the node.

`class ONNXExpand`(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Broadcast the input tensor following the given shape and the broadcast rule. The broadcast rule is similar to numpy.array(input) * numpy.ones(shape): Dimensions are right alignment; Two corresponding dimension must have the same value, or one of them is equal to 1. Also, this operator is similar to numpy.broadcast_to(input,

shape), but the major difference is numpy.broadcast_to() does not allow shape to be smaller than input.size(). It is possible that the output.shape is not equal to shape, when some dimensions in shape is equal to 1, or the shape.ndim < input.shape.ndim.

Node Inputs

- **input** (T, single) – Input tensor
- **shape** (shape_constraint, single) – A 1-D tensor indicates the shape you want to expand to, following the broadcast rule

Node Outputs

- **output** (T, single) – Output tensor

Type Constraints

- **T** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`, `bool_`, `complex64`, `complex128`
- **shape_constraint** – `int64`

Parameters **name** – the name of the node.

class ONNXEyeLike(*name*, *, *dtype=None*, *k=0*)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **input** (T1, single) – 2D input tensor to copy shape, and optionally, type information from.

Node Outputs

- **output** (T2, single) – Output tensor, same shape as input tensor T1.

Type Constraints

- **T1** – `float16`, `float32`, `float64`, `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, `uint64`, `bool_`
- **T2** – `float16`, `float32`, `float64`, `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, `uint64`, `bool_`

Parameters

- **name** – the name of the node.
- **dtype** (`Optional [int]`, default=None) – (Optional) The data type for the elements of the output tensor. If not specified,the data type of the input tensor T1 is used. If input tensor T1 is also notspecified, then type defaults to 'float'.
- **k** (`Optional [int]`, default=0) – (Optional) Index of the diagonal to be populated with ones. Default is 0. If T2 is the output, this op sets T2[i, i+k] = 1. k = 0 populates the main diagonal, k > 0 populates an upper diagonal, and k < 0 populates a lower diagonal.

dtype

(Optional) The data type for the elements of the output tensor. If not specified,the data type of the input tensor T1 is used. If input tensor T1 is also notspecified, then type defaults to 'float'.

k

(Optional) Index of the diagonal to be populated with ones. Default is 0. If T2 is the output, this op sets T2[i, i+k] = 1. k = 0 populates the main diagonal, k > 0 populates an upper diagonal, and k < 0 populates a lower diagonal.

class ONNXFeatureVectorizer(*name*, *, *inputdimensions=None*)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Concatenates input tensors into one continuous output.
 All input shapes are 2-D and are concatenated along the second dimention. 1-D tensors are treated as [1,C]. Inputs are copied to the output maintaining the order of the input arguments.
 All inputs must be integers or floats, while the output will be all floating point values.

Node Inputs

- **X** (T1, variadic) – An ordered collection of tensors, all with the same element type.

Node Outputs

- **Y** (Y_constraint, single) – The output array, elements ordered as the inputs.

Type Constraints

- **T1** – `int32`, `int64`, `float32`, `float64`
- **Y_constraint** – `float32`

Parameters

- **name** – the name of the node.
- **inputdimensions** (`Optional` [`List` [`int`]], default=None) – The size of each input in the input list

inputdimensions

The size of each input in the input list

class ONNXFlatten(*name*, *, *axis=1*)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Flattens the input tensor into a 2D matrix. If input tensor has shape (d_0, d_1, … d_n) then the output will have shape (d_0 X d_1 … d_(axis-1), d_axis X d_(axis+1) … X dn).

Node Inputs

- **input** (T, single) – A tensor of rank >= axis.

Node Outputs

- **output** (T, single) – A 2D tensor with the contents of the input tensor, with input dimensions up to axis flattened to the outer dimension of the output and remaining input dimensions flattened into the inner dimension of the output.

Type Constraints

- **T** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`, `bool_`, `complex64`, `complex128`

Parameters

- **name** – the name of the node.
- **axis** (`Optional` [`int`], default=1) – Indicate up to which input dimensions (exclusive) should be flattened to the outer dimension of the output. The value for axis must be in the range [-r, r], where r is the rank of the input tensor. Negative value means counting

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

axis
Indicate up to which input dimensions (exclusive) should be flattened to the outer dimension of the output. The value for axis must be in the range [-r, r], where r is the rank of the input tensor. Negative value means counting dimensions from the back. When axis = 0, the shape of the output tensor is (1, (d_0 X d_1 … d_n), where the shape of the input tensor is (d_0, d_1, … d_n).

class ONNXFloor(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Floor takes one input data (Tensor<T>) and produces one output data (Tensor<T>) where the floor is, y = floor(x), is applied to the tensor elementwise.

Node Inputs

- **X** (T, single) – Input tensor

Node Outputs

- **Y** (T, single) – Output tensor

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters **name** – the name of the node.

class ONNXGRU(*name*, *, *activation_alpha=None*, *activation_beta=None*, *activations=None*, *clip=None*, *direction='forward'*, *hidden_size=None*, *linear_before_reset=0*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Notations:

X - input tensor

z - update gate

r - reset gate

h - hidden gate

t - time step (t-1 means previous time step)

W[zrh] - W parameter weight matrix for update, reset, and hidden gates

R[zrh] - R recurrence weight matrix for update, reset, and hidden gates

Wb[zrh] - W bias vectors for update, reset, and hidden gates

Rb[zrh] - R bias vectors for update, reset, and hidden gates

WB[zrh] - W parameter weight matrix for backward update, reset, and hidden gates

RB[zrh] - R recurrence weight matrix for backward update, reset, and hidden gates

WBb[zrh] - W bias vectors for backward update, reset, and hidden gates

RBb[zrh] - R bias vectors for backward update, reset, and hidden gates

H - Hidden state

num_directions - 2 if direction == bidirectional else 1

Activation functions:

Relu(x) - max(0, x)

Tanh(x) - (1 - e^{-2x})/(1 + e^{-2x})

Sigmoid(x) - 1/(1 + e^{-x})

(NOTE: Below are optional)

Affine(x) - alpha*x + beta

LeakyRelu(x) - x if x >= 0 else alpha * x

ThresholdedRelu(x) - x if x >= alpha else 0

ScaledTanh(x) - alpha*Tanh(beta*x)

HardSigmoid(x) - min(max(alpha*x + beta, 0), 1)

Elu(x) - x if x >= 0 else alpha*(e^x - 1)

Softsign(x) - x/(1 + |x|)

Softplus(x) - log(1 + e^x)

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

- zt = f(Xt*(Wz^T) + Ht-1*(Rz^T) + Wbz + Rbz)
- rt = f(Xt*(Wr^T) + Ht-1*(Rr^T) + Wbr + Rbr)
- ht = g(Xt*(Wh^T) + (rt (.) Ht-1)*(Rh^T) + Rbh + Wbh) # default, when linear_before_reset = 0
- ht = g(Xt*(Wh^T) + (rt (.) (Ht-1*(Rh^T) + Rbh)) + Wbh) # when linear_before_reset != 0
- Ht = (1 - zt) (.) ht + zt (.) Ht-1

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

Node Inputs

- **X** (T, single) – The input sequences packed (and potentially padded) into one 3-D tensor with the shape of *[seq_length, batch_size, input_size]*.
- **W** (T, single) – The weight tensor for the gates. Concatenation of *W[zrh]* and *WB[zrh]* (if bidirectional) along dimension 0. This tensor has shape *[num_directions, 3*hidden_size, input_size]*.
- **R** (T, single) – The recurrence weight tensor. Concatenation of *R[zrh]* and *RB[zrh]* (if bidirectional) along dimension 0. This tensor has shape *[num_directions, 3*hidden_size, hidden_size]*.
- **B** (T, optional) – The bias tensor for the gates. Concatenation of *[Wb[zrh], Rb[zrh]]* and *[WBb[zrh], RBb[zrh]]* (if bidirectional) along dimension 0. This tensor has shape *[num_directions, 6*hidden_size]*. Optional: If not specified - assumed to be 0
- **sequence_lens** (T1, optional) – Optional tensor specifying lengths of the sequences in a batch. If not specified - assumed all sequences in the batch to have length *seq_length*. It has shape *[batch_size]*.
- **initial_h** (T, optional) – Optional initial value of the hidden. If not specified - assumed to be 0. It has shape *[num_directions, batch_size, hidden_size]*.

Node Outputs

- **Y** (T, optional) – A tensor that concats all the intermediate output values of the hidden. It has shape *[seq_length, num_directions, batch_size, hidden_size]*.
- **Y_h** (T, optional) – The last output value of the hidden. It has shape *[num_directions, batch_size, hidden_size]*.

Type Constraints

- **T** – `float16`, `float32`, `float64`
- **T1** – `int32`

Parameters

- **`name`** – the name of the node.
- **`activation_alpha`** (`Optional [List [float]]`, default=None) – Optional scaling values used by some activation functions. The values are consumed in the order of activation functions, for example (f, g, h) in LSTM. Default values are the same as of corresponding ONNX operators.For example with LeakyRelu, the default alpha is 0.01.
- **`activation_beta`** (`Optional [List [float]]`, default=None) – Optional scaling values used by some activation functions. The values are consumed in the order of activation functions, for example (f, g, h) in LSTM. Default values are the same as of corresponding ONNX operators.
- **`activations`** (`Optional [List [str]]`, default=None) – A list of 2 (or 4 if bidirectional) activation functions for update, reset, and hidden gates. The activation functions must be one of the activation functions specified above. Optional: See the equations for default if not specified.
- **`clip`** (`Optional [float]`, default=None) – Cell clip threshold. Clipping bounds the elements of a tensor in the range of [-threshold, +threshold] and is applied to the input of activations. No clip if not specified.
- **`direction`** (`Optional [str]`, default='forward') – Specify if the RNN is forward, reverse, or bidirectional. Must be one of forward (default), reverse, or bidirectional.
- **`hidden_size`** (`Optional [int]`, default=None) – Number of neurons in the hidden layer
- **`linear_before_reset`** (`Optional [int]`, default=0) – When computing the output of the hidden gate, apply the linear transformation before multiplying by the output of the reset gate.

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

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

`activations`
A list of 2 (or 4 if bidirectional) activation functions for update, reset, and hidden gates. The activation functions must be one of the activation functions specified above. Optional: See the equations for default if not specified.

clip
Cell clip threshold. Clipping bounds the elements of a tensor in the range of [-threshold, +threshold] and is applied to the input of activations. No clip if not specified.

direction
Specify if the RNN is forward, reverse, or bidirectional. Must be one of forward (default), reverse, or bidirectional.

hidden_size
Number of neurons in the hidden layer

linear_before_reset
When computing the output of the hidden gate, apply the linear transformation before multiplying by the output of the reset gate.

class ONNXGather(*name*, *, *axis=0*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

axis = 0 :

Let k = indices[i_{0}, ..., i_{q-1}] Then output[i_{0}, ..., i_{q-1}, j_{0}, ..., j_{r-2}] = input[k , j_{0}, ..., j_{r-2}]

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

axis = 1 :

Let k = indices[i_{0}, ..., i_{q-1}] Then output[i_{0}, ..., i_{q-1}, j_{0}, ..., j_{r-2}] = input[j_{0}, k, j_{1}, ..., j_{r-2}]

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

```
```

Node Inputs

- **data** (T, single) – Tensor of rank r >= 1.
- **indices** (Tind, single) – Tensor of int32/int64 indices, of any rank q. All index values are expected to be within bounds [-s, s-1] along axis of size s. It is an error if any of the index values are out of bounds.

Node Outputs

- **output** (T, single) – Tensor of rank q + (r - 1).

Type Constraints

- **T** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`, `bool_`, `complex64`, `complex128`
- **Tind** – `int32`, `int64`

Parameters

- **`name`** – the name of the node.
- **`axis`** (`Optional [int]`, default=0) – Which axis to gather on. Negative value means counting dimensions from the back. Accepted range is [-r, r-1] where r = rank(data).

`axis`
Which axis to gather on. Negative value means counting dimensions from the back. Accepted range is [-r, r-1] where r = rank(data).

`class ONNXGatherElements`(*name*, *, *axis=0*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

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

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

```

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

Example 1: ```

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

] indices = [

[0, 0], [1, 0],

] axis = 1 output = [

**[** [1, 1], [4, 3],

],

]

` Example 2: `
```

data = [[1, 2, 3], [4, 5, 6], [7, 8, 9],

] indices = [

[1, 2, 0], [2, 0, 0],

] axis = 0 output = [

[[4, 8, 3], [7, 2, 3],

],

]

```

**Node Inputs**

- **data** (T, single) – Tensor of rank r >= 1.
- **indices** (Tind, single) – Tensor of int32/int64 indices, with the same rank r as the input. All index values are expected to be within bounds [-s, s-1] along axis of size s. It is an error if any of the index values are out of bounds.

**Node Outputs**

- **output** (T, single) – Tensor of the same shape as indices.

**Type Constraints**

- **T** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`, `bool_`, `complex64`, `complex128`
- **Tind** – `int32`, `int64`

**Parameters**

- **`name`** – the name of the node.
- **`axis`** (`Optional [int]`, default=0) – Which axis to gather on. Negative value means counting dimensions from the back. Accepted range is [-r, r-1] where r = rank(data).

**`axis`**
Which axis to gather on. Negative value means counting dimensions from the back. Accepted range is [-r, r-1] where r = rank(data).

**`class ONNXGatherND`** (*name*, *, *batch_dims=0*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Given *data* tensor of rank *r* >= 1, *indices* tensor of rank *q* >= 1, and *batch_dims* integer *b*, this operator gathers slices of *data* into an output tensor of rank *q + r - indices_shape[-1] - 1 - b*.

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

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

Some salient points about the inputs' rank and shape:

1) r >= 1 and q >= 1 are to be honored. There is no dependency condition to be met between ranks *r* and *q*
2) The first *b* dimensions of the shape of *indices* tensor and *data* tensor must be equal.
3) b < min(q, r) is to be honored.
4) The *indices_shape[-1]* should have a value between 1 (inclusive) and rank *r-b* (inclusive)
```

5) All values in *indices* are expected to be within bounds [-s, s-1] along axis of size *s* (i.e.) *-data_shape[i] <= indices[...,i] <= data_shape[i] - 1*. It is an error if any of the index values are out of bounds.

The output is computed as follows:

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

1) If *indices_shape[-1] > r-b* => error condition

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

3) If *indices_shape[-1] < r-b*, since the rank of *indices* is *q*, *indices* can be thought of as *N (q-b-1)*-dimensional tensor containing 1-D tensors of dimension < *r-b*. Let us think of each such tensors as *indices_slice*. Each *tensor slice* corresponding to *data[0:b-1, indices_slice , :]* is filled into the corresponding location of the *(q-b-1)*-dimensional tensor to form the *output* tensor (Examples 2, 3, 4 and 5 below)

This operator is the inverse of *ScatterND*.

Example 1

batch_dims = 0

data = [[0,1],[2,3]] # data_shape = [2, 2]

indices = [[0,0],[1,1]] # indices_shape = [2, 2]

output = [0,3] # output_shape = [2]

Example 2

batch_dims = 0

data = [[0,1],[2,3]] # data_shape = [2, 2]

indices = [[1],[0]] # indices_shape = [2, 1]

output = [[2,3],[0,1]] # output_shape = [2, 2]

Example 3

batch_dims = 0

data = [[[0,1],[2,3]],[[4,5],[6,7]]] # data_shape = [2, 2, 2]

indices = [[0,1],[1,0]] # indices_shape = [2, 2]

output = [[2,3],[4,5]] # output_shape = [2, 2]

Example 4

batch_dims = 0

data = [[[0,1],[2,3]],[[4,5],[6,7]]] # data_shape = [2, 2, 2]

indices = [[[0,1]],[[1,0]]] # indices_shape = [2, 1, 2]

output = [[[2,3]],[[4,5]]] # output_shape = [2, 1, 2]

Example 5

batch_dims = 1

data = [[[0,1],[2,3]],[[4,5],[6,7]]] # data_shape = [2, 2, 2]

indices = [[1],[0]] # indices_shape = [2, 1]

output = [[2,3],[4,5]] # output_shape = [2, 2]

Node Inputs

- **data** (T, single) – Tensor of rank r >= 1.
- **indices** (indices_constraint, single) – Tensor of rank q >= 1. All index values are expected to be within bounds [-s, s-1] along axis of size s. It is an error if any of the index values are out of bounds.

Node Outputs

- **output** (T, single) – Tensor of rank q + r - indices_shape[-1] - 1.

Type Constraints

- **T** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`, `bool_`, `complex64`, `complex128`
- **indices_constraint** – `int64`

Parameters

- **name** – the name of the node.
- **batch_dims** (`Optional [int]`, default=0) – The number of batch dimensions. The gather of indexing starts from dimension of data[batch_dims:]

batch_dims
The number of batch dimensions. The gather of indexing starts from dimension of data[batch_dims:]

class ONNXGemm(*name*, *, *alpha=1.0*, *beta=1.0*, *transA=0*, *transB=0*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

A' = transpose(A) if transA else A

B' = transpose(B) if transB else B

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

Node Inputs

- **A** (T, single) – Input tensor A. The shape of A should be (M, K) if transA is 0, or (K, M) if transA is non-zero.
- **B** (T, single) – Input tensor B. The shape of B should be (K, N) if transB is 0, or (N, K) if transB is non-zero.
- **C** (T, optional) – Optional input tensor C. If not specified, the computation is done as if C is a scalar 0. The shape of C should be unidirectional broadcastable to (M, N).

Node Outputs

- **Y** (T, single) – Output tensor of shape (M, N).

Type Constraints

- **T** – `float16`, `float32`, `float64`, `uint32`, `uint64`, `int32`, `int64`

Parameters

- **name** – the name of the node.
- **alpha** (`Optional` [`float`], default=1.0) – Scalar multiplier for the product of input tensors A * B.
- **beta** (`Optional` [`float`], default=1.0) – Scalar multiplier for input tensor C.
- **transA** (`Optional` [`int`], default=0) – Whether A should be transposed
- **transB** (`Optional` [`int`], default=0) – Whether B should be transposed

alpha
Scalar multiplier for the product of input tensors A * B.

beta
Scalar multiplier for input tensor C.

transA
Whether A should be transposed

transB
Whether B should be transposed

class ONNXGlobalAveragePool(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **X** (T, single) – Input data tensor from the previous operator; dimensions for image case are (N x C x H x W), where N is the batch size, C is the number of channels, and H and W are the height and the width of the data. For non image case, the dimensions are in the form of (N x C x D1 x D2 … Dn), where N is the batch size.

Node Outputs

- **Y** (T, single) – Output data tensor from pooling across the input tensor. The output tensor has the same rank as the input. The first two dimensions of output shape are the same as the input (N x C), while the other dimensions are all 1.

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters **name** – the name of the node.

class ONNXGlobalLpPool(*name*, *, *p=2*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **X** (T, single) – Input data tensor from the previous operator; dimensions for image case are (N x C x H x W), where N is the batch size, C is the number of channels, and H and W are the height and the width of the data. For non image case, the dimensions are in the form of (N x C x D1 x D2 … Dn), where N is the batch size.

Node Outputs

- **Y** (T, single) – Output data tensor from pooling across the input tensor. The output tensor has the same rank as the input. The first two dimensions of output shape are the same as the input (N x C), while the other dimensions are all 1.

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters

- **`name`** – the name of the node.
- **`p`** (`Optional` [`int`], default=2) – p value of the Lp norm used to pool over the input data.

`p`
p value of the Lp norm used to pool over the input data.

`class ONNXGlobalMaxPool`(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **X** (T, single) – Input data tensor from the previous operator; dimensions for image case are (N x C x H x W), where N is the batch size, C is the number of channels, and H and W are the height and the width of the data. For non image case, the dimensions are in the form of (N x C x D1 x D2 … Dn), where N is the batch size.

Node Outputs

- **Y** (T, single) – Output data tensor from pooling across the input tensor. The output tensor has the same rank as the input. The first two dimensions of output shape are the same as the input (N x C), while the other dimensions are all 1.

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters **`name`** – the name of the node.

`class ONNXGradient`(*name*, *, *xs*, *y*, *zs=None*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Gradient operator computes the partial derivatives of a specific tensor w.r.t. some other tensors. This operator is widely used in gradient-based training algorithms. To illustrate its use, let's consider a computation graph,

``` X —-.

v

**W –> Conv –> H –> Gemm –> Y** Z
```

```
```

, where W and Z are trainable tensors. Note that operators' attributes are omitted for the sake of simplicity. Let dY/dW (dY/dZ) be the gradient of Y with respect to W (Z). The user can compute gradient by inserting Gradient operator to form another graph shown below.

```
``` W –> Conv –> H –> Gemm –> Y | ^ ^ | | | | X Z | | | | | .————-‘ | | | (W/Z/X is the 1st/2nd/3rd input of Gradient as shown in | | | “xs” followed by “zs”) | v v ‘—> Gradient(xs=[“W”, “Z”], zs=[“X”], y=”Y”)

    |
    ‘————————————————> dY/dW (1st output of Gradient)


    ‘————————————————> dY/dZ (2nd output of Gradient)
```

By definition, the tensor “y” is a function of independent variables in “xs” and “zs”. Since we only compute the gradient of “y” w.r.t. the differentiable variables in “xs”, this Gradient only outputs dY/dW and dY/dZ. Note that “H” cannot appear in “xs” and “zs”. The reason is that “H” can be determined by tensors “W” and “X” and therefore “H” is not an independent variable.

All outputs are optional. If needed, for example, user can assign an empty string to the 1st output name of that Gradient to skip the generation of dY/dW. Note that the concept of optional outputs can also be found in ONNX's RNN, GRU, and LSTM.

Gradient operator can compute derivative against intermediate tensors. For example, the gradient of Y with respect to H can be done via

```
``` W –> Conv –> H –> Gemm –> Y

        ^ | ^ | | | X | Z .——-‘ | | .————-‘ | | (H/Z is the 1st/2nd input of Gradient as shown in “xs”)
        v v

    Gradient(xs=[“H”, “Z”], y=”Y”)

        |
        ‘————————————————> dY/dH (1st output of Gradient)


        ‘————————————————> dY/dZ (2nd output of Gradient)
```

```
``` W –> Gemm –> Y –> Loss –> O

    ^ ^ | | X L
```

```
To compute the 2nd order derivative of O with respect to W (denoted by d^2O/dW^2), one can do

```
``` W –> Gemm –> Y –> Loss –> O | ^ ^ | | | | X .————-L | | | | | | | v +——+-+> Gradient(xs=[“X”, “W”], zs=[“L”], y=”O”) —> dO/dX (1st output of Gradient) | | | | | | ‘—> dO/dW (2nd output of Gradient) | v v ‘—> Gradient(xs=[“X”, “W”], zs=[“L”], y=”dO/dW”) —> d(dO/dW)dX (1st output of

    Gradient)
```
```

‘—> d^2O/dW^2 (2nd output of Gradient)

```

The tensors named in attributes “xs”, “zs”, and “y” define the differentiated computation graph, and the inputs to Gradient node define the values at which the gradient is computed. We can feed different tensors to the identified graph. For example, one can compute the gradient of Y with respect to H at a specific value of H, H_1, by providing that value as an input to the Gradient node.

``` W –> Conv –> H –> Gemm –> Y

^ ^ | | X Z

**Z_1 (2nd input of Gradient)**

v

**H_1 –> Gradient(xs=[“H”, “Z”], y=”Y”) —> dY/dH when H = H_1 and Y = Y_1.**

‘————————————> dY/dZ (2nd output of Gradient)

```

When the inputs of Gradient are the tensors named in “xs” and “zs”, the computation can be optimized. More specifically, intermediate variables in forward pass can be reused if the gradient is computed via reverse-mode auto-differentiation.

Node Inputs

- **Inputs** (T1, variadic) – The values fed into graph identified by the attributes. The i-th input is the value of the i-th tensor specified in the concatenated list of the attribute “xs” and the attribute “zs”. For example, if xs=[“A”, “B”] and zs=[“C”], the first input is used as the value of symbol “A” and the 3rd input is substituted for all the occurrences of “C”.

Node Outputs

- **Outputs** (T2, variadic) – The gradient of the tensor specified by the attribute “y” with respect to each of tensors specified in the attribute “xs”. The i-th output is the gradient of “y” with respect to the i-th tensor specified in the attribute “xs”.

Type Constraints

- **T1** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`, `bool_`, `complex64`, `complex128`
- **T2** – `float16`, `float32`, `float64`

Parameters

- **`name`** – the name of the node.
- **`xs`** (`List [str]`) – Input tensor names of the differentiated sub-graph. It contains only the necessary differentiated inputs of a (sub-)graph. Variables (usually called intermediate variables) that can be generated from inputs cannot be included in this attribute.
- **`y`** (`str`) – The targeted tensor. It can be viewed as the output of the differentiated function. The attribute “xs” and attribute “zs” are the minimal independent variable set that determines the value of “y”.
```

- **zs** (Optional [List [str]], default=None) – Input tensor names of the differentiated sub-graph. It contains only the necessary non-differentiated inputs of a (sub-)graph. Variables (usually called intermediate variables) that can be generated from inputs cannot be included in this attribute.

xs
Input tensor names of the differentiated sub-graph. It contains only the necessary differentiated inputs of a (sub-)graph. Variables (usually called intermediate variables) that can be generated from inputs cannot be included in this attribute.

y
The targeted tensor. It can be viewed as the output of the differentiated function. The attribute "xs" and attribute "zs" are the minimal independent variable set that determines the value of "y".

zs
Input tensor names of the differentiated sub-graph. It contains only the necessary non-differentiated inputs of a (sub-)graph. Variables (usually called intermediate variables) that can be generated from inputs cannot be included in this attribute.

class ONNXGraphCall(*name*, *, *graph_name*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

The GraphCall operator invokes a graph inside TrainingInfoProto's algorithm field. The GraphCall inputs and outputs are bound to those of invoked graph by position. If a graph input has an initializer, that input is considered optional. All graph outputs are optional.

Below Python syntax is used for describing dictionary and list.

Assume that ModelProto's graph field has - name: "MyInferenceGraph" - input: ["X", "W", "Z"] - initializer: [W] - output: ["Y"]

as visualized below for inference.

``` X —-.

v

**W –> Conv –> H –> Gemm –> Y** Z

```

Assume that the training algorithm contains

- inputs: ["X_1", "Z_1", "C"]
- initializer: [T]
- outputs: ["W_new"]

with a dictionary

- update_binding: {"W": "W_new", "T": "T_new"}

Inside the training algorithm graph, one can invoke the inference graph via adding a GraphCall node with

- inputs: ["X_1", "W", Z_1"]
- outputs: ["Y_1"]
- an attribute graph_name="MyInferenceGraph",

The initializers, “W” and “T” in this case, in update_binding are considered globally-visible and mutable variables, which can be used as inputs of operators in the training graph.

An example training algorithm graph may look like

```
` .-------- W (a global and mutable variable from | | the inference
graph) | | | .-----'-----------. | | | | v | | .-- X_1 -->
GraphCall(graph_name="MyInferenceGraph") | | | | | | | | | | | Z_1
-----' | | | | V | | | | Y_1 ---> Loss ---> O | | | | ^ | | | | | |
| `--. | C | | | | | | | | .----------------' | | | | | | v v v |
`--> Gradient(xs=["W"], zs=["X_1", "Z_1", "C"], y="O") | | | v | dO_dW
(gradient of W) 1 (a scalar one) | | | | V v | Div <--- T ------------>
Add ---> T_new | | (T is the number of training iterations. | | T is also
globally visible and mutable.) | v `-----> Sub ----> W_new `
```

where Loss is a dummy node which computes the minimized objective function.

The variable “W” is an optional input in the called graph. If the user omits it, the input list of GraphCall becomes [“X_1”, “”, “Z_1”]. In this case, from the view of computation graph, the Conv operator invoked by GraphCall’s may be still connected the global “W” variable and therefore the structure of the computation graph is unchanged.

Node Inputs

- **Inputs** (T, variadic) – Inputs fed to the invoked graph. The i-th input here goes to the i-th input of the invoked graph. To omit an optional input in this field, the user can drop it or use an empty string.

Node Outputs

- **Outputs** (T, variadic) – The outputs generated by the called graph. Its i-th value is bound to the i-th output of the called graph. Similar to the inputs, all outputs are optional.

Type Constraints

- **T** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`, `bool_`, `complex64`, `complex128`

Parameters

- **`name`** – the name of the node.
- **`graph_name`** (`str`) – The invoked graph’s name. The only allowed value is the name of the inference graph, which is stored in “ModelProto.graph.name” in the ONNX model format.

`graph_name`
The invoked graph’s name. The only allowed value is the name of the inference graph, which is stored in “ModelProto.graph.name” in the ONNX model format.

`class ONNXGreater`(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

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

Node Inputs

- **A** (T, single) – First input operand for the logical operator.
- **B** (T, single) – Second input operand for the logical operator.

Node Outputs

- **C** (T1, single) – Result tensor.

Type Constraints

- **T** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`
- **T1** – `bool_`

Parameters **name** – the name of the node.

class ONNXGreaterOrEqual(*name*, *)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

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

Node Inputs

- **A** (T, single) – First input operand for the logical operator.
- **B** (T, single) – Second input operand for the logical operator.

Node Outputs

- **C** (T1, single) – Result tensor.

Type Constraints

- **T** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`
- **T1** – `bool_`

Parameters **name** – the name of the node.

class ONNXHardSigmoid(*name*, *, *alpha=0.20000000298023224*, *beta=0.5*)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **X** (T, single) – Input tensor

Node Outputs

- **Y** (T, single) – Output tensor

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters

- **name** – the name of the node.
- **alpha** (`Optional` [`float`], default=0.20000000298023224) – Value of alpha.
- **beta** (`Optional` [`float`], default=0.5) – Value of beta.

alpha

Value of alpha.

beta
Value of beta.

class ONNXHardmax(*name*, *, *axis=1*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

The operator computes the hardmax (1 for the first maximum value, and 0 for all others) values for each layer in the batch
of the given input.

The input does not need to explicitly be a 2D vector; rather, it will be coerced into one. For an arbitrary n-dimensional tensor input in [a_0, a_1, ..., a_{k-1}, a_k, ..., a_{n-1}] and k is the axis provided, then input will be coerced into a 2-dimensional tensor with dimensions [a_0 * ... * a_{k-1}, a_k * ... * a_{n-1}]. For the default case where axis=1, this means the input tensor will be coerced into a 2D tensor of dimensions [a_0, a_1 * ... * a_{n-1}], where a_0 is often the batch size. In this situation, we must have a_0 = N and a_1 * ... * a_{n-1} = D. Each of these dimensions must be matched correctly, or else the operator will throw errors. The output tensor has the same shape and contains the hardmax values of the corresponding input.

Node Inputs

- **input** (T, single) – The input tensor that's coerced into a 2D matrix of size (NxD) as described above.

Node Outputs

- **output** (T, single) – The output values with the same shape as input tensor (the original size without coercion).

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters

- **name** – the name of the node.
- **axis** (`Optional` [`int`], default=1) – Describes the axis of the inputs when coerced to 2D; defaults to one because the 0th axis most likely describes the batch_size. Negative value means counting dimensions from the back. Accepted range is [-r, r-1] where r = rank(input).

axis
Describes the axis of the inputs when coerced to 2D; defaults to one because the 0th axis most likely describes the batch_size. Negative value means counting dimensions from the back. Accepted range is [-r, r-1] where r = rank(input).

class ONNXIdentity(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Identity operator

Node Inputs

- **input** (T, single) – Input tensor

Node Outputs

- **output** (T, single) – Tensor to copy input into.

Type Constraints

- **T** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`, `bool_`, `complex64`, `complex128`

Parameters **name** – the name of the node.

class ONNXImputer(*name*, ***, *imputed_value_floats=None*, *imputed_value_int64s=None*, *replaced_value_float=0.0*, *replaced_value_int64=0*)

Bases: daceml.onnx.nodes.onnx_op.ONNXOp

Replaces inputs that equal one value with another, leaving all other elements alone.
 This operator is typically used to replace missing values in situations where they have a canonical representation, such as -1, 0, NaN, or some extreme value.
 One and only one of imputed_value_floats or imputed_value_int64s should be defined – floats if the input tensor holds floats, integers if the input tensor holds integers. The imputed values must all fit within the width of the tensor element type. One and only one of the replaced_value_float or replaced_value_int64 should be defined, which one depends on whether floats or integers are being processed.
 The imputed_value attribute length can be 1 element, or it can have one element per input feature.
In other words, if the input tensor has the shape [*,F], then the length of the attribute array may be 1 or F. If it is 1, then it is broadcast along the last dimension and applied to each feature.

Node Inputs

- **X** (T, single) – Data to be processed.

Node Outputs

- **Y** (T, single) – Imputed output data

Type Constraints

- **T** – `float32`, `float64`, `int64`, `int32`

Parameters

- **name** – the name of the node.
- **imputed_value_floats** (`Optional [List [float]]`, default=None) – Value(s) to change to
- **imputed_value_int64s** (`Optional [List [int]]`, default=None) – Value(s) to change to.
- **replaced_value_float** (`Optional [float]`, default=0.0) – A value that needs replacing.
- **replaced_value_int64** (`Optional [int]`, default=0) – A value that needs replacing.

imputed_value_floats
Value(s) to change to

imputed_value_int64s
Value(s) to change to.

replaced_value_float
A value that needs replacing.

replaced_value_int64
A value that needs replacing.

class ONNXInstanceNormalization(*name*, ***, *epsilon=9.999999747378752e-06*)

Bases: daceml.onnx.nodes.onnx_op.ONNXOp

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

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

Node Inputs

- **input** (T, single) – Input data tensor from the previous operator; dimensions for image case are (N x C x H x W), where N is the batch size, C is the number of channels, and H and W are the height and the width of the data. For non image case, the dimensions are in the form of (N x C x D1 x D2 … Dn), where N is the batch size.
- **scale** (T, single) – The input 1-dimensional scale tensor of size C.
- **B** (T, single) – The input 1-dimensional bias tensor of size C.

Node Outputs

- **output** (T, single) – The output tensor of the same shape as input.

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters

- **`name`** – the name of the node.
- **`epsilon`** (`Optional` [`float`], default=9.999999747378752e-06) – The epsilon value to use to avoid division by zero.

`epsilon`
The epsilon value to use to avoid division by zero.

`class ONNXIsInf`(*name*, *, *detect_negative=1*, *detect_positive=1*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Map infinity to true and other values to false.

Node Inputs

- **X** (T1, single) – input

Node Outputs

- **Y** (T2, single) – output

Type Constraints

- **T1** – `float32`, `float64`
- **T2** – `bool_`

Parameters

- **`name`** – the name of the node.
- **`detect_negative`** (`Optional` [`int`], default=1) – (Optional) Whether map negative infinity to true. Default to 1 so that negative infinity induces true. Set this attribute to 0 if negative infinity should be mapped to false.
- **`detect_positive`** (`Optional` [`int`], default=1) – (Optional) Whether map positive infinity to true. Default to 1 so that positive infinity induces true. Set this attribute to 0 if positive infinity should be mapped to false.

`detect_negative`
(Optional) Whether map negative infinity to true. Default to 1 so that negative infinity induces true. Set this attribute to 0 if negative infinity should be mapped to false.

`detect_positive`
(Optional) Whether map positive infinity to true. Default to 1 so that positive infinity induces true. Set this attribute to 0 if positive infinity should be mapped to false.

class ONNXIsNaN(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Returns which elements of the input are NaN.

Node Inputs

- **X** (T1, single) – input

Node Outputs

- **Y** (T2, single) – output

Type Constraints

- **T1** – `float16`, `float32`, `float64`
- **T2** – `bool_`

Parameters **name** – the name of the node.

class ONNXLRN(*name*, *, *alpha=9.999999747378752e-05*, *beta=0.75*, *bias=1.0*, *size*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Local Response Normalization proposed in the [AlexNet paper](https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf). It normalizes over local input regions. The local region is defined across the channels. For an element X[n, c, d1, ..., dk] in a tensor of shape (N x C x D1 x D2, ..., Dk), its region is {X[n, i, d1, ..., dk] | max(0, c - floor((size - 1) / 2)) <= i <= min(C - 1, c + ceil((size - 1) / 2))}.

square_sum[n, c, d1, ..., dk] = sum(X[n, i, d1, ..., dk] ^ 2), where max(0, c - floor((size - 1) / 2)) <= i <= min(C - 1, c + ceil((size - 1) / 2)).

Y[n, c, d1, ..., dk] = X[n, c, d1, ..., dk] / (bias + alpha / size * square_sum[n, c, d1, ..., dk]) ^ beta

Node Inputs

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

Node Outputs

- **Y** (T, single) – Output tensor, which has the shape and type as input tensor

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters

- **name** – the name of the node.
- **alpha** (`Optional` [`float`], default=9.999999747378752e-05) – Scaling parameter.
- **beta** (`Optional` [`float`], default=0.75) – The exponent.
- **bias** (`Optional` [`float`], default=1.0) –
- **size** (`int`) – The number of channels to sum over

alpha
Scaling parameter.

beta
The exponent.

bias
Object property of type float

size
The number of channels to sum over

class ONNXLSTM(*name*, ***, *activation_alpha=None*, *activation_beta=None*, *activations=None*, *clip=None*, *direction='forward'*, *hidden_size=None*, *input_forget=0*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Notations:

X - input tensor

i - input gate

o - output gate

f - forget gate

c - cell gate

t - time step (t-1 means previous time step)

W[iofc] - W parameter weight matrix for input, output, forget, and cell gates

R[iofc] - R recurrence weight matrix for input, output, forget, and cell gates

Wb[iofc] - W bias vectors for input, output, forget, and cell gates

Rb[iofc] - R bias vectors for input, output, forget, and cell gates

P[iof] - P peephole weight vector for input, output, and forget gates

WB[iofc] - W parameter weight matrix for backward input, output, forget, and cell gates

RB[iofc] - R recurrence weight matrix for backward input, output, forget, and cell gates

WBb[iofc] - W bias vectors for backward input, output, forget, and cell gates

RBb[iofc] - R bias vectors for backward input, output, forget, and cell gates

PB[iof] - P peephole weight vector for backward input, output, and forget gates

H - Hidden state

num_directions - 2 if direction == bidirectional else 1

Activation functions:

Relu(x) - max(0, x)

Tanh(x) - (1 - e^{-2x})/(1 + e^{-2x})

Sigmoid(x) - 1/(1 + e^{-x})

(NOTE: Below are optional)

Affine(x) - alpha*x + beta

LeakyRelu(x) - x if x >= 0 else alpha * x

ThresholdedRelu(x) - x if x >= alpha else 0

ScaledTanh(x) - alpha*Tanh(beta*x)

HardSigmoid(x) - min(max(alpha*x + beta, 0), 1)

Elu(x) - x if x >= 0 else alpha*(e^x - 1)

Softsign(x) - x/(1 + |x|)

Softplus(x) - log(1 + e^x)

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

- it = f(Xt*(Wi^T) + Ht-1*(Ri^T) + Pi (.) Ct-1 + Wbi + Rbi)
- ft = f(Xt*(Wf^T) + Ht-1*(Rf^T) + Pf (.) Ct-1 + Wbf + Rbf)
- ct = g(Xt*(Wc^T) + Ht-1*(Rc^T) + Wbc + Rbc)
- Ct = ft (.) Ct-1 + it (.) ct
- ot = f(Xt*(Wo^T) + Ht-1*(Ro^T) + Po (.) Ct + Wbo + Rbo)
- Ht = ot (.) h(Ct)

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

Node Inputs

- **X** (T, single) – The input sequences packed (and potentially padded) into one 3-D tensor with the shape of *[seq_length, batch_size, input_size]*.
- **W** (T, single) – The weight tensor for the gates. Concatenation of *W[iofc]* and *WB[iofc]* (if bidirectional) along dimension 0. The tensor has shape *[num_directions, 4*hidden_size, input_size]*.
- **R** (T, single) – The recurrence weight tensor. Concatenation of *R[iofc]* and *RB[iofc]* (if bidirectional) along dimension 0. This tensor has shape *[num_directions, 4*hidden_size, hidden_size]*.
- **B** (T, optional) – The bias tensor for input gate. Concatenation of *[Wb[iofc], Rb[iofc]]*, and *[WBb[iofc], RBb[iofc]]* (if bidirectional) along dimension 0. This tensor has shape *[num_directions, 8*hidden_size]*. Optional: If not specified - assumed to be 0.
- **sequence_lens** (T1, optional) – Optional tensor specifying lengths of the sequences in a batch. If not specified - assumed all sequences in the batch to have length *seq_length*. It has shape *[batch_size]*.
- **initial_h** (T, optional) – Optional initial value of the hidden. If not specified - assumed to be 0. It has shape *[num_directions, batch_size, hidden_size]*.
- **initial_c** (T, optional) – Optional initial value of the cell. If not specified - assumed to be 0. It has shape *[num_directions, batch_size, hidden_size]*.
- **P** (T, optional) – The weight tensor for peepholes. Concatenation of *P[iof]* and *PB[iof]* (if bidirectional) along dimension 0. It has shape *[num_directions, 3*hidde_size]*. Optional: If not specified - assumed to be 0.

Node Outputs

- **Y** (T, optional) – A tensor that concats all the intermediate output values of the hidden. It has shape *[seq_length, num_directions, batch_size, hidden_size]*.

- **Y_h** (T, optional) – The last output value of the hidden. It has shape *[num_directions, batch_size, hidden_size]*.
- **Y_c** (T, optional) – The last output value of the cell. It has shape *[num_directions, batch_size, hidden_size]*.

Type Constraints

- **T** – `float16`, `float32`, `float64`
- **T1** – `int32`

Parameters

- **`name`** – the name of the node.
- **`activation_alpha`** (`Optional [List [float]]`, default=None) – Optional scaling values used by some activation functions. The values are consumed in the order of activation functions, for example (f, g, h) in LSTM. Default values are the same as of corresponding ONNX operators.For example with LeakyRelu, the default alpha is 0.01.
- **`activation_beta`** (`Optional [List [float]]`, default=None) – Optional scaling values used by some activation functions. The values are consumed in the order of activation functions, for example (f, g, h) in LSTM. Default values are the same as of corresponding ONNX operators.
- **`activations`** (`Optional [List [str]]`, default=None) – A list of 3 (or 6 if bidirectional) activation functions for input, output, forget, cell, and hidden. The activation functions must be one of the activation functions specified above. Optional: See the equations for default if not specified.
- **`clip`** (`Optional [float]`, default=None) – Cell clip threshold. Clipping bounds the elements of a tensor in the range of [-threshold, +threshold] and is applied to the input of activations. No clip if not specified.
- **`direction`** (`Optional [str]`, default='forward') – Specify if the RNN is forward, reverse, or bidirectional. Must be one of forward (default), reverse, or bidirectional.
- **`hidden_size`** (`Optional [int]`, default=None) – Number of neurons in the hidden layer
- **`input_forget`** (`Optional [int]`, default=0) – Couple the input and forget gates if 1.

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

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

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

`clip`
Cell clip threshold. Clipping bounds the elements of a tensor in the range of [-threshold, +threshold] and is applied to the input of activations. No clip if not specified.

direction
Specify if the RNN is forward, reverse, or bidirectional. Must be one of forward (default), reverse, or bidirectional.

hidden_size
Number of neurons in the hidden layer

input_forget
Couple the input and forget gates if 1.

class ONNXLabelEncoder(*name*, ***, *default_float=- 0.0*, *default_int64=- 1*, *default_string='_Unused'*, *keys_floats=None*, *keys_int64s=None*, *keys_strings=None*, *values_floats=None*, *values_int64s=None*, *values_strings=None*)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Maps each element in the input tensor to another value.
 The mapping is determined by the two parallel attributes, 'keys_*' and 'values_*' attribute. The i-th value in the specified 'keys_*' attribute would be mapped to the i-th value in the specified 'values_*' attribute. It implies that input's element type and the element type of the specified 'keys_*' should be identical while the output type is identical to the specified 'values_*' attribute. If an input element can not be found in the specified 'keys_*' attribute, the 'default_*' that matches the specified 'values_*' attribute may be used as its output value.
 Let's consider an example which maps a string tensor to an integer tensor. Assume and 'keys_strings' is ["Amy", "Sally"], 'values_int64s' is [5, 6], and 'default_int64' is '-1'. The input ["Dori", "Amy", "Amy", "Sally", "Sally"] would be mapped to [-1, 5, 5, 6, 6].
 Since this operator is an one-to-one mapping, its input and output shapes are the same. Notice that only one of 'keys_*'/'values_*' can be set.
 For key look-up, bit-wise comparison is used so even a float NaN can be mapped to a value in 'values_*' attribute.

Node Inputs

- **X** (T1, single) – Input data. It can be either tensor or scalar.

Node Outputs

- **Y** (T2, single) – Output data.

Type Constraints

- **T1** – `int64`, `float32`
- **T2** – `int64`, `float32`

Parameters

- **name** – the name of the node.
- **default_float** (`Optional` [`float`], default=-0.0) – A float.
- **default_int64** (`Optional` [`int`], default=-1) – An integer.
- **default_string** (`Optional` [`str`], default='_Unused') – A string.
- **keys_floats** (`Optional` [`List` [`float`]], default=None) – A list of floats.
- **keys_int64s** (`Optional` [`List` [`int`]], default=None) – A list of ints.
- **keys_strings** (`Optional` [`List` [`str`]], default=None) – A list of strings. One and only one of 'keys_*'s should be set.
- **values_floats** (`Optional` [`List` [`float`]], default=None) – A list of floats.
- **values_int64s** (`Optional` [`List` [`int`]], default=None) – A list of ints.
- **values_strings** (`Optional` [`List` [`str`]], default=None) – A list of strings. One and only one of 'value_*'s should be set.

default_float
A float.

default_int64
An integer.

default_string
A string.

keys_floats
A list of floats.

keys_int64s
A list of ints.

keys_strings
A list of strings. One and only one of 'keys_*'s should be set.

values_floats
A list of floats.

values_int64s
A list of ints.

values_strings
A list of strings. One and only one of 'value_*'s should be set.

class ONNXLeakyRelu(*name*, *, *alpha=0.009999999776482582*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **X** (T, single) – Input tensor

Node Outputs

- **Y** (T, single) – Output tensor

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters

- **name** – the name of the node.
- **alpha** (`Optional` [`float`], default=0.009999999776482582) – Coefficient of leakage.

alpha
Coefficient of leakage.

class ONNXLess(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

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

Node Inputs

- **A** (T, single) – First input operand for the logical operator.

- **B** (T, single) – Second input operand for the logical operator.

Node Outputs

- **C** (T1, single) – Result tensor.

Type Constraints

- **T** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`
- **T1** – `bool_`

Parameters **name** – the name of the node.

class ONNXLessOrEqual(*name*, *)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

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

Node Inputs

- **A** (T, single) – First input operand for the logical operator.
- **B** (T, single) – Second input operand for the logical operator.

Node Outputs

- **C** (T1, single) – Result tensor.

Type Constraints

- **T** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`
- **T1** – `bool_`

Parameters **name** – the name of the node.

class ONNXLinearClassifier(*name*, *, *classlabels_ints=None*, *classlabels_strings=None*, *coefficients*, *intercepts=None*, *multi_class=0*, *post_transform='NONE'*)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Linear classifier

Node Inputs

- **X** (T1, single) – Data to be classified.

Node Outputs

- **Y** (T2, single) – Classification outputs (one class per example).
- **Z** (Z_constraint, single) – Classification scores ([N,E] - one score for each class and example

Type Constraints

- **T1** – `float32`, `float64`, `int64`, `int32`
- **T2** – `int64`
- **Z_constraint** – `float32`

Parameters

- **name** – the name of the node.
- **classlabels_ints** (Optional [List [int]], default=None) – Class labels when using integer labels. One and only one 'classlabels' attribute must be defined.
- **classlabels_strings** (Optional [List [str]], default=None) – Class labels when using string labels. One and only one 'classlabels' attribute must be defined.
- **coefficients** (List [float]) – A collection of weights of the model(s).
- **intercepts** (Optional [List [float]], default=None) – A collection of intercepts.
- **multi_class** (Optional [int], default=0) – Indicates whether to do OvR or multinomial (0=OvR is the default).
- **post_transform** (Optional [str], default='NONE') – Indicates the transform to apply to the scores vector.
One of 'NONE,' 'SOFTMAX,' 'LOGISTIC,' 'SOFTMAX_ZERO,' or 'PROBIT'

classlabels_ints
Class labels when using integer labels. One and only one 'classlabels' attribute must be defined.

classlabels_strings
Class labels when using string labels. One and only one 'classlabels' attribute must be defined.

coefficients
A collection of weights of the model(s).

intercepts
A collection of intercepts.

multi_class
Indicates whether to do OvR or multinomial (0=OvR is the default).

post_transform
Indicates the transform to apply to the scores vector.
One of 'NONE,' 'SOFTMAX,' 'LOGISTIC,' 'SOFTMAX_ZERO,' or 'PROBIT'

class ONNXLinearRegressor(*name*, ***, *coefficients=None*, *intercepts=None*, *post_transform='NONE'*, *targets=1*)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Generalized linear regression evaluation.
 If targets is set to 1 (default) then univariate regression is performed.
 If targets is set to M then M sets of coefficients must be passed in as a sequence and M results will be output for each input n in N.
 The coefficients array is of length n, and the coefficients for each target are contiguous. Intercepts are optional but if provided must match the number of targets.

Node Inputs

- **X** (T, single) – Data to be regressed.

Node Outputs

- **Y** (Y_constraint, single) – Regression outputs (one per target, per example).

Type Constraints

- **T** – float32, float64, int64, int32
- **Y_constraint** – float32

Parameters

- **name** – the name of the node.

- **coefficients** (Optional [List [float]], default=None) – Weights of the model(s).
- **intercepts** (Optional [List [float]], default=None) – Weights of the intercepts, if used.
- **post_transform** (Optional [str], default='NONE') – Indicates the transform to apply to the regression output vector.
One of 'NONE,' 'SOFTMAX,' 'LOGISTIC,' 'SOFTMAX_ZERO,' or 'PROBIT'
- **targets** (Optional [int], default=1) – The total number of regression targets, 1 if not defined.

coefficients
Weights of the model(s).

intercepts
Weights of the intercepts, if used.

post_transform
Indicates the transform to apply to the regression output vector.
One of 'NONE,' 'SOFTMAX,' 'LOGISTIC,' 'SOFTMAX_ZERO,' or 'PROBIT'

targets
The total number of regression targets, 1 if not defined.

class ONNXLog(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **input** (T, single) – Input tensor

Node Outputs

- **output** (T, single) – The natural log of the input tensor computed element-wise

Type Constraints

- **T** – float16, float32, float64

Parameters **name** – the name of the node.

class ONNXLogSoftmax(*name*, *, *axis=1*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

The operator computes the logsoftmax (log of softmax) values for each layer in the batch of the given input.

The input does not need to explicitly be a 2D vector; rather, it will be coerced into one. For an arbitrary n-dimensional tensor input in [a_0, a_1, ..., a_{k-1}, a_k, ..., a_{n-1}] and k is the axis provided, then input will be coerced into a 2-dimensional tensor with dimensions [a_0 * ... * a_{k-1}, a_k * ... * a_{n-1}]. For the default case where axis=1, this means the input tensor will be coerced into a 2D tensor of dimensions [a_0, a_1 * ... * a_{n-1}], where a_0 is often the batch size. In this situation, we must have a_0 = N and a_1 * ... * a_{n-1} = D. Each of these dimensions must be matched correctly, or else the operator will throw errors. The output tensor has the same shape and contains the logsoftmax values of the corresponding input.

Node Inputs

- **input** (T, single) – The input tensor that's coerced into a 2D matrix of size (NxD) as described above.

Node Outputs

- **output** (T, single) – The output values with the same shape as input tensor (the original size without coercion).

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters

- **`name`** – the name of the node.
- **`axis`** (`Optional` [`int`], default=1) – Describes the axis of the inputs when coerced to 2D; defaults to one because the 0th axis most likely describes the batch_size. Negative value means counting dimensions from the back. Accepted range is [-r, r-1] where r = rank(input).

`axis`
Describes the axis of the inputs when coerced to 2D; defaults to one because the 0th axis most likely describes the batch_size. Negative value means counting dimensions from the back. Accepted range is [-r, r-1] where r = rank(input).

`class ONNXLpNormalization`(*name*, *, *axis=- 1*, *p=2*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Given a matrix, apply Lp-normalization along the provided axis.

Node Inputs

- **input** (T, single) – Input matrix

Node Outputs

- **output** (T, single) – Matrix after normalization

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters

- **`name`** – the name of the node.
- **`axis`** (`Optional` [`int`], default=-1) – The axis on which to apply normalization, -1 mean last axis.
- **`p`** (`Optional` [`int`], default=2) – The order of the normalization, only 1 or 2 are supported.

`axis`
The axis on which to apply normalization, -1 mean last axis.

`p`
The order of the normalization, only 1 or 2 are supported.

`class ONNXLpPool`(*name*, *, *auto_pad='NOTSET'*, *kernel_shape*, *p=2*, *pads=None*, *strides=None*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **X** (T, single) – Input data tensor from the previous operator; dimensions for image case are (N x C x H x W), where N is the batch size, C is the number of channels, and H and W are the height and the width of the data. For non image case, the dimensions are in the form of (N x C x D1 x D2 … Dn), where N is the batch size.

Node Outputs

- **Y** (T, single) – Output data tensor from Lp pooling across the input tensor. Dimensions will vary based on various kernel, stride, and pad sizes.

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters

- **`name`** – the name of the node.
- **`auto_pad`** (`Optional` [`str`], default='NOTSET') – auto_pad must be either NOTSET, SAME_UPPER, SAME_LOWER or VALID. Where default value is NOTSET, which means explicit padding is used. SAME_UPPER or SAME_LOWER mean pad the input so that the output spatial size match the input.In case of odd number add the extra padding at the end for SAME_UPPER and at the beginning for SAME_LOWER. VALID mean no padding.
- **`kernel_shape`** (`List` [`int`]) – The size of the kernel along each axis.
- **`p`** (`Optional` [`int`], default=2) – p value of the Lp norm used to pool over the input data.
- **`pads`** (`Optional` [`List` [`int`]], default=None) – Padding for the beginning and ending along each spatial axis, it can take any value greater than or equal to 0. The value represent the number of pixels added to the beginning and end part of the corresponding axis. *pads* format should be as follow [x1_begin, x2_begin…x1_end, x2_end,…], where xi_begin the number of pixels added at the beginning of axis *i* and xi_end, the number of pixels added at the end of axis *i*. This attribute cannot be used simultaneously with auto_pad attribute. If not present, the padding defaults to 0 along start and end of each spatial axis.
- **`strides`** (`Optional` [`List` [`int`]], default=None) – Stride along each spatial axis. If not present, the stride defaults to 1 along each spatial axis.

`auto_pad`
auto_pad must be either NOTSET, SAME_UPPER, SAME_LOWER or VALID. Where default value is NOTSET, which means explicit padding is used. SAME_UPPER or SAME_LOWER mean pad the input so that the output spatial size match the input.In case of odd number add the extra padding at the end for SAME_UPPER and at the beginning for SAME_LOWER. VALID mean no padding.

`kernel_shape`
The size of the kernel along each axis.

`p`
p value of the Lp norm used to pool over the input data.

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

strides
Stride along each spatial axis. If not present, the stride defaults to 1 along each spatial axis.

class ONNXMatMul(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Matrix product that behaves like numpy.matmul: https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.matmul.html

Node Inputs

- **A** (T, single) – N-dimensional matrix A
- **B** (T, single) – N-dimensional matrix B

Node Outputs

- **Y** (T, single) – Matrix multiply results from A * B

Type Constraints

- **T** – `float16`, `float32`, `float64`, `uint32`, `uint64`, `int32`, `int64`

Parameters **name** – the name of the node.

class ONNXMatMulInteger(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Matrix product that behaves like numpy.matmul: https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.matmul.html. The production MUST never overflow. The accumulation may overflow if and only if in 32 bits.

Node Inputs

- **A** (T1, single) – N-dimensional matrix A
- **B** (T2, single) – N-dimensional matrix B
- **a_zero_point** (T1, optional) – Zero point tensor for input 'A'. It's optional and default value is 0. It could be a scalar or a 1-D tensor, which means a per-tensor or per-row quantization. If it's a 1-D tensor, its number of elements should be equal to the number of rows of input 'A'.
- **b_zero_point** (T2, optional) – Zero point tensor for input 'B'. It's optional and default value is 0. It could be a scalar or a 1-D tensor, which means a per-tensor or per-column quantization. If it's a 1-D tensor, its number of elements should be equal to the number of columns of input 'B'.

Node Outputs

- **Y** (T3, single) – Matrix multiply results from A * B

Type Constraints

- **T1** – `int8`, `uint8`
- **T2** – `int8`, `uint8`
- **T3** – `int32`

Parameters **name** – the name of the node.

class ONNXMax(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **data_0** (T, variadic) – List of tensors for max.

Node Outputs

- **max** (T, single) – Output tensor.

Type Constraints

- **T** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`

Parameters **`name`** – the name of the node.

`class ONNXMaxPool`(*name*, ***, *auto_pad='NOTSET'*, *ceil_mode=0*, *dilations=None*, *kernel_shape*, *pads=None*, *storage_order=0*, *strides=None*)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

MaxPool consumes an input tensor X and applies max pooling across the tensor according to kernel sizes, stride sizes, and pad lengths. max pooling consisting of computing the max on all values of a subset of the input tensor according to the kernel size and downsampling the data into the output tensor Y for further processing. The output spatial shape will be following: ` output_spatial_shape[i] = floor((input_spatial_shape[i] + pad_shape[i] - ((kernel_spatial_shape[i] - 1) * dilations[i] + 1)) / strides_spatial_shape[i] + 1) ` or ` output_spatial_shape[i] = ceil((input_spatial_shape[i] + pad_shape[i] - ((kernel_spatial_shape[i] - 1) * dilations[i] + 1)) / strides_spatial_shape[i] + 1) ` if ceil_mode is enabled

` * pad_shape[i] is sum of pads along axis i `

auto_pad is a DEPRECATED attribute. If you are using them currently, the output spatial shape will be following: ` VALID: output_spatial_shape[i] = ceil((input_spatial_shape[i] - ((kernel_spatial_shape[i] - 1) * dilations[i] + 1) + 1) / strides_spatial_shape[i]) SAME_UPPER or SAME_LOWER: output_spatial_shape[i] = ceil(input_spatial_shape[i] / strides_spatial_shape[i]) ` And pad shape will be following if *SAME_UPPER* or *SAME_LOWER*: ` pad_shape[i] = (output_spatial_shape[i] - 1) * strides_spatial_shape[i] + ((kernel_spatial_shape[i] - 1) * dilations[i] + 1) - input_spatial_shape[i] ` The output of each pooling window is maximum number of elements exclude pad.

Node Inputs

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

Node Outputs

- **Y** (T, single) – Output data tensor from average or max pooling across the input tensor. Dimensions will vary based on various kernel, stride, and pad sizes. Floor value of the dimension is used

- **Indices** (I, optional) – Indices tensor from max pooling across the input tensor. The dimensions of indices are the same as output tensor. The values in indices of are the indices of the selected values during pooling. The indices are computed as flatten 1-D tensor, and the indices do not consider padding. So the values in indices are in [0, N x C x D1 x ... x Dn).

Type Constraints

- **T** – `float16`, `float32`, `float64`, `int8`, `uint8`
- **I** – `int64`

Parameters

- **`name`** – the name of the node.
- **`auto_pad`** (`Optional [str]`, default='NOTSET') – auto_pad must be either NOTSET, SAME_UPPER, SAME_LOWER or VALID. Where default value is NOTSET, which means explicit padding is used. SAME_UPPER or SAME_LOWER mean pad the input so that the output spatial size match the input.In case of odd number add the extra padding at the end for SAME_UPPER and at the beginning for SAME_LOWER. VALID mean no padding.
- **`ceil_mode`** (`Optional [int]`, default=0) – Whether to use ceil or floor (default) to compute the output shape.
- **`dilations`** (`Optional [List [int]]`, default=None) – Dilation value along each spatial axis of filter. If not present, the dilation defaults to 1 along each spatial axis.
- **`kernel_shape`** (`List [int]`) – The size of the kernel along each axis.
- **`pads`** (`Optional [List [int]]`, default=None) – Padding for the beginning and ending along each spatial axis, it can take any value greater than or equal to 0. The value represent the number of pixels added to the beginning and end part of the corresponding axis. *pads* format should be as follow [x1_begin, x2_begin...x1_end, x2_end,...], where xi_begin the number of pixels added at the beginning of axis *i* and xi_end, the number of pixels added at the end of axis *i*. This attribute cannot be used simultaneously with auto_pad attribute. If not present, the padding defaults to 0 along start and end of each spatial axis.
- **`storage_order`** (`Optional [int]`, default=0) – The storage order of the tensor. 0 is row major, and 1 is column major.
- **`strides`** (`Optional [List [int]]`, default=None) – Stride along each spatial axis. If not present, the stride defaults to 1 along each spatial axis.

`auto_pad`
auto_pad must be either NOTSET, SAME_UPPER, SAME_LOWER or VALID. Where default value is NOTSET, which means explicit padding is used. SAME_UPPER or SAME_LOWER mean pad the input so that the output spatial size match the input.In case of odd number add the extra padding at the end for SAME_UPPER and at the beginning for SAME_LOWER. VALID mean no padding.

`ceil_mode`
Whether to use ceil or floor (default) to compute the output shape.

`dilations`
Dilation value along each spatial axis of filter. If not present, the dilation defaults to 1 along each spatial axis.

`kernel_shape`
The size of the kernel along each axis.

`pads`
Padding for the beginning and ending along each spatial axis, it can take any value greater than or equal to

0. The value represent the number of pixels added to the beginning and end part of the corresponding axis. *pads* format should be as follow [x1_begin, x2_begin...x1_end, x2_end,...], where xi_begin the number of pixels added at the beginning of axis *i* and xi_end, the number of pixels added at the end of axis *i*. This attribute cannot be used simultaneously with auto_pad attribute. If not present, the padding defaults to 0 along start and end of each spatial axis.

storage_order
The storage order of the tensor. 0 is row major, and 1 is column major.

strides
Stride along each spatial axis. If not present, the stride defaults to 1 along each spatial axis.

class ONNXMaxRoiPool(*name*, *, *pooled_shape*, *spatial_scale=1.0*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **X** (T, single) – Input data tensor from the previous operator; dimensions for image case are (N x C x H x W), where N is the batch size, C is the number of channels, and H and W are the height and the width of the data.
- **rois** (T, single) – 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], ...].

Node Outputs

- **Y** (T, single) – RoI pooled output 4-D tensor of shape (num_rois, channels, pooled_shape[0], pooled_shape[1]).

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters

- **name** – the name of the node.
- **pooled_shape** (`List [int]`) – ROI pool output shape (height, width).
- **spatial_scale** (`Optional [float]`, default=1.0) – Multiplicative spatial scale factor to translate ROI coordinates from their input scale to the scale used when pooling.

pooled_shape
ROI pool output shape (height, width).

spatial_scale
Multiplicative spatial scale factor to translate ROI coordinates from their input scale to the scale used when pooling.

class ONNXMaxUnpool(*name*, *, *kernel_shape*, *pads=None*, *strides=None*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

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

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

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

Node Inputs

- **X** (T1, single) – Input data tensor that has to be unpooled. This tensor is typically the first output of the MaxPool op.Dimensions for image case are (N x C x H x W), where N is the batch size, C is the number of channels, and H and W are the height and the width of the data. For non-image case, the dimensions are in the form of (N x C x D1 x D2 … Dn), where N is the batch size. Optionally, if dimension denotation is in effect, the operation expects the input data tensor to arrive with the dimension denotation of [DATA_BATCH, DATA_CHANNEL, DATA_FEATURE, DATA_FEATURE …].
- **I** (T2, single) – Input data tensor containing the indices corresponding to elements in the first input tensor X.This tensor is typically the second output of the MaxPool op.Dimensions must be the same as input tensor X. The indices are linear, i.e. computed considering the tensor as flattened 1-D tensor, assuming row-major storage. Also, the linear indices should not consider padding. So the values in indices are in the range [0, N x C x D1 x … x Dn).
- **output_shape** (T2, optional) – The shape of the output can be explicitly set which will cause pads values to be auto generated. If 'output_shape' is specified, 'pads' values are ignored.

Node Outputs

- **output** (T1, single) – Output data tensor that contains the result of the unpooling.

Type Constraints

- **T1** – `float16`, `float32`, `float64`
- **T2** – `int64`

Parameters

- **`name`** – the name of the node.
- **`kernel_shape`** (`List [int]`) – The size of the kernel along each axis.
- **`pads`** (`Optional [List [int]]`, default=None) – Padding for the beginning and ending along each spatial axis, it can take any value greater than or equal to 0. The value represent the number of pixels added to the beginning and end part of the corresponding axis. *pads* format should be as follow [x1_begin, x2_begin…x1_end, x2_end,…], where xi_begin the number of pixels added at the beginning of axis *i* and xi_end, the number of pixels added at the end of axis *i*. This attribute cannot be used simultaneously with auto_pad attribute. If not present, the padding defaults to 0 along start and end of each spatial axis.
- **`strides`** (`Optional [List [int]]`, default=None) – Stride along each spatial axis. If not present, the stride defaults to 1 along each spatial axis.

`kernel_shape`
The size of the kernel along each axis.

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

strides
Stride along each spatial axis. If not present, the stride defaults to 1 along each spatial axis.

class ONNXMean(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **data_0** (T, variadic) – List of tensors for mean.

Node Outputs

- **mean** (T, single) – Output tensor.

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters **name** – the name of the node.

class ONNXMeanVarianceNormalization(*name*, *, *axes=[0, 2, 3]*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **X** (T, single) – Input tensor

Node Outputs

- **Y** (T, single) – Output tensor

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters

- **name** – the name of the node.
- **axes** (`Optional [List [int]]`, default=[0, 2, 3]) – A list of integers, along which to reduce. The default is to caculate along axes [0,2,3] for calculating mean and variance along each channel. Two variables with the same C-coordinate are associated with the same mean and variance.

axes
A list of integers, along which to reduce. The default is to caculate along axes [0,2,3] for calculating mean and variance along each channel. Two variables with the same C-coordinate are associated with the same mean and variance.

class ONNXMin(*name*, *)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **data_0** (T, variadic) – List of tensors for min.

Node Outputs

- **min** (T, single) – Output tensor.

Type Constraints

- **T** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`

Parameters **name** – the name of the node.

class ONNXMod(*name*, *, *fmod=0*)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Performs element-wise binary modulus (with Numpy-style broadcasting support). The sign of the remainder is the same as that of the Divisor.

Mod operator can also behave like C fmod() or numpy.fmod. In this case, the sign of the remainder however, will be the same as the Dividend (in contrast to integer mod). To force a behavior like numpy.fmod() an 'fmod' Attribute is provided. This attribute is set to 0 by default causing the behavior to be like integer mod. Setting this attribute to 1 causes the remainder to be calculated similar to that of numpy.fmod().

If the input type is floating point, then *fmod* attribute must be set to 1.

In case of dividend being zero, the results will be platform dependent.

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

Node Inputs

- **A** (T, single) – Dividend tensor
- **B** (T, single) – Divisor tensor

Node Outputs

- **C** (T, single) – Remainder tensor

Type Constraints

- **T** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`

Parameters

- **name** – the name of the node.
- **fmod** (`Optional` [`int`], default=0) – Whether the operator should behave like fmod (default=0 meaning it will do integer mods); Set this to 1 to force fmod treatment

fmod
Whether the operator should behave like fmod (default=0 meaning it will do integer mods); Set this to 1 to force fmod treatment

class ONNXMomentum(*name*, *, *alpha*, *beta*, *mode*, *norm_coefficient*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Compute one iteration of stochastic gradient update with momentum. This operator can conduct the optimization of multiple tensor variables.

Let's define the behavior of this operator. As you can imagine, SG with momentum requires several parameters:

- The learning-rate "R".
- The update count "T". That is, the number of conducted training iterations. It should be zero in the first training iteration.
- A L2-norm regularization coefficient "norm_coefficient".
- A decay coefficient of previous accumulated gradient (i.e., momentum) "alpha".
- The scaling coefficient of current gradient "beta".
- An attribute to choose either standard momentum or Nesterov's momentum "mode" should be used.

For the sake of simplicity, assume that there is only one tensor (called "X") to be optimized. Other necessary inputs are "X"'s gradient (called "G") and "X"'s momentum (called "V"). This Momentum operator maps all these inputs to the new value of "X" (called "X_new") and its new momentum (called "V_new").

This operator supports two different momentum algorithms. Set the attribute "mode" to "nesterov" if Nesterov's momentum is desired. Otherwise, set the attribute "model" to "standard" to use standard momentum. Computation details are described subsequently.

Let "+", "-", "*", and "/" are all element-wise operations with numpy-style broadcasting.

Pseudo code for SG with standard momentum:

// Add gradient of 0.5 * norm_coefficient * ||X||^2, where ||X|| is the sum of squared // values of all elements in X. G_regularized = norm_coefficient * X + G

// In the first training iteration, beta should always be 1. beta_adjusted = T > 0 ? beta : 1

// Compute the current momentum based on previous momentum and the current gradient. V_new = alpha * V + beta_adjusted * G_regularized

// Update X. X_new = X - R * V_new

Pseudo code for SG with Nesterov's momentum:

// Add gradient of 0.5 * norm_coefficient * ||X||^2, where ||X|| is the sum of squared // values of all elements in X. G_regularized = norm_coefficient * X + G;

// In the first training iteration, beta should always be 1. beta_adjusted = T > 0 ? beta : 1

// Compute the current momentum based on previous momentum and the current gradient. V_new = alpha * V + beta_adjusted * G_regularized;

// Compute final update direction and then update X. X_new = X - R * (G_regularized + alpha * V_new)

If one assign this operators to optimize multiple inputs, for example, "X_1" and "X_2". The same pseudo code would be extended to handle all tensors jointly. More specifically, we can view "X" as

a concatenation of "X_1" and "X_2" (of course, their gradient and accumulate gradient should be concatenated too) and then our pseudo code becomes applicable.

Node Inputs

- **R** (T1, single) – The learning rate.
- **T** (T2, single) – Update count of "X". It should be a scalar.
- **inputs** (T3, variadic) – It sequentially contains the current values of optimized tensors, then their gradient tensors, and finally their momentum tensors. For example, if two tensors "X_1" and "X_2" are optimized, The expected input list would be ["X_1", "X_2", gradient of "X_1", gradient of "X_2", momentum of "X_1", momentum of "X_2"].

Node Outputs

- **outputs** (T3, variadic) – It sequentially contains the new values of optimized tensors and then the new values of their momentum tensors. For example, if two tensors "X_1" and "X_2" are optimized, the output list would be [new value of "X_1," new value of "X_2" new momentum of "X_1", new momentum of "X_2"].

Type Constraints

- **T1** – `float32`, `float64`
- **T2** – `int64`
- **T3** – `float32`, `float64`

Parameters

- **`name`** – the name of the node.
- **`alpha`** (`float`) – The decay factor of momentum. It should be a scalar.
- **`beta`** (`float`) – The coefficient of gradient in computing new momentum. It should be a scalar.
- **`mode`** (`str`) – Its value should be either "nesterov" or "standard". The value "nesterov" leads to the use of Nesterov's momentum while "standard" invokes stochastic gradient method using standard momentum
- **`norm_coefficient`** (`float`) – Coefficient of 0.5 * norm_coefficient * ||X||^2.

`alpha`
The decay factor of momentum. It should be a scalar.

`beta`
The coefficient of gradient in computing new momentum. It should be a scalar.

`mode`
Its value should be either "nesterov" or "standard". The value "nesterov" leads to the use of Nesterov's momentum while "standard" invokes stochastic gradient method using standard momentum

`norm_coefficient`
Coefficient of 0.5 * norm_coefficient * ||X||^2.

`class ONNXMul`(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

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

Node Inputs

- **A** (T, single) – First operand.
- **B** (T, single) – Second operand.

Node Outputs

- **C** (T, single) – Result, has same element type as two inputs

Type Constraints

- **T** – `uint32`, `uint64`, `int32`, `int64`, `float16`, `float32`, `float64`

Parameters **`name`** – the name of the node.

`class ONNXMultinomial`(*name*, ***, *dtype=6*, *sample_size=1*, *seed=None*)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **input** (T1, single) – Input tensor with shape [batch_size, class_size], where class_size is the number of all possible outcomes. Each value along the axis zero represents the unnormalized log-probability of each corresponding outcome in a batch.

Node Outputs

- **output** (T2, single) – Output tensor with shape [batch_size, sample_size], where sample_size is the number of times to sample. Each value along the axis zero represents the outcome of the corresponding sample in a batch.

Type Constraints

- **T1** – `float16`, `float32`, `float64`
- **T2** – `int32`, `int64`

Parameters

- **`name`** – the name of the node.
- **`dtype`** (`Optional` [`int`], default=6) – (Optional) The data type for the elements of the output tensor, if not specified, we will use int32.
- **`sample_size`** (`Optional` [`int`], default=1) – Number of times to sample.
- **`seed`** (`Optional` [`float`], default=None) – (Optional) Seed to the random generator, if not specified we will auto generate one.

`dtype`
(Optional) The data type for the elements of the output tensor, if not specified, we will use int32.

`sample_size`
Number of times to sample.

`seed`
(Optional) Seed to the random generator, if not specified we will auto generate one.

`class ONNXNeg`(*name*, ***)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **X** (T, single) – Input tensor

Node Outputs

- **Y** (T, single) – Output tensor

Type Constraints

- **T** – `float32`, `int32`, `int8`, `int16`, `int64`, `float16`, `float64`

Parameters **`name`** – the name of the node.

`class ONNXNegativeLogLikelihoodLoss`(*name*, ***, *ignore_index=None*, *reduction='mean'*)

Bases: *`daceml.onnx.nodes.onnx_op.ONNXOp`*

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

loss[n][d_1][d_2]. . . [d_k] = -input[n][c][d_1][d_2]. . . [d_k].

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

loss[n][d_1][d_2]. . . [d_k] = -input[n][c][d_1][d_2]. . . [d_k] * weight[c].

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

loss[n][d_1][d_2]. . . [d_k] = 0, when target[n][d_1][d_2]. . . [d_k] = ignore_index

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

mean(loss), if "weight" is not provided,

or if weight is provided,

sum(loss) / sum(weight[target[n][d_1][d_2]. . . [d_k]]]), for all samples.

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

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

Example 1:

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

[[0.0, 1.0], [2.0, 2.0], [1.0, 2]]]

target = [[2, 1], [0, 2]]

loss = np.zeros((N, d1)) for n in range(N):

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

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

Example 2:

// weighted negative log likelihood loss, sum reduction N, C, d1 = 2, 3, 2 input = [[[1.0, 2.0], [2.0, 2.0], [3.0, 2.0]],

[[0.0, 1.0], [2.0, 2.0], [1.0, 2]]]

target = [[2, 1], [0, 2]] weight = [0.2, 0.3, 0.1] loss = np.zeros((N, d1)) for n in range(N):

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

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

Example 3:

// weighted negative log likelihood loss, mean reduction N, C, d1 = 2, 3, 2 input = [[[1.0, 2.0], [2.0, 2.0], [3.0, 2.0]],

[[0.0, 1.0], [2.0, 2.0], [1.0, 2]]]

target = [[2, 1], [0, 2]] weight = [0.2, 0.3, 0.1] loss = np.zeros((N, d1)) weight_total = 0 for n in range(N):

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

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

Node Inputs

- **input** (T, single) – Input tensor of shape (N, C) or (N, C, d1, d2, . . . , dk).
- **target** (Tind, single) – Target tensor of shape (N) or (N, d1, d2, . . . , dk). Target element value shall be in range of [0, C). If ignore_index is specified, it may have a value outside [0, C) and the target values should either be in the range [0, C) or have the value ignore_index.
- **weight** (T, optional) – Optional rescaling weight tensor. If given, it has to be a tensor of size C. Otherwise, it is treated as if having all ones.

Node Outputs

- **loss** (T, single) – The negative log likelihood loss

Type Constraints

- **T** – `float16`, `float32`, `float64`
- **Tind** – `int32`, `int64`

Parameters

- **`name`** – the name of the node.
- **`ignore_index`** (`Optional [int]`, default=None) – Specifies a target value that is ignored and does not contribute to the input gradient. It's an optional value.
- **`reduction`** (`Optional [str]`, default='mean') – Type of reduction to apply to loss: none, sum, mean (default). 'none': the output is the loss for each sample. 'sum': the output will be summed. 'mean': the sum of the output will be divided by the sum of applied weights.

`ignore_index`
Specifies a target value that is ignored and does not contribute to the input gradient. It's an optional value.

`reduction`
Type of reduction to apply to loss: none, sum, mean (default). 'none': the output is the loss for each sample. 'sum': the output will be summed. 'mean': the sum of the output will be divided by the sum of applied weights.

class ONNXNonMaxSuppression(*name*, *, *center_point_box=0*)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **boxes** (boxes_constraint, single) – An input tensor with shape [num_batches, spatial_dimension, 4]. The single box data format is indicated by center_point_box.
- **scores** (scores_constraint, single) – An input tensor with shape [num_batches, num_classes, spatial_dimension]
- **max_output_boxes_per_class** (max_output_boxes_per_class_constraint, optional) – Integer representing the maximum number of boxes to be selected per batch per class. It is a scalar. Default to 0, which means no output.
- **iou_threshold** (iou_threshold_constraint, optional) – Float representing the threshold for deciding whether boxes overlap too much with respect to IOU. It is scalar. Value range [0, 1]. Default to 0.
- **score_threshold** (score_threshold_constraint, optional) – Float representing the threshold for deciding when to remove boxes based on score. It is a scalar.

Node Outputs

- **selected_indices** (selected_indices_constraint, single) – selected indices from the boxes tensor. [num_selected_indices, 3], the selected index format is [batch_index, class_index, box_index].

Type Constraints

- **boxes_constraint** – `float32`
- **scores_constraint** – `float32`
- **max_output_boxes_per_class_constraint** – `int64`
- **iou_threshold_constraint** – `float32`
- **score_threshold_constraint** – `float32`
- **selected_indices_constraint** – `int64`

Parameters

- **name** – the name of the node.
- **center_point_box** (`Optional [int]`, default=0) – Integer indicate the format of the box data. The default is 0. 0 - the box data is supplied as [y1, x1, y2, x2] where (y1, x1) and (y2, x2) are the coordinates of any diagonal pair of box corners and the coordinates can be provided as normalized (i.e., lying in the interval [0, 1]) or absolute. Mostly used for TF models. 1 - the box data is supplied as [x_center, y_center, width, height]. Mostly used for Pytorch models.

center_point_box

Integer indicate the format of the box data. The default is 0. 0 - the box data is supplied as [y1, x1, y2,

x2] where (y1, x1) and (y2, x2) are the coordinates of any diagonal pair of box corners and the coordinates can be provided as normalized (i.e., lying in the interval [0, 1]) or absolute. Mostly used for TF models. 1 - the box data is supplied as [x_center, y_center, width, height]. Mostly used for Pytorch models.

class ONNXNonZero(*name*, *)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Returns the indices of the elements that are non-zero (in row-major order - by dimension). NonZero behaves similar to numpy.nonzero: https://docs.scipy.org/doc/numpy/reference/generated/numpy.nonzero.html

Node Inputs

- **X** (T, single) – input

Node Outputs

- **Y** (Y_constraint, single) – output

Type Constraints

- **T** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`, `bool_`, `complex64`, `complex128`
- **Y_constraint** – `int64`

Parameters **name** – the name of the node.

class ONNXNormalizer(*name*, *, *norm='MAX'*)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Normalize the input. There are three normalization modes, which have the corresponding formulas, defined using element-wise infix operators '/' and '^' and tensor-wide functions 'max' and 'sum':

**
** Max: Y = X / max(X)
 L1: Y = X / sum(X)
 L2: Y = sqrt(X^2 / sum(X^2)}
 In all modes, if the divisor is zero, Y == X.

**
** For batches, that is, [N,C] tensors, normalization is done along the C axis. In other words, each row of the batch is normalized independently.

Node Inputs

- **X** (T, single) – Data to be encoded, a tensor of shape [N,C] or [C]

Node Outputs

- **Y** (Y_constraint, single) – Encoded output data

Type Constraints

- **T** – `float32`, `float64`, `int64`, `int32`
- **Y_constraint** – `float32`

Parameters

- **name** – the name of the node.
- **norm** (`Optional` [`str`], default='MAX') – One of 'MAX,' 'L1,' 'L2'

norm

One of 'MAX,' 'L1,' 'L2'

class ONNXNot(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Returns the negation of the input tensor element-wise.

Node Inputs

- **X** (T, single) – Input tensor

Node Outputs

- **Y** (T, single) – Output tensor

Type Constraints

- **T** – `bool_`

Parameters **name** – the name of the node.

class ONNXOneHot(*name*, *, *axis=- 1*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

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

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

Node Inputs

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

Node Outputs

- **output** (T3, single) – Tensor of rank one greater than input tensor 'indices', i.e. rank(output) = rank(indices) + 1. The data type for the elements of the output tensor is the same as the type of input 'values' is used.

Type Constraints

- **T1** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`
- **T2** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`
- **T3** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`, `bool_`, `complex64`, `complex128`

Parameters

- **name** – the name of the node.
- **axis** (`Optional [int]`, default=-1) – (Optional) Axis along which one-hot representation in added. Default: axis=-1. axis=-1 means that the additional dimension will be inserted as the innermost/last dimension in the output tensor. Negative value means counting dimensions from the back. Accepted range is [-r-1, r] where r = rank(indices).

axis

(Optional) Axis along which one-hot representation in added. Default: axis=-1. axis=-1 means that the additional dimension will be inserted as the innermost/last dimension in the output tensor. Negative value means counting dimensions from the back. Accepted range is [-r-1, r] where r = rank(indices).

class ONNXOneHotEncoder(*name*, ***, *cats_int64s=None*, *cats_strings=None*, *zeros=1*)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Replace each input element with an array of ones and zeros, where a single one is placed at the index of the category that was passed in. The total category count will determine the size of the extra dimension of the output array Y.
 For example, if we pass a tensor with a single value of 4, and a category count of 8, the output will be a tensor with `[0,0,0,0,1,0,0,0]`.
 This operator assumes every input feature is from the same set of categories.
 If the input is a tensor of float, int32, or double, the data will be cast to integers and the cats_int64s category list will be used for the lookups.

Node Inputs

- **X** (T, single) – Data to be encoded.

Node Outputs

- **Y** (Y_constraint, single) – Encoded output data, having one more dimension than X.

Type Constraints

- **T** – `int64`, `int32`, `float32`, `float64`
- **Y_constraint** – `float32`

Parameters

- **name** – the name of the node.
- **cats_int64s** (`Optional [List [int]]`, default=None) – List of categories, ints.
One and only one of the 'cats_*' attributes must be defined.
- **cats_strings** (`Optional [List [str]]`, default=None) – List of categories, strings.
One and only one of the 'cats_*' attributes must be defined.
- **zeros** (`Optional [int]`, default=1) – If true and category is not present, will return all zeros; if false and a category if not found, the operator will fail.

cats_int64s

List of categories, ints.
One and only one of the 'cats_*' attributes must be defined.

cats_strings
List of categories, strings.
One and only one of the 'cats_*' attributes must be defined.

zeros
If true and category is not present, will return all zeros; if false and a category if not found, the operator will fail.

class ONNXOr(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

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

Node Inputs

- **A** (T, single) – First input operand for the logical operator.
- **B** (T, single) – Second input operand for the logical operator.

Node Outputs

- **C** (T1, single) – Result tensor.

Type Constraints

- **T** – `bool_`
- **T1** – `bool_`

Parameters **name** – the name of the node.

class ONNXPRelu(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

PRelu takes input data (Tensor<T>) and slope tensor as input, and produces one output data (Tensor<T>) where the function *f(x) = slope * x for x < 0*, *f(x) = x for x >= 0.*, is applied to the data tensor elementwise. This operator supports **unidirectional broadcasting** (tensor slope should be unidirectional broadcastable to input tensor X); for more details please check [the doc](Broadcasting.md).

Node Inputs

- **X** (T, single) – Input tensor
- **slope** (T, single) – Slope tensor. The shape of slope can be smaller then first input X; if so, its shape must be unidirectional broadcastable to X

Node Outputs

- **Y** (T, single) – Output tensor (same size as X)

Type Constraints

- **T** – `float16`, `float32`, `float64`, `uint32`, `uint64`, `int32`, `int64`

Parameters **name** – the name of the node.

class ONNXPad(*name*, *, *mode='constant'*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

The three supported *modes* are (similar to corresponding modes supported by *numpy.pad*):

1) *constant`(default) - pads with a given constant value as specified by `constant_value* (which defaults to 0)
2) *reflect* - pads with the reflection of the vector mirrored on the first and last values of the vector along each axis
3) *edge* - pads with the edge values of array

Example 1 (*constant* mode): Insert 0 pads to the beginning of the second dimension.

data = [

[1.0, 1.2], [2.3, 3.4], [4.5, 5.7],

]

pads = [0, 2, 0, 0]

mode = 'constant'

constant_value = 0.0

output = [

[[0.0, 0.0, 1.0, 1.2], [0.0, 0.0, 2.3, 3.4], [0.0, 0.0, 4.5, 5.7],

],

]

Example 2 (*reflect* mode): data = [

[1.0, 1.2], [2.3, 3.4], [4.5, 5.7],

]

pads = [0, 2, 0, 0]

mode = 'reflect'

output = [

[[1.0, 1.2, 1.0, 1.2], [2.3, 3.4, 2.3, 3.4], [4.5, 5.7, 4.5, 5.7],

],

]

Example 3 (*edge* mode): data = [

[1.0, 1.2], [2.3, 3.4], [4.5, 5.7],

]

pads = [0, 2, 0, 0]

mode = 'edge'

output = [

[[1.0, 1.0, 1.0, 1.2], [2.3, 2.3, 2.3, 3.4], [4.5, 4.5, 4.5, 5.7],

],

]

Node Inputs

- **data** (T, single) – Input tensor.

- **pads** (pads_constraint, single) – Tensor of integers indicating the number of padding elements to add or remove (if negative) at the beginning and end of each axis. For 2D input tensor, it is the number of pixels. *pads* should be a 1D tensor of shape [2 * input_rank]. *pads* format should be: [x1_begin, x2_begin,...,x1_end, x2_end,...], where xi_begin is the number of pad values added at the beginning of axis *i* and xi_end, the number of pad values added at the end of axis *i*.
- **constant_value** (T, optional) – (Optional) A scalar value to be used if the mode chosen is *constant* (by default it is 0).

Node Outputs

- **output** (T, single) – Tensor after padding.

Type Constraints

- **T** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`
- **pads_constraint** – `int64`

Parameters

- **name** – the name of the node.
- **mode** (`Optional [str]`, default='constant') – Supported modes: *constant`(default), `reflect*, *edge*

mode
Supported modes: *constant`(default), `reflect*, *edge*

class ONNXPow(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Pow takes input data (Tensor<T>) and exponent Tensor, and produces one output data (Tensor<T>) where the function *f(x) = x^exponent*, is applied to the data tensor elementwise. This operator supports **multidirectional (i.e., Numpy-style) broadcasting**; for more details please check [the doc](Broadcasting.md).

Node Inputs

- **X** (T, single) – First operand, base of the exponent.
- **Y** (T1, single) – Second operand, power of the exponent.

Node Outputs

- **Z** (T, single) – Output tensor (same size as X)

Type Constraints

- **T** – `int32`, `int64`, `float16`, `float32`, `float64`
- **T1** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`

Parameters **name** – the name of the node.

class ONNXQLinearConv(*name*, *, *auto_pad='NOTSET'*, *dilations=None*, *group=1*, *kernel_shape=None*, *pads=None*, *strides=None*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

The convolution operator consumes a quantized input tensor, its scale and zero point, a quantized filter, its scale and zero point, and output's scale and zero point, and computes the quantized output. Each scale and zero-point pair must have same shape. It means they must be either scalars (per tensor) or 1-D tensors (per output channel).

Each input or output and its related zero point must have same type. When bias is present it must be quantized using scale = input scale * weight scale and zero point as 0.

Node Inputs

- **x** (T1, single) – Input data tensor from previous layer; has size (N x C x H x W), where N is the batch size, C is the number of channels, and H and W are the height and width. Note that this is for the 2D image. Otherwise the size is (N x C x D1 x D2 … x Dn). Optionally, if dimension denotation is in effect, the operation expects input data tensor to arrive with the dimension denotation of [DATA_BATCH, DATA_CHANNEL, DATA_FEATURE, DATA_FEATURE …].
- **x_scale** (x_scale_constraint, single) – Scale tensor for input 'x'. It's a scalar, which means a per-tensor/layer quantization.
- **x_zero_point** (T1, single) – Zero point tensor for input 'x'. It's a scalar, which means a per-tensor/layer quantization.
- **w** (T2, single) – The weight tensor that will be used in the convolutions; has size (M x C/group x kH x kW), where C is the number of channels, and kH and kW are the height and width of the kernel, and M is the number of feature maps. For more than 2 dimensions, the kernel shape will be (M x C/group x k1 x k2 x … x kn), where (k1 x k2 x … kn) is the dimension of the kernel. Optionally, if dimension denotation is in effect, the operation expects the weight tensor to arrive with the dimension denotation of [FILTER_OUT_CHANNEL, FILTER_IN_CHANNEL, FILTER_SPATIAL, FILTER_SPATIAL …]. X.shape[1] == (W.shape[1] * group) == C (assuming zero based indices for the shape array). Or in other words FILTER_IN_CHANNEL should be equal to DATA_CHANNEL.
- **w_scale** (w_scale_constraint, single) – Scale tensor for input 'w'. It could be a scalar or a 1-D tensor, which means a per-tensor/layer or per output channel quantization. If it's a 1-D tensor, its number of elements should be equal to the number of output channels (M).
- **w_zero_point** (T2, single) – Zero point tensor for input 'w'. It could be a scalar or a 1-D tensor, which means a per-tensor/layer or per output channel quantization. If it's a 1-D tensor, its number of elements should be equal to the number of output channels (M).
- **y_scale** (y_scale_constraint, single) – Scale tensor for output 'y'. It's a scalar, which means a per-tensor/layer quantization.
- **y_zero_point** (T3, single) – Zero point tensor for output 'y'. It's a scalar, which means a per-tensor/layer quantization.
- **B** (T4, optional) – Optional 1D bias to be added to the convolution, has size of M. Bias must be quantized using scale = x_scale * w_scale and zero_point = 0

Node Outputs

- **y** (T3, single) – Output data tensor that contains the result of the convolution. The output dimensions are functions of the kernel size, stride size, and pad lengths.

Type Constraints

- **T1** – `int8`, `uint8`
- **T2** – `int8`, `uint8`
- **T3** – `int8`, `uint8`
- **T4** – `int32`
- **x_scale_constraint** – `float32`

- **w_scale_constraint** – `float32`
- **y_scale_constraint** – `float32`

Parameters

- **name** – the name of the node.
- **auto_pad** (`Optional [str]`, default='NOTSET') – auto_pad must be either NOTSET, SAME_UPPER, SAME_LOWER or VALID. Where default value is NOTSET, which means explicit padding is used. SAME_UPPER or SAME_LOWER mean pad the input so that the output spatial size match the input.In case of odd number add the extra padding at the end for SAME_UPPER and at the beginning for SAME_LOWER. VALID mean no padding.
- **dilations** (`Optional [List [int]]`, default=None) – dilation value along each spatial axis of the filter. If not present, the dilation defaults to 1 along each spatial axis.
- **group** (`Optional [int]`, default=1) – number of groups input channels and output channels are divided into. default is 1.
- **kernel_shape** (`Optional [List [int]]`, default=None) – The shape of the convolution kernel. If not present, should be inferred from input 'w'.
- **pads** (`Optional [List [int]]`, default=None) – Padding for the beginning and ending along each spatial axis, it can take any value greater than or equal to 0.The value represent the number of pixels added to the beginning and end part of the corresponding axis.`pads` format should be as follow [x1_begin, x2_begin...x1_end, x2_end,...], where xi_begin the number ofpixels added at the beginning of axis *i* and xi_end, the number of pixels added at the end of axis *i*.This attribute cannot be used simultaneously with auto_pad attribute. If not present, the padding defaultsto 0 along start and end of each spatial axis.
- **strides** (`Optional [List [int]]`, default=None) – Stride along each spatial axis. If not present, the stride defaults to 1 along each spatial axis.

auto_pad
: auto_pad must be either NOTSET, SAME_UPPER, SAME_LOWER or VALID. Where default value is NOTSET, which means explicit padding is used. SAME_UPPER or SAME_LOWER mean pad the input so that the output spatial size match the input.In case of odd number add the extra padding at the end for SAME_UPPER and at the beginning for SAME_LOWER. VALID mean no padding.

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

group
: number of groups input channels and output channels are divided into. default is 1.

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

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

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

class ONNXQLinearMatMul(*name*, *)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Matrix product that behaves like numpy.matmul: https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.matmul.html. It consumes two quantized input tensors, their scales and zero points, scale and zero point of output, and computes the quantized output. The quantization formula is y = saturate((x / y_scale) + y_zero_point). For (x / y_scale), it is rounding to nearest ties to even. Refer to https://en.wikipedia.org/wiki/Rounding for details. Scale and zero point must have same shape. They must be either scalar (per tensor) or 1-D tensor (per row for 'a' and per column for 'b'). If scale and zero point are 1-D tensor, the number of elements of scale and zero point tensor of input 'a' and output 'y' should be equal to the number of rows of input 'a', and the number of elements of scale and zero point tensor of input 'b' should be equal to the number of columns of input 'b'. Production must never overflow, and accumulation may overflow if and only if in 32 bits.

Node Inputs

- **a** (T1, single) – N-dimensional quantized matrix a
- **a_scale** (a_scale_constraint, single) – scale of quantized input a
- **a_zero_point** (T1, single) – zero point of quantized input a
- **b** (T2, single) – N-dimensional quantized matrix b
- **b_scale** (b_scale_constraint, single) – scale of quantized input b
- **b_zero_point** (T2, single) – zero point of quantized input b
- **y_scale** (y_scale_constraint, single) – scale of quantized output y
- **y_zero_point** (T3, single) – zero point of quantized output y

Node Outputs

- **y** (T3, single) – Quantized matrix multiply results from a * b

Type Constraints

- **T1** – `int8`, `uint8`
- **T2** – `int8`, `uint8`
- **T3** – `int8`, `uint8`
- **a_scale_constraint** – `float32`
- **b_scale_constraint** – `float32`
- **y_scale_constraint** – `float32`

Parameters **name** – the name of the node.

class ONNXQuantizeLinear(*name*, *)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

The linear per-tensor/layer quantization operator. It consumes a high precision tensor, a scale, a zero point to compute the low precision / quantized tensor. The quantization formula is y = saturate ((x / y_scale) + y_zero_point). For saturation, it saturates to [0, 255] if it's uint8, or [-128, 127] if it's int8. For (x / y_scale), it's rounding to nearest ties to even. Refer to https://en.wikipedia.org/wiki/Rounding for details. 'y_zero_point' and 'y' must have same type.

Node Inputs

- **x** (T1, single) – N-D full precision Input tensor to be quantized.
- **y_scale** (y_scale_constraint, single) – Scale for doing quantization to get 'y'. It's a scalar, which means a per-tensor/layer quantization.

- **y_zero_point** (T2, optional) – Zero point for doing quantization to get 'y'. It's a scalar, which means a per-tensor/layer quantization. Default value is uint8 typed 0 if it's not specified.

Node Outputs

- **y** (T2, single) – N-D quantized output tensor. It has same shape as input 'x'.

Type Constraints

- **T1** – `float32`, `int32`
- **T2** – `int8`, `uint8`
- **y_scale_constraint** – `float32`

Parameters **`name`** – the name of the node.

`class ONNXRNN`(*name*, ***, *activation_alpha=None*, *activation_beta=None*, *activations=['Tanh', 'Tanh']*, *clip=None*, *direction='forward'*, *hidden_size=None*)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Notations:

X - input tensor

i - input gate

t - time step (t-1 means previous time step)

Wi - W parameter weight matrix for input gate

Ri - R recurrence weight matrix for input gate

Wbi - W parameter bias vector for input gate

Rbi - R parameter bias vector for input gate

WBi - W parameter weight matrix for backward input gate

RBi - R recurrence weight matrix for backward input gate

WBbi - WR bias vectors for backward input gate

RBbi - RR bias vectors for backward input gate

H - Hidden state

num_directions - 2 if direction == bidirectional else 1

Activation functions:

Relu(x) - max(0, x)

Tanh(x) - (1 - e^{-2x})/(1 + e^{-2x})

Sigmoid(x) - 1/(1 + e^{-x})

(NOTE: Below are optional)

Affine(x) - alpha*x + beta

LeakyRelu(x) - x if x >= 0 else alpha * x

ThresholdedRelu(x) - x if x >= alpha else 0

ScaledTanh(x) - alpha*Tanh(beta*x)

HardSigmoid(x) - min(max(alpha*x + beta, 0), 1)

Elu(x) - x if x >= 0 else alpha*(e^x - 1)

Softsign(x) - x/(1 + |x|)

Softplus(x) - log(1 + e^x)

Equations (Default: f=Tanh):

- Ht = f(Xt*(Wi^T) + Ht-1*(Ri^T) + Wbi + Rbi)

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

Node Inputs

- **X** (T, single) – The input sequences packed (and potentially padded) into one 3-D tensor with the shape of *[seq_length, batch_size, input_size]*.
- **W** (T, single) – The weight tensor for input gate. Concatenation of *Wi* and *WBi* (if bidirectional). The tensor has shape *[num_directions, hidden_size, input_size]*.
- **R** (T, single) – The recurrence weight tensor. Concatenation of *Ri* and *RBi* (if bidirectional). The tensor has shape *[num_directions, hidden_size, hidden_size]*.
- **B** (T, optional) – The bias tensor for input gate. Concatenation of *[Wbi, Rbi]* and *[WBbi, RBbi]* (if bidirectional). The tensor has shape *[num_directions, 2*hidden_size]*. Optional: If not specified - assumed to be 0.
- **sequence_lens** (T1, optional) – Optional tensor specifying lengths of the sequences in a batch. If not specified - assumed all sequences in the batch to have length *seq_length*. It has shape *[batch_size]*.
- **initial_h** (T, optional) – Optional initial value of the hidden. If not specified - assumed to be 0. It has shape *[num_directions, batch_size, hidden_size]*.

Node Outputs

- **Y** (T, optional) – A tensor that concats all the intermediate output values of the hidden. It has shape *[seq_length, num_directions, batch_size, hidden_size]*.
- **Y_h** (T, optional) – The last output value of the hidden. It has shape *[num_directions, batch_size, hidden_size]*.

Type Constraints

- **T** – `float16`, `float32`, `float64`
- **T1** – `int32`

Parameters

- **`name`** – the name of the node.
- **`activation_alpha`** (`Optional` [`List` [`float`]], default=None) – Optional scaling values used by some activation functions. The values are consumed in the order of activation functions, for example (f, g, h) in LSTM. Default values are the same as of corresponding ONNX operators.For example with LeakyRelu, the default alpha is 0.01.
- **`activation_beta`** (`Optional` [`List` [`float`]], default=None) – Optional scaling values used by some activation functions. The values are consumed in the order of activation

functions, for example (f, g, h) in LSTM. Default values are the same as of corresponding ONNX operators.

- **activations** (Optional [List [str]], default=['Tanh', 'Tanh']) – One (or two if bidirectional) activation function for input gate. The activation function must be one of the activation functions specified above. Optional: Default *Tanh* if not specified.
- **clip** (Optional [float], default=None) – Cell clip threshold. Clipping bounds the elements of a tensor in the range of [-threshold, +threshold] and is applied to the input of activations. No clip if not specified.
- **direction** (Optional [str], default='forward') – Specify if the RNN is forward, reverse, or bidirectional. Must be one of forward (default), reverse, or bidirectional.
- **hidden_size** (Optional [int], default=None) – Number of neurons in the hidden layer

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

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

activations
One (or two if bidirectional) activation function for input gate. The activation function must be one of the activation functions specified above. Optional: Default *Tanh* if not specified.

clip
Cell clip threshold. Clipping bounds the elements of a tensor in the range of [-threshold, +threshold] and is applied to the input of activations. No clip if not specified.

direction
Specify if the RNN is forward, reverse, or bidirectional. Must be one of forward (default), reverse, or bidirectional.

hidden_size
Number of neurons in the hidden layer

class ONNXRandomNormal(*name*, *, *dtype=1*, *mean=0.0*, *scale=1.0*, *seed=None*, *shape*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

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

Node Inputs

Node Outputs

- **output** (T, single) – Output tensor of random values drawn from normal distribution

Type Constraints

- **T** – float16, float32, float64

Parameters

- **name** – the name of the node.

- **dtype** (Optional [int], default=1) – The data type for the elements of the output tensor. Default is TensorProto::FLOAT.
- **mean** (Optional [float], default=0.0) – The mean of the normal distribution.
- **scale** (Optional [float], default=1.0) – The standard deviation of the normal distribution.
- **seed** (Optional [float], default=None) – (Optional) Seed to the random generator, if not specified we will auto generate one.
- **shape** (List [int]) – The shape of the output tensor.

dtype
The data type for the elements of the output tensor. Default is TensorProto::FLOAT.

mean
The mean of the normal distribution.

scale
The standard deviation of the normal distribution.

seed
(Optional) Seed to the random generator, if not specified we will auto generate one.

shape
The shape of the output tensor.

class ONNXRandomNormalLike(*name*, *, *dtype=None*, *mean=0.0*, *scale=1.0*, *seed=None*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

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

Node Inputs

- **input** (T1, single) – Input tensor to copy shape and optionally type information from.

Node Outputs

- **output** (T2, single) – Output tensor of random values drawn from normal distribution

Type Constraints

- **T1** – uint8, uint16, uint32, uint64, int8, int16, int32, int64, float16, float32, float64, bool_, complex64, complex128
- **T2** – float16, float32, float64

Parameters

- **name** – the name of the node.
- **dtype** (Optional [int], default=None) – (Optional) The data type for the elements of the output tensor, if not specified, we will use the data type of the input tensor.
- **mean** (Optional [float], default=0.0) – The mean of the normal distribution.
- **scale** (Optional [float], default=1.0) – The standard deviation of the normal distribution.

- **seed** (Optional [float], default=None) – (Optional) Seed to the random generator, if not specified we will auto generate one.

dtype
(Optional) The data type for the elements of the output tensor, if not specified, we will use the data type of the input tensor.

mean
The mean of the normal distribution.

scale
The standard deviation of the normal distribution.

seed
(Optional) Seed to the random generator, if not specified we will auto generate one.

class ONNXRandomUniform(*name*, ***, *dtype=1*, *high=1.0*, *low=0.0*, *seed=None*, *shape*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

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

Node Inputs

Node Outputs

- **output** (T, single) – Output tensor of random values drawn from uniform distribution

Type Constraints

- **T** – float16, float32, float64

Parameters

- **name** – the name of the node.
- **dtype** (Optional [int], default=1) – The data type for the elements of the output tensor. If not specified, default is TensorProto::FLOAT.
- **high** (Optional [float], default=1.0) – Upper boundary of the output values.
- **low** (Optional [float], default=0.0) – Lower boundary of the output values.
- **seed** (Optional [float], default=None) – (Optional) Seed to the random generator, if not specified we will auto generate one.
- **shape** (List [int]) – The shape of the output tensor.

dtype
The data type for the elements of the output tensor. If not specified, default is TensorProto::FLOAT.

high
Upper boundary of the output values.

low
Lower boundary of the output values.

seed
(Optional) Seed to the random generator, if not specified we will auto generate one.

shape
The shape of the output tensor.

class ONNXRandomUniformLike(*name*, *, *dtype=None*, *high=1.0*, *low=0.0*, *seed=None*)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

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

Node Inputs

- **input** (T1, single) – Input tensor to copy shape and optionally type information from.

Node Outputs

- **output** (T2, single) – Output tensor of random values drawn from uniform distribution

Type Constraints

- **T1** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`, `bool_`, `complex64`, `complex128`
- **T2** – `float16`, `float32`, `float64`

Parameters

- **name** – the name of the node.
- **dtype** (`Optional` [`int`], default=None) – (Optional) The data type for the elements of the output tensor, if not specified, we will use the data type of the input tensor.
- **high** (`Optional` [`float`], default=1.0) – Upper boundary of the output values.
- **low** (`Optional` [`float`], default=0.0) – Lower boundary of the output values.
- **seed** (`Optional` [`float`], default=None) – (Optional) Seed to the random generator, if not specified we will auto generate one.

dtype
(Optional) The data type for the elements of the output tensor, if not specified, we will use the data type of the input tensor.

high
Upper boundary of the output values.

low
Lower boundary of the output values.

seed
(Optional) Seed to the random generator, if not specified we will auto generate one.

class ONNXRange(*name*, *)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

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

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

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

for(int i=0; i<number_of_elements; ++i)

{

` output[i] = start + (i * delta); `

}

Example 1 Inputs: start = 3, limit = 9, delta = 3 Output: [3, 6]

Example 2 Inputs: start = 10, limit = 4, delta = -2 Output: [10, 8, 6]

Node Inputs

- **start** (T, single) – Scalar. First entry for the range of output values.
- **limit** (T, single) – Scalar. Exclusive upper limit for the range of output values.
- **delta** (T, single) – Scalar. Value to step by.

Node Outputs

- **output** (T, single) – A 1-D tensor with same type as the inputs containing generated range of values.

Type Constraints

- **T** – `float32`, `float64`, `int16`, `int32`, `int64`

Parameters **`name`** – the name of the node.

`class ONNXReciprocal`(*name*, *)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **X** (T, single) – Input tensor

Node Outputs

- **Y** (T, single) – Output tensor

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters **`name`** – the name of the node.

`class ONNXReduceL1`(*name*, *, *axes=None*, *keepdims=1*)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Computes the L1 norm of the input tensor's element along the provided axes. The resulted tensor has the same rank as the input if keepdims equal 1. If keepdims equal 0, then the resulted tensor have the reduced dimension pruned.

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

Node Inputs

- **data** (T, single) – An input tensor.

Node Outputs

- **reduced** (T, single) – Reduced output tensor.

Type Constraints

- **T** – `uint32`, `uint64`, `int32`, `int64`, `float16`, `float32`, `float64`

Parameters

- **name** – the name of the node.
- **axes** (`Optional [List [int]]`, default=None) – A list of integers, along which to reduce. The default is to reduce over all the dimensions of the input tensor. Accepted range is [-r, r-1] where r = rank(data).
- **keepdims** (`Optional [int]`, default=1) – Keep the reduced dimension or not, default 1 mean keep reduced dimension.

axes
A list of integers, along which to reduce. The default is to reduce over all the dimensions of the input tensor. Accepted range is [-r, r-1] where r = rank(data).

keepdims
Keep the reduced dimension or not, default 1 mean keep reduced dimension.

class ONNXReduceL2(*name*, *, *axes=None*, *keepdims=1*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Computes the L2 norm of the input tensor's element along the provided axes. The resulted tensor has the same rank as the input if keepdims equal 1. If keepdims equal 0, then the resulted tensor have the reduced dimension pruned.

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

Node Inputs

- **data** (T, single) – An input tensor.

Node Outputs

- **reduced** (T, single) – Reduced output tensor.

Type Constraints

- **T** – `uint32`, `uint64`, `int32`, `int64`, `float16`, `float32`, `float64`

Parameters

- **name** – the name of the node.
- **axes** (`Optional [List [int]]`, default=None) – A list of integers, along which to reduce. The default is to reduce over all the dimensions of the input tensor. Accepted range is [-r, r-1] where r = rank(data).
- **keepdims** (`Optional [int]`, default=1) – Keep the reduced dimension or not, default 1 mean keep reduced dimension.

axes
A list of integers, along which to reduce. The default is to reduce over all the dimensions of the input tensor. Accepted range is [-r, r-1] where r = rank(data).

keepdims
Keep the reduced dimension or not, default 1 mean keep reduced dimension.

class ONNXReduceLogSum(*name*, *, *axes=None*, *keepdims=1*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Computes the log sum of the input tensor's element along the provided axes. The resulted tensor has the same rank as the input if keepdims equal 1. If keepdims equal 0, then the resulted tensor have the reduced dimension pruned.

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

Node Inputs

- **data** (T, single) – An input tensor.

Node Outputs

- **reduced** (T, single) – Reduced output tensor.

Type Constraints

- **T** – `uint32`, `uint64`, `int32`, `int64`, `float16`, `float32`, `float64`

Parameters

- **name** – the name of the node.
- **axes** (`Optional [List [int]]`, default=None) – A list of integers, along which to reduce. The default is to reduce over all the dimensions of the input tensor. Accepted range is [-r, r-1] where r = rank(data).
- **keepdims** (`Optional [int]`, default=1) – Keep the reduced dimension or not, default 1 mean keep reduced dimension.

axes
A list of integers, along which to reduce. The default is to reduce over all the dimensions of the input tensor. Accepted range is [-r, r-1] where r = rank(data).

keepdims
Keep the reduced dimension or not, default 1 mean keep reduced dimension.

class ONNXReduceLogSumExp(*name*, *, *axes=None*, *keepdims=1*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Computes the log sum exponent of the input tensor's element along the provided axes. The resulted tensor has the same rank as the input if keepdims equal 1. If keepdims equal 0, then the resulted tensor have the reduced dimension pruned.

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

Node Inputs

- **data** (T, single) – An input tensor.

Node Outputs

- **reduced** (T, single) – Reduced output tensor.

Type Constraints

- **T** – `uint32`, `uint64`, `int32`, `int64`, `float16`, `float32`, `float64`

Parameters

- **name** – the name of the node.
- **axes** (`Optional [List [int]]`, default=None) – A list of integers, along which to reduce. The default is to reduce over all the dimensions of the input tensor. Accepted range is [-r, r-1] where r = rank(data).

- **keepdims** (Optional [int], default=1) – Keep the reduced dimension or not, default 1 mean keep reduced dimension.

axes
A list of integers, along which to reduce. The default is to reduce over all the dimensions of the input tensor. Accepted range is [-r, r-1] where r = rank(data).

keepdims
Keep the reduced dimension or not, default 1 mean keep reduced dimension.

class ONNXReduceMax(*name*, ***, *axes=None*, *keepdims=1*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Computes the max of the input tensor's element along the provided axes. The resulted tensor has the same rank as the input if keepdims equal 1. If keepdims equal 0, then the resulted tensor have the reduced dimension pruned.

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

Node Inputs

- **data** (T, single) – An input tensor.

Node Outputs

- **reduced** (T, single) – Reduced output tensor.

Type Constraints

- **T** – uint32, uint64, int32, int64, float16, float32, float64, uint8, int8

Parameters

- **name** – the name of the node.
- **axes** (Optional [List [int]], default=None) – A list of integers, along which to reduce. The default is to reduce over all the dimensions of the input tensor. Accepted range is [-r, r-1] where r = rank(data).
- **keepdims** (Optional [int], default=1) – Keep the reduced dimension or not, default 1 mean keep reduced dimension.

axes
A list of integers, along which to reduce. The default is to reduce over all the dimensions of the input tensor. Accepted range is [-r, r-1] where r = rank(data).

keepdims
Keep the reduced dimension or not, default 1 mean keep reduced dimension.

class ONNXReduceMean(*name*, ***, *axes=None*, *keepdims=1*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Computes the mean of the input tensor's element along the provided axes. The resulted tensor has the same rank as the input if keepdims equal 1. If keepdims equal 0, then the resulted tensor have the reduced dimension pruned.

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

Node Inputs

- **data** (T, single) – An input tensor.

Node Outputs

- **reduced** (T, single) – Reduced output tensor.

Type Constraints

- **T** – `uint32`, `uint64`, `int32`, `int64`, `float16`, `float32`, `float64`

Parameters

- **name** – the name of the node.
- **axes** (`Optional` [`List` [`int`]], default=None) – A list of integers, along which to reduce. The default is to reduce over all the dimensions of the input tensor. Accepted range is [-r, r-1] where r = rank(data).
- **keepdims** (`Optional` [`int`], default=1) – Keep the reduced dimension or not, default 1 mean keep reduced dimension.

axes
A list of integers, along which to reduce. The default is to reduce over all the dimensions of the input tensor. Accepted range is [-r, r-1] where r = rank(data).

keepdims
Keep the reduced dimension or not, default 1 mean keep reduced dimension.

class ONNXReduceMin(*name*, ***, *axes=None*, *keepdims=1*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Computes the min of the input tensor's element along the provided axes. The resulted tensor has the same rank as the input if keepdims equal 1. If keepdims equal 0, then the resulted tensor have the reduced dimension pruned.

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

Node Inputs

- **data** (T, single) – An input tensor.

Node Outputs

- **reduced** (T, single) – Reduced output tensor.

Type Constraints

- **T** – `uint32`, `uint64`, `int32`, `int64`, `float16`, `float32`, `float64`, `uint8`, `int8`

Parameters

- **name** – the name of the node.
- **axes** (`Optional` [`List` [`int`]], default=None) – A list of integers, along which to reduce. The default is to reduce over all the dimensions of the input tensor. Accepted range is [-r, r-1] where r = rank(data).
- **keepdims** (`Optional` [`int`], default=1) – Keep the reduced dimension or not, default 1 mean keep reduced dimension.

axes
A list of integers, along which to reduce. The default is to reduce over all the dimensions of the input tensor. Accepted range is [-r, r-1] where r = rank(data).

keepdims
Keep the reduced dimension or not, default 1 mean keep reduced dimension.

class ONNXReduceProd(*name*, *, *axes=None*, *keepdims=1*)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Computes the product of the input tensor's element along the provided axes. The resulted tensor has the same rank as the input if keepdims equal 1. If keepdims equal 0, then the resulted tensor have the reduced dimension pruned.

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

Node Inputs

- **data** (T, single) – An input tensor.

Node Outputs

- **reduced** (T, single) – Reduced output tensor.

Type Constraints

- **T** – `uint32`, `uint64`, `int32`, `int64`, `float16`, `float32`, `float64`

Parameters

- **name** – the name of the node.
- **axes** (`Optional [List [int]]`, default=None) – A list of integers, along which to reduce. The default is to reduce over all the dimensions of the input tensor. Accepted range is [-r, r-1] where r = rank(data).
- **keepdims** (`Optional [int]`, default=1) – Keep the reduced dimension or not, default 1 mean keep reduced dimension.

axes
A list of integers, along which to reduce. The default is to reduce over all the dimensions of the input tensor. Accepted range is [-r, r-1] where r = rank(data).

keepdims
Keep the reduced dimension or not, default 1 mean keep reduced dimension.

class ONNXReduceSum(*name*, *, *axes=None*, *keepdims=1*)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Computes the sum of the input tensor's element along the provided axes. The resulted tensor has the same rank as the input if keepdims equal 1. If keepdims equal 0, then the resulted tensor have the reduced dimension pruned.

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

Node Inputs

- **data** (T, single) – An input tensor.

Node Outputs

- **reduced** (T, single) – Reduced output tensor.

Type Constraints

- **T** – `uint32`, `uint64`, `int32`, `int64`, `float16`, `float32`, `float64`

Parameters

- **name** – the name of the node.

- **axes** (Optional [List [int]], default=None) – A list of integers, along which to reduce. The default is to reduce over all the dimensions of the input tensor. Accepted range is [-r, r-1] where r = rank(data).
- **keepdims** (Optional [int], default=1) – Keep the reduced dimension or not, default 1 mean keep reduced dimension.

axes
A list of integers, along which to reduce. The default is to reduce over all the dimensions of the input tensor. Accepted range is [-r, r-1] where r = rank(data).

keepdims
Keep the reduced dimension or not, default 1 mean keep reduced dimension.

class ONNXReduceSumSquare(*name*, *, *axes=None*, *keepdims=1*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Computes the sum square of the input tensor's element along the provided axes. The resulted tensor has the same rank as the input if keepdims equal 1. If keepdims equal 0, then the resulted tensor have the reduced dimension pruned.

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

Node Inputs

- **data** (T, single) – An input tensor.

Node Outputs

- **reduced** (T, single) – Reduced output tensor.

Type Constraints

- **T** – uint32, uint64, int32, int64, float16, float32, float64

Parameters

- **name** – the name of the node.
- **axes** (Optional [List [int]], default=None) – A list of integers, along which to reduce. The default is to reduce over all the dimensions of the input tensor. Accepted range is [-r, r-1] where r = rank(data).
- **keepdims** (Optional [int], default=1) – Keep the reduced dimension or not, default 1 mean keep reduced dimension.

axes
A list of integers, along which to reduce. The default is to reduce over all the dimensions of the input tensor. Accepted range is [-r, r-1] where r = rank(data).

keepdims
Keep the reduced dimension or not, default 1 mean keep reduced dimension.

class ONNXRelu(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **X** (T, single) – Input tensor

Node Outputs

- **Y** (T, single) – Output tensor

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters **name** – the name of the node.

class ONNXReshape(*name*, *)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Reshape the input tensor similar to numpy.reshape. First input is the data tensor, second input is a shape tensor which specifies the output shape. It outputs the reshaped tensor. At most one dimension of the new shape can be -1. In this case, the value is inferred from the size of the tensor and the remaining dimensions. A dimension could also be 0, in which case the actual dimension value is unchanged (i.e. taken from the input tensor).

Node Inputs

- **data** (T, single) – An input tensor.
- **shape** (shape_constraint, single) – Specified shape for output.

Node Outputs

- **reshaped** (T, single) – Reshaped data.

Type Constraints

- **T** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`, `bool_`, `complex64`, `complex128`
- **shape_constraint** – `int64`

Parameters **name** – the name of the node.

class ONNXResize(*name*, *, *coordinate_transformation_mode='half_pixel'*, *cubic_coeff_a=-0.75*, *exclude_outside=0*, *extrapolation_value=0.0*, *mode='nearest'*, *nearest_mode='round_prefer_floor'*)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Resize the input tensor. In general, it calculates every value in the output tensor as a weighted average of neighborhood (a.k.a. sampling locations) in the input tensor. Each dimension value of the output tensor is:

output_dimension = floor(input_dimension * (roi_end - roi_start) * scale) if input "sizes" is not specified.

Node Inputs

- **X** (T1, single) – N-D tensor
- **roi** (T2, single) – 1-D tensor given as [start1, . . . , startN, end1, . . . , endN], where N is the rank of X. The RoIs' coordinates are normalized in the coordinate system of the input image. It only takes effect when coordinate_transformation_mode is "tf_crop_and_resize"
- **scales** (scales_constraint, single) – The scale array along each dimension. It takes value greater than 0. If it's less than 1, it's sampling down, otherwise, it's upsampling. The number of elements of 'scales' should be the same as the rank of input 'X'. Only one of 'scales' and 'sizes' can be specified. If 'size' is needed, the user can use an empty string as the name of 'scales' in this operator's input list.
- **sizes** (sizes_constraint, optional) – The size of the output tensor. The number of elements of 'sizes' should be the same as the rank of input 'X'. Only one of 'scales' and 'sizes' can be specified.

Node Outputs

- **Y** (T1, single) – N-D tensor after resizing

Type Constraints

- **T1** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`, `bool_`, `complex64`, `complex128`
- **T2** – `float16`, `float32`, `float64`
- **scales_constraint** – `float32`
- **sizes_constraint** – `int64`

Parameters

- **`name`** – the name of the node.
- **`coordinate_transformation_mode`** –

This attribute describes how to transform the coordinate in the resized tensor to the coordinate in the original tensor.

The coordinate of each dimension is transformed individually. Let's describe a case using axis x as an example. Denote x_resized as the coordinate of axis x in the resized tensor, x_original as the coordinate of axis x in the original tensor, length_original as the length of the original tensor in axis x, length_resized as the length of the resized tensor in axis x, roi_x = (start_x, end_x) of the axis x in input "roi", scale = length_resized / length_original,

if coordinate_transformation_mode is "half_pixel",
 x_original = (x_resized + 0.5) / scale - 0.5,

if coordinate_transformation_mode is "pytorch_half_pixel",
 x_original = length_resized > 1 ? (x_resized + 0.5) / scale - 0.5 : 0,

if coordinate_transformation_mode is "align_corners",
 x_original = x_resized * (length_original - 1) / (length_resized - 1),

if coordinate_transformation_mode is "asymmetric",
 x_original = x_resized / scale,

if coordinate_transformation_mode is "tf_half_pixel_for_nn",
 x_original = (x_resized + 0.5) / scale,

if coordinate_transformation_mode is "tf_crop_and_resize",
 x_original = length_resized > 1 ? start_x * (length_original - 1) + x_resized * (end_x - start_x) * (length_original - 1) / (length_resized - 1) : 0.5 * (start_x + end_x) * (length_original - 1). :type coordinate_transformation_mode: `Optional` [`str`], default='half_pixel' :param cubic_coeff_a: The coefficient 'a' used in cubic interpolation. Two common choice are -0.5 (in some cases of TensorFlow) and -0.75 (in PyTorch). Check out Equation (4) in https://ieeexplore.ieee.org/document/1163711 for the details. This attribute is valid only if "mode" is "cubic". :type cubic_coeff_a: `Optional` [`float`], default=-0.75 :param exclude_outside: If set to 1, the weight of sampling locations outside the tensor will be set to 0 and the weight will be renormalized so that their sum is 1.0. The default value is 0. :type exclude_outside: `Optional` [`int`], default=0 :param extrapolation_value: When coordinate_transformation_mode is "tf_crop_and_resize" and x_original is outside the range [0, length_original - 1], this value is used as the corresponding output value. Default is 0.0f. :type extrapolation_value: `Optional` [`float`], default=0.0 :param mode: Three interpolation modes: nearest (default), linear and cubic. The "linear" mode includes linear interpolation for 1D tensor and N-linear interpolation for N-D tensor (for example, bilinear interpolation for 2D tensor). The "cubic" mode includes cubic interpolation for 1D tensor and N-cubic interpolation for N-D tensor (for example, bicubic interpolation for 2D tensor). :type mode: `Optional` [`str`], default='nearest' :param nearest_mode: Four modes: round_prefer_floor (default, as known as round half down), round_prefer_ceil (as known as round half up), floor, ceil. Only used by nearest interpolation. It indicates how to get "nearest" pixel in input tensor from x_original, so this attribute is valid only if "mode" is "nearest". :type nearest_mode: `Optional` [`str`], default='round_prefer_floor'

coordinate_transformation_mode
This attribute describes how to transform the coordinate in the resized tensor to the coordinate in the original tensor.

The coordinate of each dimension is transformed individually. Let's describe a case using axis x as an example. Denote x_resized as the coordinate of axis x in the resized tensor, x_original as the coordinate of axis x in the original tensor, length_original as the length of the original tensor in axis x, length_resized as the length of the resized tensor in axis x, roi_x = (start_x, end_x) of the axis x in input "roi", scale = length_resized / length_original,

if coordinate_transformation_mode is "half_pixel",
 x_original = (x_resized + 0.5) / scale - 0.5,

if coordinate_transformation_mode is "pytorch_half_pixel",
 x_original = length_resized > 1 ? (x_resized + 0.5) / scale - 0.5 : 0,

if coordinate_transformation_mode is "align_corners",
 x_original = x_resized * (length_original - 1) / (length_resized - 1),

if coordinate_transformation_mode is "asymmetric",
 x_original = x_resized / scale,

if coordinate_transformation_mode is "tf_half_pixel_for_nn",
 x_original = (x_resized + 0.5) / scale,

if coordinate_transformation_mode is "tf_crop_and_resize",
 x_original = length_resized > 1 ? start_x * (length_original - 1) + x_resized * (end_x - start_x) * (length_original - 1) / (length_resized - 1) : 0.5 * (start_x + end_x) * (length_original - 1).

cubic_coeff_a
The coefficient 'a' used in cubic interpolation. Two common choice are -0.5 (in some cases of TensorFlow) and -0.75 (in PyTorch). Check out Equation (4) in https://ieeexplore.ieee.org/document/1163711 for the details. This attribute is valid only if "mode" is "cubic".

exclude_outside
If set to 1, the weight of sampling locations outside the tensor will be set to 0 and the weight will be renormalized so that their sum is 1.0. The default value is 0.

extrapolation_value
When coordinate_transformation_mode is "tf_crop_and_resize" and x_original is outside the range [0, length_original - 1], this value is used as the corresponding output value. Default is 0.0f.

mode
Three interpolation modes: nearest (default), linear and cubic. The "linear" mode includes linear interpolation for 1D tensor and N-linear interpolation for N-D tensor (for example, bilinear interpolation for 2D tensor). The "cubic" mode includes cubic interpolation for 1D tensor and N-cubic interpolation for N-D tensor (for example, bicubic interpolation for 2D tensor).

nearest_mode
Four modes: round_prefer_floor (default, as known as round half down), round_prefer_ceil (as known as round half up), floor, ceil. Only used by nearest interpolation. It indicates how to get "nearest" pixel in input tensor from x_original, so this attribute is valid only if "mode" is "nearest".

class ONNXReverseSequence(*name*, *, *batch_axis=1*, *time_axis=0*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Reverse batch of sequences having different lengths specified by *sequence_lens*.

For each slice i iterating on batch axis, the operator reverses the first sequence_lens[i] elements on time axis, and copies elements whose index's beyond sequence_lens[i] to the output. So the output slice i contains reversed sequences on the first sequence_lens[i] elements, then have original values copied for the other elements.

Example 1:

input = [[0.0, 4.0, 8.0, 12.0], [1.0, 5.0, 9.0, 13.0], [2.0, 6.0, 10.0, 14.0], [3.0, 7.0, 11.0, 15.0]]

sequence_lens = [4, 3, 2, 1] time_axis = 0 batch_axis = 1

output = [[3.0, 6.0, 9.0, 12.0], [2.0, 5.0, 8.0, 13.0], [1.0, 4.0, 10.0, 14.0], [0.0, 7.0, 11.0, 15.0]]

Example 2:

input = [[0.0, 1.0, 2.0, 3.0], [4.0, 5.0, 6.0, 7.0], [8.0, 9.0, 10.0, 11.0], [12.0, 13.0, 14.0, 15.0]]

sequence_lens = [1, 2, 3, 4] time_axis = 1 batch_axis = 0

output = [[0.0, 1.0, 2.0, 3.0], [5.0, 4.0, 6.0, 7.0], [10.0, 9.0, 8.0, 11.0], [15.0, 14.0, 13.0, 12.0]]

Node Inputs

- **input** (T, single) – Tensor of rank r >= 2.
- **sequence_lens** (sequence_lens_constraint, single) – Tensor specifying lengths of the sequences in a batch. It has shape *[batch_size]*.

Node Outputs

- **Y** (T, single) – Tensor with same shape of input.

Type Constraints

- **T** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`, `bool_`, `complex64`, `complex128`
- **sequence_lens_constraint** – `int64`

Parameters

- **name** – the name of the node.
- **batch_axis** (`Optional [int]`, default=1) – (Optional) Specify which axis is batch axis. Must be one of 1 (default), or 0.
- **time_axis** (`Optional [int]`, default=0) – (Optional) Specify which axis is time axis. Must be one of 0 (default), or 1.

batch_axis
(Optional) Specify which axis is batch axis. Must be one of 1 (default), or 0.

time_axis
(Optional) Specify which axis is time axis. Must be one of 0 (default), or 1.

class ONNXRoiAlign(*name*, ***, *mode='avg'*, *output_height=1*, *output_width=1*, *sampling_ratio=0*, *spatial_scale=1.0*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Region of Interest (RoI) align operation described in the [Mask R-CNN paper](https://arxiv.org/abs/1703.06870). RoiAlign consumes an input tensor X and region of interests (rois) to apply pooling across each RoI; it produces a 4-D tensor of shape (num_rois, C, output_height, output_width).

RoiAlign is proposed to avoid the misalignment by removing quantizations while converting from original image into feature map and from feature map into RoI feature; in each ROI bin, the value of the sampled locations are computed directly through bilinear interpolation.

Node Inputs

- **X** (T1, single) – Input data tensor from the previous operator; 4-D feature map of shape (N, C, H, W), where N is the batch size, C is the number of channels, and H and W are the height and the width of the data.

- **rois** (T1, single) – RoIs (Regions of Interest) to pool over; rois is 2-D input of shape (num_rois, 4) given as [[x1, y1, x2, y2], ...]. The RoIs' coordinates are in the coordinate system of the input image. Each coordinate set has a 1:1 correspondence with the 'batch_indices' input.
- **batch_indices** (T2, single) – 1-D tensor of shape (num_rois,) with each element denoting the index of the corresponding image in the batch.

Node Outputs

- **Y** (T1, single) – RoI pooled output, 4-D tensor of shape (num_rois, C, output_height, output_width). The r-th batch element Y[r-1] is a pooled feature map corresponding to the r-th RoI X[r-1].

Type Constraints

- **T1** – `float16`, `float32`, `float64`
- **T2** – `int64`

Parameters

- **`name`** – the name of the node.
- **`mode`** (`Optional` [`str`], default='avg') – The pooling method. Two modes are supported: 'avg' and 'max'. Default is 'avg'.
- **`output_height`** (`Optional` [`int`], default=1) – default 1; Pooled output Y's height.
- **`output_width`** (`Optional` [`int`], default=1) – default 1; Pooled output Y's width.
- **`sampling_ratio`** (`Optional` [`int`], default=0) – Number of sampling points in the interpolation grid used to compute the output value of each pooled output bin. If > 0, then exactly sampling_ratio x sampling_ratio grid points are used. If == 0, then an adaptive number of grid points are used (computed as ceil(roi_width / output_width), and likewise for height). Default is 0.
- **`spatial_scale`** (`Optional` [`float`], default=1.0) – Multiplicative spatial scale factor to translate ROI coordinates from their input spatial scale to the scale used when pooling, i.e., spatial scale of the input feature map X relative to the input image. E.g.; default is 1.0f.

`mode`
The pooling method. Two modes are supported: 'avg' and 'max'. Default is 'avg'.

`output_height`
default 1; Pooled output Y's height.

`output_width`
default 1; Pooled output Y's width.

`sampling_ratio`
Number of sampling points in the interpolation grid used to compute the output value of each pooled output bin. If > 0, then exactly sampling_ratio x sampling_ratio grid points are used. If == 0, then an adaptive number of grid points are used (computed as ceil(roi_width / output_width), and likewise for height). Default is 0.

`spatial_scale`
Multiplicative spatial scale factor to translate ROI coordinates from their input spatial scale to the scale used when pooling, i.e., spatial scale of the input feature map X relative to the input image. E.g.; default is 1.0f.

class `ONNXRound`(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Round takes one input Tensor and rounds the values, element-wise, meaning it finds the nearest integer for each value. In case of halfs, the rule is to round them to the nearest even integer. The output tensor has the same shape and type as the input.

Examples: ` round([0.9]) = [1.0] round([2.5]) = [2.0] round([2.3]) = [2.0] round([1.5]) = [2.0] round([-4.5]) = [-4.0] `

Node Inputs

- **X** (T, single) – Input tensor

Node Outputs

- **Y** (T, single) – Output tensor

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters **`name`** – the name of the node.

`class ONNXSVMClassifier`(*name*, ***, *classlabels_ints=None*, *classlabels_strings=None*, *coefficients=None*, *kernel_params=None*, *kernel_type='LINEAR'*, *post_transform='NONE'*, *prob_a=None*, *prob_b=None*, *rho=None*, *support_vectors=None*, *vectors_per_class=None*)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Support Vector Machine classifier

Node Inputs

- **X** (T1, single) – Data to be classified.

Node Outputs

- **Y** (T2, single) – Classification outputs (one class per example).
- **Z** (Z_constraint, single) – Class scores (one per class per example), if prob_a and prob_b are provided they are probabilities for each class, otherwise they are raw scores.

Type Constraints

- **T1** – `float32`, `float64`, `int64`, `int32`
- **T2** – `int64`
- **Z_constraint** – `float32`

Parameters

- **`name`** – the name of the node.
- **`classlabels_ints`** (`Optional` [`List` [`int`]], default=None) – Class labels if using integer labels.
One and only one of the 'classlabels_*' attributes must be defined.
- **`classlabels_strings`** (`Optional` [`List` [`str`]], default=None) – Class labels if using string labels.
One and only one of the 'classlabels_*' attributes must be defined.
- **`coefficients`** (`Optional` [`List` [`float`]], default=None) –
- **`kernel_params`** (`Optional` [`List` [`float`]], default=None) – List of 3 elements containing gamma, coef0, and degree, in that order. Zero if unused for the kernel.
- **`kernel_type`** (`Optional` [`str`], default='LINEAR') – The kernel type, one of 'LINEAR,' 'POLY,' 'RBF,' 'SIGMOID'.

- **post_transform** (Optional [str], default='NONE') – Indicates the transform to apply to the score.
One of 'NONE,' 'SOFTMAX,' 'LOGISTIC,' 'SOFTMAX_ZERO,' or 'PROBIT'
- **prob_a** (Optional [List [float]], default=None) – First set of probability coefficients.
- **prob_b** (Optional [List [float]], default=None) – Second set of probability coefficients. This array must be same size as prob_a.
If these are provided then output Z are probability estimates, otherwise they are raw scores.
- **rho** (Optional [List [float]], default=None) –
- **support_vectors** (Optional [List [float]], default=None) –
- **vectors_per_class** (Optional [List [int]], default=None) –

classlabels_ints
Class labels if using integer labels.
One and only one of the 'classlabels_*' attributes must be defined.

classlabels_strings
Class labels if using string labels.
One and only one of the 'classlabels_*' attributes must be defined.

coefficients
Object property of type list

kernel_params
List of 3 elements containing gamma, coef0, and degree, in that order. Zero if unused for the kernel.

kernel_type
The kernel type, one of 'LINEAR,' 'POLY,' 'RBF,' 'SIGMOID'.

post_transform
Indicates the transform to apply to the score.
One of 'NONE,' 'SOFTMAX,' 'LOGISTIC,' 'SOFTMAX_ZERO,' or 'PROBIT'

prob_a
First set of probability coefficients.

prob_b
Second set of probability coefficients. This array must be same size as prob_a.
If these are provided then output Z are probability estimates, otherwise they are raw scores.

rho
Object property of type list

support_vectors
Object property of type list

vectors_per_class
Object property of type list

class ONNXSVMRegressor(*name*, ***, *coefficients=None*, *kernel_params=None*, *kernel_type='LINEAR'*, *n_supports=0*, *one_class=0*, *post_transform='NONE'*, *rho=None*, *support_vectors=None*)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Support Vector Machine regression prediction and one-class SVM anomaly detection.

Node Inputs

- **X** (T, single) – Data to be regressed.

Node Outputs

- **Y** (Y_constraint, single) – Regression outputs (one score per target per example).

Type Constraints

- **T** – `float32`, `float64`, `int64`, `int32`
- **Y_constraint** – `float32`

Parameters

- **name** – the name of the node.
- **coefficients** (`Optional` [`List` [`float`]], default=None) – Support vector coefficients.
- **kernel_params** (`Optional` [`List` [`float`]], default=None) – List of 3 elements containing gamma, coef0, and degree, in that order. Zero if unused for the kernel.
- **kernel_type** (`Optional` [`str`], default='LINEAR') – The kernel type, one of 'LINEAR,' 'POLY,' 'RBF,' 'SIGMOID'.
- **n_supports** (`Optional` [`int`], default=0) – The number of support vectors.
- **one_class** (`Optional` [`int`], default=0) – Flag indicating whether the regression is a one-class SVM or not.
- **post_transform** (`Optional` [`str`], default='NONE') – Indicates the transform to apply to the score.
One of 'NONE,' 'SOFTMAX,' 'LOGISTIC,' 'SOFTMAX_ZERO,' or 'PROBIT.'
- **rho** (`Optional` [`List` [`float`]], default=None) –
- **support_vectors** (`Optional` [`List` [`float`]], default=None) – Chosen support vectors

coefficients
Support vector coefficients.

kernel_params
List of 3 elements containing gamma, coef0, and degree, in that order. Zero if unused for the kernel.

kernel_type
The kernel type, one of 'LINEAR,' 'POLY,' 'RBF,' 'SIGMOID'.

n_supports
The number of support vectors.

one_class
Flag indicating whether the regression is a one-class SVM or not.

post_transform
Indicates the transform to apply to the score.
One of 'NONE,' 'SOFTMAX,' 'LOGISTIC,' 'SOFTMAX_ZERO,' or 'PROBIT.'

rho
Object property of type list

support_vectors
Chosen support vectors

class ONNXScaler(*name*, *, *offset=None*, *scale=None*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Rescale input data, for example to standardize features by removing the mean and scaling to unit variance.

Node Inputs

- **X** (T, single) – Data to be scaled.

Node Outputs

- **Y** (Y_constraint, single) – Scaled output data.

Type Constraints

- **T** – `float32`, `float64`, `int64`, `int32`
- **Y_constraint** – `float32`

Parameters

- **`name`** – the name of the node.
- **`offset`** (`Optional` [`List` [`float`]], default=None) – First, offset by this.
Can be length of features in an [N,F] tensor or length 1, in which case it applies to all features, regardless of dimension count.
- **`scale`** (`Optional` [`List` [`float`]], default=None) – Second, multiply by this.
Can be length of features in an [N,F] tensor or length 1, in which case it applies to all features, regardless of dimension count.
Must be same length as 'offset'

`offset`
First, offset by this.
Can be length of features in an [N,F] tensor or length 1, in which case it applies to all features, regardless of dimension count.

`scale`
Second, multiply by this.
Can be length of features in an [N,F] tensor or length 1, in which case it applies to all features, regardless of dimension count.
Must be same length as 'offset'

class `ONNXScatter`(*name*, *, *axis=0*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

This operator is deprecated. Please use ScatterElements, which provides the same functionality.

Scatter takes three inputs *data*, *updates*, and *indices* of the same rank r >= 1 and an optional attribute axis that identifies an axis of *data* (by default, the outer-most axis, that is axis 0). The output of the operation is produced by creating a copy of the input *data*, and then updating its value to values specified by *updates* at specific index positions specified by *indices*. Its output shape is the same as the shape of *data*.

For each entry in *updates*, the target index in *data* is obtained by combining the corresponding entry in *indices* with the index of the entry itself: the index-value for dimension = axis is obtained from the value of the corresponding entry in *indices* and the index-value for dimension != axis is obtained from the index of the entry itself.

For instance, in a 2-D tensor case, the update corresponding to the [i][j] entry is performed as below: ```

output[indices[i][j]][j] = updates[i][j] if axis = 0, output[i][indices[i][j]] = updates[i][j] if axis = 1,

```

This operator is the inverse of GatherElements. It is similar to Torch's Scatter operation.

Example 1: ```

**data = [** [0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0],

] indices = [

[1, 0, 2], [0, 2, 1],

] updates = [
```

[1.0, 1.1, 1.2], [2.0, 2.1, 2.2],

] output = [

[2.0, 1.1, 0.0] [1.0, 0.0, 2.2] [0.0, 2.1, 1.2]

]

` Example 2: `

data = [[1.0, 2.0, 3.0, 4.0, 5.0]] indices = [[1, 3]] updates = [[1.1, 2.1]] axis = 1 output = [[1.0, 1.1, 3.0, 2.1, 5.0]]

```

**Node Inputs**

- **data** (T, single) – Tensor of rank r >= 1.
- **indices** (Tind, single) – Tensor of int32/int64 indices, of r >= 1 (same rank as input). All index values are expected to be within bounds [-s, s-1] along axis of size s. It is an error if any of the index values are out of bounds.
- **updates** (T, single) – Tensor of rank r >=1 (same rank and shape as indices)

**Node Outputs**

- **output** (T, single) – Tensor of rank r >= 1 (same rank as input).

**Type Constraints**

- **T** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`, `bool_`, `complex64`, `complex128`
- **Tind** – `int32`, `int64`

**Parameters**

- **`name`** – the name of the node.
- **`axis`** (`Optional` [`int`], default=0) – Which axis to scatter on. Negative value means counting dimensions from the back. Accepted range is [-r, r-1] where r = rank(data).

**`axis`**
Which axis to scatter on. Negative value means counting dimensions from the back. Accepted range is [-r, r-1] where r = rank(data).

**`class ONNXScatterElements`**(*name*, *, *axis=0*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

ScatterElements takes three inputs *data*, *updates*, and *indices* of the same rank r >= 1 and an optional attribute axis that identifies an axis of *data* (by default, the outer-most axis, that is axis 0). The output of the operation is produced by creating a copy of the input *data*, and then updating its value to values specified by *updates* at specific index positions specified by *indices*. Its output shape is the same as the shape of *data*.

For each entry in *updates*, the target index in *data* is obtained by combining the corresponding entry in *indices* with the index of the entry itself: the index-value for dimension = axis is obtained from the value of the corresponding entry in *indices* and the index-value for dimension != axis is obtained from the index of the entry itself.

For instance, in a 2-D tensor case, the update corresponding to the [i][j] entry is performed as below: ```

output[indices[i][j]][j] = updates[i][j] if axis = 0, output[i][indices[i][j]] = updates[i][j] if axis = 1,

```

This operator is the inverse of GatherElements. It is similar to Torch's Scatter operation.

Example 1: ```

data = [[0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0],

] indices = [

[1, 0, 2], [0, 2, 1],

] updates = [

[1.0, 1.1, 1.2], [2.0, 2.1, 2.2],

] output = [

[2.0, 1.1, 0.0] [1.0, 0.0, 2.2] [0.0, 2.1, 1.2]

]

` Example 2: `

data = [[1.0, 2.0, 3.0, 4.0, 5.0]] indices = [[1, 3]] updates = [[1.1, 2.1]] axis = 1 output = [[1.0, 1.1, 3.0, 2.1, 5.0]]

```

**Node Inputs**

- **data** (T, single) – Tensor of rank r >= 1.
- **indices** (Tind, single) – Tensor of int32/int64 indices, of r >= 1 (same rank as input). All index values are expected to be within bounds [-s, s-1] along axis of size s. It is an error if any of the index values are out of bounds.
- **updates** (T, single) – Tensor of rank r >=1 (same rank and shape as indices)

**Node Outputs**

- **output** (T, single) – Tensor of rank r >= 1 (same rank as input).

**Type Constraints**

- **T** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`, `bool_`, `complex64`, `complex128`
- **Tind** – `int32`, `int64`

**Parameters**

- **`name`** – the name of the node.
- **`axis`** (`Optional` [`int`], default=0) – Which axis to scatter on. Negative value means counting dimensions from the back. Accepted range is [-r, r-1] where r = rank(data).

**`axis`**
Which axis to scatter on. Negative value means counting dimensions from the back. Accepted range is [-r, r-1] where r = rank(data).

**`class ONNXScatterND`**(*name*, *)
Bases: *`daceml.onnx.nodes.onnx_op.ONNXOp`*

ScatterND takes three inputs *data* tensor of rank r >= 1, *indices* tensor of rank q >= 1, and *updates* tensor of rank q + r - indices.shape[-1] - 1. The output of the operation is produced by creating a copy of the input *data*, and then updating its value to values specified by *updates* at specific index positions specified by *indices*. Its output shape is the same as the shape of *data*. Note that *indices* should not have duplicate entries. That is, two or more *updates* for the same index-location is not supported.
```

***indices* is an integer tensor. Let k denote indices.shape[-1], the last dimension in the shape of *indices*.**

indices is treated as a (q-1)-dimensional tensor of k-tuples, where each k-tuple is a partial-index into *data*.

Hence, k can be a value at most the rank of *data*. When k equals rank(data), each update entry specifies an update to a single element of the tensor. When k is less than rank(data) each update entry specifies an update to a slice of the tensor.

updates is treated as a (q-1)-dimensional tensor of replacement-slice-values. Thus, the first (q-1) dimensions of updates.shape must match the first (q-1) dimensions of indices.shape. The remaining dimensions of *updates* correspond to the dimensions of the replacement-slice-values. Each replacement-slice-value is a (r-k) dimensional tensor, corresponding to the trailing (r-k) dimensions of *data*. Thus, the shape of *updates* must equal indices.shape[0:q-1] ++ data.shape[k:r-1], where ++ denotes the concatenation of shapes.

The *output* is calculated via the following equation:

output = np.copy(data) update_indices = indices.shape[:-1] for idx in np.ndindex(update_indices):

output[indices[idx]] = updates[idx]

The order of iteration in the above loop is not specified. In particular, indices should not have duplicate entries: that is, if idx1 != idx2, then indices[idx1] != indices[idx2]. This ensures that the output value does not depend on the iteration order.

This operator is the inverse of GatherND.

Example 1: ```

data = [1, 2, 3, 4, 5, 6, 7, 8] indices = [[4], [3], [1], [7]] updates = [9, 10, 11, 12] output = [1, 11, 3, 10, 9, 6, 7, 12]

```

Example 2: ```

**data = [[[1, 2, 3, 4], [5, 6, 7, 8], [8, 7, 6, 5], [4, 3, 2, 1]],** [[1, 2, 3, 4], [5, 6, 7, 8], [8, 7, 6, 5], [4, 3, 2, 1]], [[8, 7, 6, 5], [4, 3, 2, 1], [1, 2, 3, 4], [5, 6, 7, 8]], [[8, 7, 6, 5], [4, 3, 2, 1], [1, 2, 3, 4], [5, 6, 7, 8]]]

indices = [[0], [2]] updates = [[[5, 5, 5, 5], [6, 6, 6, 6], [7, 7, 7, 7], [8, 8, 8, 8]],

[[1, 1, 1, 1], [2, 2, 2, 2], [3, 3, 3, 3], [4, 4, 4, 4]]]

**output = [[[5, 5, 5, 5], [6, 6, 6, 6], [7, 7, 7, 7], [8, 8, 8, 8]],** [[1, 2, 3, 4], [5, 6, 7, 8], [8, 7, 6, 5], [4, 3, 2, 1]], [[1, 1, 1, 1], [2, 2, 2, 2], [3, 3, 3, 3], [4, 4, 4, 4]], [[8, 7, 6, 5], [4, 3, 2, 1], [1, 2, 3, 4], [5, 6, 7, 8]]]

```

Node Inputs

- **data** (T, single) – Tensor of rank r >= 1.
- **indices** (indices_constraint, single) – Tensor of rank q >= 1.
- **updates** (T, single) – Tensor of rank q + r - indices_shape[-1] - 1.

Node Outputs

- **output** (T, single) – Tensor of rank r >= 1.

Type Constraints

- **T** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`, `bool_`, `complex64`, `complex128`

- **indices_constraint** – `int64`

Parameters **`name`** – the name of the node.

class **`ONNXSelu`**(*name*, *, *alpha=1.6732631921768188*, *gamma=1.0507010221481323*)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Selu takes one input data (Tensor<T>) and produces one output data (Tensor<T>) where the scaled exponential linear unit function, *y = gamma * (alpha * e^x - alpha) for x <= 0*, *y = gamma * x for x > 0*, is applied to the tensor elementwise.

Node Inputs

- **X** (T, single) – Input tensor

Node Outputs

- **Y** (T, single) – Output tensor

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters

- **`name`** – the name of the node.
- **`alpha`** (`Optional [float]`, default=1.6732631921768188) – Coefficient of SELU default to 1.6732631921768188 (i.e., float32 approximation of 1.67326324235437728481704299167 17).
- **`gamma`** (`Optional [float]`, default=1.0507010221481323) – Coefficient of SELU default to 1.0507010221481323 (i.e., float32 approximation of 1.05070098735548049341933498529 46).

`alpha`
Coefficient of SELU default to 1.6732631921768188 (i.e., float32 approximation of 1.67326324235437728481704299167 17).

`gamma`
Coefficient of SELU default to 1.0507010221481323 (i.e., float32 approximation of 1.05070098735548049341933498529 46).

class **`ONNXShape`**(*name*, *)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Takes a tensor as input and outputs an 1D int64 tensor containing the shape of the input tensor.

Node Inputs

- **data** (T, single) – An input tensor.

Node Outputs

- **shape** (T1, single) – Shape of the input tensor

Type Constraints

- **T** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`, `bool_`, `complex64`, `complex128`
- **T1** – `int64`

Parameters **`name`** – the name of the node.

class ONNXShrink(*name*, *, *bias=0.0*, *lambd=0.5*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Shrink takes one input data (Tensor<numeric>) and produces one Tensor output, having same datatype and shape with input. It has two attributes, lambd and bias. The formula of this operator is: If x < -lambd, y = x + bias; If x > lambd, y = x - bias; Otherwise, y = 0.

Node Inputs

- **input** (T, single) – The input data as Tensor.

Node Outputs

- **output** (T, single) – The output.

Type Constraints

- **T** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`

Parameters

- **name** – the name of the node.
- **bias** (`Optional [float]`, default=0.0) – The bias value added to output. Default is 0.
- **lambd** (`Optional [float]`, default=0.5) – The lambd value for the Shrink formulation. Default is 0.5.

bias
The bias value added to output. Default is 0.

lambd
The lambd value for the Shrink formulation. Default is 0.5.

class ONNXSigmoid(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **X** (T, single) – Input tensor

Node Outputs

- **Y** (T, single) – Output tensor

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters **name** – the name of the node.

class ONNXSign(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Calculate the sign of the given input tensor element-wise. If input > 0, output 1. if input < 0, output -1. if input == 0, output 0.

Node Inputs

- **input** (T, single) – Input tensor

Node Outputs

- **output** (T, single) – The sign of the input tensor computed element-wise. It has the same shape and type of the input.

Type Constraints

- **T** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`

Parameters **name** – the name of the node.

class ONNXSin(*name*, *)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **input** (T, single) – Input tensor

Node Outputs

- **output** (T, single) – The sine of the input tensor computed element-wise

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters **name** – the name of the node.

class ONNXSinh(*name*, *)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **input** (T, single) – Input tensor

Node Outputs

- **output** (T, single) – The hyperbolic sine values of the input tensor computed element-wise

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters **name** – the name of the node.

class ONNXSize(*name*, *)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Takes a tensor as input and outputs a int64 scalar that equals to the total number of elements of the input tensor.

Node Inputs

- **data** (T, single) – An input tensor.

Node Outputs

- **size** (T1, single) – Total number of elements of the input tensor

Type Constraints

- **T** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`, `bool_`, `complex64`, `complex128`
- **T1** – `int64`

Parameters **name** – the name of the node.

class ONNXSlice(*name*, *)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Produces a slice of the input tensor along multiple axes. Similar to numpy: https://docs.scipy.org/doc/numpy/reference/arrays.indexing.html Slices uses *starts*, *ends*, *axes* and *steps* inputs to specify the start and end dimension and step for each axis in the list of axes, it uses this information to slice the input *data* tensor. If a negative value is passed for any of the start or end indices, it represents number of elements before the end of that dimension. If the value passed to start or end is larger than the *n* (the number of elements in this dimension), it represents *n*. For slicing to the end of a dimension with unknown size, it is recommended to pass in *INT_MAX* when sclicing forward and 'INT_MIN' when slicing backward. If a negative value is passed for step, it represents slicing backward. However step value cannot be 0. If *axes* are omitted, they are set to *[0, ..., ndim-1]*. If *steps* are omitted, they are set to *[1, ..., 1]* of length *len(starts)* Example 1:

data = [[1, 2, 3, 4], [5, 6, 7, 8],

] axes = [0, 1] starts = [1, 0] ends = [2, 3] steps = [1, 2] result = [

[5, 7],

]

Example 2:

data = [[1, 2, 3, 4], [5, 6, 7, 8],

] starts = [0, 1] ends = [-1, 1000] result = [

[2, 3, 4],

]

Node Inputs

- **data** (T, single) – Tensor of data to extract slices from.
- **starts** (Tind, single) – 1-D tensor of starting indices of corresponding axis in *axes*
- **ends** (Tind, single) – 1-D tensor of ending indices (exclusive) of corresponding axis in *axes*
- **axes** (Tind, optional) – 1-D tensor of axes that *starts* and *ends* apply to. Negative value means counting dimensions from the back. Accepted range is [-r, r-1] where r = rank(data).
- **steps** (Tind, optional) – 1-D tensor of slice step of corresponding axis in *axes*. Negative value means slicing backward. 'steps' cannot be 0. Defaults to 1.

Node Outputs

- **output** (T, single) – Sliced data tensor.

Type Constraints

- **T** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`, `bool_`, `complex64`, `complex128`
- **Tind** – `int32`, `int64`

Parameters **name** – the name of the node.

class ONNXSoftmax(*name*, *, *axis=1*)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

The operator computes the softmax (normalized exponential) values for each layer in the batch of the given input.

The input does not need to explicitly be a 2D vector; rather, it will be coerced into one. For an arbitrary n-dimensional tensor input in [a_0, a_1, ..., a_{k-1}, a_k, ..., a_{n-1}] and k is the axis provided, then input will be coerced into a 2-dimensional tensor with dimensions [a_0 * ... * a_{k-1}, a_k * ... * a_{n-1}]. For the default case where axis=1, this means the input tensor will be coerced into a 2D tensor of dimensions [a_0, a_1 * ... * a_{n-1}], where a_0 is often the batch size. In this situation, we must have a_0 = N and a_1 * ... * a_{n-1} = D. Each of these dimensions must be matched correctly, or else the operator will throw errors. The output tensor has the same shape and contains the softmax values of the corresponding input.

Node Inputs

- **input** (T, single) – The input tensor that's coerced into a 2D matrix of size (NxD) as described above.

Node Outputs

- **output** (T, single) – The output values with the same shape as input tensor (the original size without coercion).

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters

- **`name`** – the name of the node.
- **`axis`** (`Optional` [`int`], default=1) – Describes the axis of the inputs when coerced to 2D; defaults to one because the 0th axis most likely describes the batch_size. Negative value means counting dimensions from the back. Accepted range is [-r, r-1] where r = rank(input).

`axis`
Describes the axis of the inputs when coerced to 2D; defaults to one because the 0th axis most likely describes the batch_size. Negative value means counting dimensions from the back. Accepted range is [-r, r-1] where r = rank(input).

`class ONNXSoftmaxCrossEntropyLoss`(*name*, *, *ignore_index=None*, *reduction='mean'*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Loss function that measures the softmax cross entropy between 'scores' and 'labels'. This operator first computes a loss tensor whose shape is identical to the labels input. If the input is 2-D with shape (N, C), the loss tensor may be a N-element vector L = (l_1, l_2, ..., l_N). If the input is N-D tensor with shape (N, C, D1, D2, ..., Dk), the loss tensor L may have (N, D1, D2, ..., Dk) as its shape and L[i,][j_1][j_2]...[j_k] denotes a scalar element in L. After L is available, this operator can optionally do a reduction operator.

shape(scores): (N, C) where C is the number of classes, or (N, C, D1, D2,..., Dk), with K >= 1 in case of K-dimensional loss.

shape(labels): (N) where each value is 0 <= labels[i] <= C-1, or (N, D1, D2,..., Dk), with K >= 1 in case of K-dimensional loss.

The loss for one sample, l_i, can caculated as follows: l[i][d1][d2]...[dk] = -y[i][c][d1][d2]..[dk], where i is the index of classes.

or l[i][d1][d2]...[dk] = -y[i][c][d1][d2]..[dk] * weights[c], if 'weights' is provided.

loss is zero for the case when label-value equals ignore_index. l[i][d1][d2]...[dk] = 0, when labels[n][d1][d2]...[dk] = ignore_index

where: p = Softmax(scores) y = Log(p) c = labels[i][d1][d2]...[dk]

Finally, L is optionally reduced: If reduction = 'none', the output is L with shape (N, D1, D2, ..., Dk). If reduction = 'sum', the output is scalar: Sum(L). If reduction = 'mean', the output is scalar: ReduceMean(L), or

if weight is provided: ReduceSum(L) / ReduceSum(W), where tensor W is of shape (N, D1, D2, . . . , Dk) and W[n][d1][d2]. . . [dk] = weights[labels[i][d1][d2]. . . [dk]].

Node Inputs

- **scores** (T, single) – The predicted outputs with shape [batch_size, class_size], or [batch_size, class_size, D1, D2 , . . . , Dk], where K is the number of dimensions.
- **labels** (Tind, single) – The ground truth output tensor, with shape [batch_size], or [batch_size, D1, D2, . . . , Dk], where K is the number of dimensions. Labels element value shall be in range of [0, C). If ignore_index is specified, it may have a value outside [0, C) and the label values should either be in the range [0, C) or have the value ignore_index.
- **weights** (T, optional) – A manual rescaling weight given to each class. If given, it has to be a 1D Tensor assigning weight to each of the classes. Otherwise, it is treated as if having all ones.

Node Outputs

- **output** (T, single) – Weighted loss float Tensor. If reduction is 'none', this has the shape of [batch_size], or [batch_size, D1, D2, . . . , Dk] in case of K-dimensional loss. Otherwise, it is a scalar.
- **log_prob** (T, optional) – Log probability tensor. If the output of softmax is prob, its value is log(prob).

Type Constraints

- **T** – `float16`, `float32`, `float64`
- **Tind** – `int32`, `int64`

Parameters

- **`name`** – the name of the node.
- **`ignore_index`** (`Optional [int]`, default=None) – Specifies a target value that is ignored and does not contribute to the input gradient. It's an optional value.
- **`reduction`** (`Optional [str]`, default='mean') – Type of reduction to apply to loss: none, sum, mean(default). 'none': no reduction will be applied, 'sum': the output will be summed. 'mean': the sum of the output will be divided by the number of elements in the output.

`ignore_index`
Specifies a target value that is ignored and does not contribute to the input gradient. It's an optional value.

`reduction`
Type of reduction to apply to loss: none, sum, mean(default). 'none': no reduction will be applied, 'sum': the output will be summed. 'mean': the sum of the output will be divided by the number of elements in the output.

`class ONNXSoftplus`(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Softplus takes one input data (Tensor<T>) and produces one output data (Tensor<T>) where the softplus function, y = ln(exp(x) + 1), is applied to the tensor elementwise.

Node Inputs

- **X** (T, single) – 1D input tensor

Node Outputs

- **Y** (T, single) – 1D input tensor

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters **name** – the name of the node.

class ONNXSoftsign(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Calculates the softsign (x/(1+|x|)) of the given input tensor element-wise.

Node Inputs

- **input** (T, single) – Input tensor

Node Outputs

- **output** (T, single) – The softsign (x/(1+|x|)) values of the input tensor computed element-wise

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters **name** – the name of the node.

class ONNXSpaceToDepth(*name*, *, *blocksize*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

SpaceToDepth rearranges blocks of spatial data into depth. More specifically, this op outputs a copy of the input tensor where values from the height and width dimensions are moved to the depth dimension.

Node Inputs

- **input** (T, single) – Input tensor of [N,C,H,W], where N is the batch axis, C is the channel or depth, H is the height and W is the width.

Node Outputs

- **output** (T, single) – Output tensor of [N, C * blocksize * blocksize, H/blocksize, W/blocksize].

Type Constraints

- **T** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`, `bool_`, `complex64`, `complex128`

Parameters

- **name** – the name of the node.
- **blocksize** (`int`) – Blocks of [blocksize, blocksize] are moved.

blocksize
Blocks of [blocksize, blocksize] are moved.

class ONNXSplit(*name*, *, *axis=0*, *split=None*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Split a tensor into a list of tensors, along the specified 'axis'. Lengths of the parts can be specified using argument 'split'. Otherwise, the tensor is split to equal sized parts.

Node Inputs

- **input** (T, single) – The tensor to split

Node Outputs

- **outputs** (T, variadic) – One or more outputs forming list of tensors after splitting

Type Constraints

- **T** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`, `bool_`, `complex64`, `complex128`

Parameters

- **name** – the name of the node.
- **axis** (`Optional [int]`, default=0) – Which axis to split on. A negative value means counting dimensions from the back. Accepted range is [-rank, rank-1] where r = rank(input).
- **split** (`Optional [List [int]]`, default=None) – length of each output. Values should be >= 0.

axis
Which axis to split on. A negative value means counting dimensions from the back. Accepted range is [-rank, rank-1] where r = rank(input).

split
length of each output. Values should be >= 0.

class ONNXSqrt(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Square root takes one input data (Tensor<T>) and produces one output data (Tensor<T>) where the square root is, y = x^0.5, is applied to the tensor elementwise. If x is negative, then it will return NaN.

Node Inputs

- **X** (T, single) – Input tensor

Node Outputs

- **Y** (T, single) – Output tensor

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters **name** – the name of the node.

class ONNXSqueeze(*name*, *, *axes=None*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Remove single-dimensional entries from the shape of a tensor. Takes a parameter *axes* with a list of axes to squeeze. If *axes* is not provided, all the single dimensions will be removed from the shape. If an axis is selected with shape entry not equal to one, an error is raised.

Node Inputs

- **data** (T, single) – Tensors with at least max(dims) dimensions.

Node Outputs

- **squeezed** (T, single) – Reshaped tensor with same data as input.

Type Constraints

- **T** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`, `bool_`, `complex64`, `complex128`

Parameters

- **name** – the name of the node.

- **axes** (Optional [List [int]], default=None) – List of integers indicating the dimensions to squeeze. Negative value means counting dimensions from the back. Accepted range is [-r, r-1] where r = rank(data).

axes
List of integers indicating the dimensions to squeeze. Negative value means counting dimensions from the back. Accepted range is [-r, r-1] where r = rank(data).

class ONNXSub(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

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

Node Inputs

- **A** (T, single) – First operand.
- **B** (T, single) – Second operand.

Node Outputs

- **C** (T, single) – Result, has same element type as two inputs

Type Constraints

- **T** – `uint32`, `uint64`, `int32`, `int64`, `float16`, `float32`, `float64`

Parameters **name** – the name of the node.

class ONNXSum(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **data_0** (T, variadic) – List of tensors for sum.

Node Outputs

- **sum** (T, single) – Output tensor.

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters **name** – the name of the node.

class ONNXTan(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **input** (T, single) – Input tensor

Node Outputs

- **output** (T, single) – The tangent of the input tensor computed element-wise

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters **`name`** – the name of the node.

`class ONNXTanh`(*name*, *)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **input** (T, single) – Input tensor

Node Outputs

- **output** (T, single) – The hyperbolic tangent values of the input tensor computed element-wise

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters **`name`** – the name of the node.

`class ONNXTfIdfVectorizer`(*name*, *, *max_gram_length*, *max_skip_count*, *min_gram_length*, *mode*, *ngram_counts*, *ngram_indexes*, *pool_int64s=None*, *pool_strings=None*, *weights=None*)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

This transform extracts n-grams from the input sequence and save them as a vector. Input can be either a 1-D or 2-D tensor. For 1-D input, output is the n-gram representation of that input. For 2-D input, the output is also a 2-D tensor whose i-th row is the n-gram representation of the i-th input row. More specifically, if input shape is [C], the corresponding output shape would be [max(ngram_indexes) + 1]. If input shape is [N, C], this operator produces a [N, max(ngram_indexes) + 1]-tensor.

In contrast to standard n-gram extraction, here, the indexes of extracting an n-gram from the original sequence are not necessarily consecutive numbers. The discontinuity between indexes are controlled by the number of skips. If the number of skips is 2, we should skip two tokens when scanning through the original sequence. Let's consider an example. Assume that input sequence is [94, 17, 36, 12, 28] and the number of skips is 2. The associated 2-grams are [94, 12] and [17, 28] respectively indexed by [0, 3] and [1, 4]. If the number of skips becomes 0, the 2-grams generated are [94, 17], [17, 36], [36, 12], [12, 28] indexed by [0, 1], [1, 2], [2, 3], [3, 4], respectively.

The output vector (denoted by Y) stores the count of each n-gram; Y[ngram_indexes[i]] indicates the times that the i-th n-gram is found. The attribute ngram_indexes is used to determine the mapping between index i and the corresponding n-gram's output coordinate. If pool_int64s is [94, 17, 17, 36], ngram_indexes is [1, 0], ngram_counts=[0, 0], then the Y[0] (first element in Y) and Y[1] (second element in Y) are the counts of [17, 36] and [94, 17], respectively. An n-gram which cannot be found in pool_strings/pool_int64s should be ignored and has no effect on the output. Note that we may consider all skips up to S when generating the n-grams.

The examples used above are true if mode is "TF". If mode is "IDF", all the counts larger than 1 would be truncated to 1 and the i-th element in weights would be used to scale (by multiplication) the count of the i-th n-gram in pool. If mode is "TFIDF", this operator first computes the counts of all n-grams and then scale them by the associated values in the weights attribute.

Only one of pool_strings and pool_int64s can be set. If pool_int64s is set, the input should be an integer tensor. If pool_strings is set, the input must be a string tensor.

Node Inputs

- **X** (T, single) – Input for n-gram extraction

Node Outputs

- **Y** (T1, single) – Ngram results

Type Constraints

- **T** – `int32`, `int64`
- **T1** – `float32`

Parameters

- **`name`** – the name of the node.
- **`max_gram_length`** (`int`) – Maximum n-gram length. If this value is 3, 3-grams will be used to generate the output.
- **`max_skip_count`** (`int`) – Maximum number of items (integers/strings) to be skipped when constructing an n-gram from X. If max_skip_count=1, min_gram_length=2, max_gram_length=3, this operator may generate 2-grams with skip_count=0 and skip_count=1, and 3-grams with skip_count=0 and skip_count=1
- **`min_gram_length`** (`int`) – Minimum n-gram length. If this value is 2 and max_gram_length is 3, output may contain counts of 2-grams and 3-grams.
- **`mode`** (`str`) – The weighting criteria. It can be one of "TF" (term frequency), "IDF" (inverse document frequency), and "TFIDF" (the combination of TF and IDF)
- **`ngram_counts`** (`List [int]`) – The starting indexes of 1-grams, 2-grams, and so on in pool. It is useful when determining the boundary between two consecutive collections of n-grams. For example, if ngram_counts is [0, 17, 36], the first index (zero-based) of 1-gram/2-gram/3-gram in pool are 0/17/36. This format is essentially identical to CSR (or CSC) sparse matrix format, and we choose to use this due to its popularity.
- **`ngram_indexes`** (`List [int]`) – list of int64s (type: AttributeProto::INTS). This list is parallel to the specified 'pool_*' attribute. The i-th element in ngram_indexes indicate the coordinate of the i-th n-gram in the output tensor.
- **`pool_int64s`** (`Optional [List [int]]`, default=None) – List of int64 n-grams learned from the training set. Either this or pool_strings attributes must be present but not both. It's an 1-D tensor starting with the collections of all 1-grams and ending with the collections of n-grams. The i-th element in pool stores the n-gram that should be mapped to coordinate ngram_indexes[i] in the output vector.
- **`pool_strings`** (`Optional [List [str]]`, default=None) – List of strings n-grams learned from the training set. Either this or pool_int64s attributes must be present but not both. It's an 1-D tensor starting with the collections of all 1-grams and ending with the collections of n-grams. The i-th element in pool stores the n-gram that should be mapped to coordinate ngram_indexes[i] in the output vector.
- **`weights`** (`Optional [List [float]]`, default=None) – list of floats. This attribute stores the weight of each n-gram in pool. The i-th element in weights is the weight of the i-th n-gram in pool. Its length equals to the size of ngram_indexes. By default, weights is an all-one tensor.This attribute is used when mode is "IDF" or "TFIDF" to scale the associated word counts.

`max_gram_length`
Maximum n-gram length. If this value is 3, 3-grams will be used to generate the output.

`max_skip_count`
Maximum number of items (integers/strings) to be skipped when constructing an n-gram from X. If max_skip_count=1, min_gram_length=2, max_gram_length=3, this operator may generate 2-grams with skip_count=0 and skip_count=1, and 3-grams with skip_count=0 and skip_count=1

min_gram_length
Minimum n-gram length. If this value is 2 and max_gram_length is 3, output may contain counts of 2-grams and 3-grams.

mode
The weighting criteria. It can be one of "TF" (term frequency), "IDF" (inverse document frequency), and "TFIDF" (the combination of TF and IDF)

ngram_counts
The starting indexes of 1-grams, 2-grams, and so on in pool. It is useful when determining the boundary between two consecutive collections of n-grams. For example, if ngram_counts is [0, 17, 36], the first index (zero-based) of 1-gram/2-gram/3-gram in pool are 0/17/36. This format is essentially identical to CSR (or CSC) sparse matrix format, and we choose to use this due to its popularity.

ngram_indexes
list of int64s (type: AttributeProto::INTS). This list is parallel to the specified 'pool_*' attribute. The i-th element in ngram_indexes indicate the coordinate of the i-th n-gram in the output tensor.

pool_int64s
List of int64 n-grams learned from the training set. Either this or pool_strings attributes must be present but not both. It's an 1-D tensor starting with the collections of all 1-grams and ending with the collections of n-grams. The i-th element in pool stores the n-gram that should be mapped to coordinate ngram_indexes[i] in the output vector.

pool_strings
List of strings n-grams learned from the training set. Either this or pool_int64s attributes must be present but not both. It's an 1-D tensor starting with the collections of all 1-grams and ending with the collections of n-grams. The i-th element in pool stores the n-gram that should be mapped to coordinate ngram_indexes[i] in the output vector.

weights
list of floats. This attribute stores the weight of each n-gram in pool. The i-th element in weights is the weight of the i-th n-gram in pool. Its length equals to the size of ngram_indexes. By default, weights is an all-one tensor.This attribute is used when mode is "IDF" or "TFIDF" to scale the associated word counts.

class ONNXThresholdedRelu(*name*, *, *alpha=1.0*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

ThresholdedRelu takes one input data (Tensor<T>) and produces one output data (Tensor<T>) where the rectified linear function, y = x for x > alpha, y = 0 otherwise, is applied to the tensor elementwise.

Node Inputs

- **X** (T, single) – Input tensor

Node Outputs

- **Y** (T, single) – Output tensor

Type Constraints

- **T** – `float16`, `float32`, `float64`

Parameters

- **name** – the name of the node.
- **alpha** (`Optional` [`float`], default=1.0) – Threshold value

alpha
Threshold value

class ONNXTile(*name*, *)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Constructs a tensor by tiling a given tensor. This is the same as function *tile* in Numpy, but no broadcast. For example A = [[1, 2], [3, 4]], B = [1, 2], tile(A, B) = [[1, 2, 1, 2], [3, 4, 3, 4]]

Node Inputs

- **input** (T, single) – Input tensor of any shape.
- **repeats** (T1, single) – 1D int64 tensor of the same length as input's dimension number, includes numbers of repeated copies along input's dimensions.

Node Outputs

- **output** (T, single) – Output tensor of the same dimension and type as tensor input. output_dim[i] = input_dim[i] * repeats[i]

Type Constraints

- **T** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`, `bool_`, `complex64`, `complex128`
- **T1** – `int64`

Parameters **name** – the name of the node.

class ONNXTopK(*name*, *, *axis=- 1*, *largest=1*, *sorted=1*)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Retrieve the top-K largest or smallest elements along a specified axis. Given an input tensor of shape [a_1, a_2, . . . , a_n, r] and integer argument k, return two outputs:

-Value tensor of shape [a_1, a_2, . . . , a_{axis-1}, k, a_{axis+1}, . . . a_n] which contains the values of the top k elements along the specified axis

-Index tensor of shape [a_1, a_2, . . . , a_{axis-1}, k, a_{axis+1}, . . . a_n] which contains the indices of the top k elements (original indices from the input tensor).

If "largest" is 1 (the default value) then the k largest elements are returned. If "sorted" is 1 (the default value) then the resulting k elements will be sorted. If "sorted" is 0, order of returned 'Values' and 'Indices' are undefined.

Given two equivalent values, this operator uses the indices along the axis as a tiebreaker. That is, the element with the lower index will appear first.

Node Inputs

- **X** (T, single) – Tensor of shape [a_1, a_2, . . . , a_n, r]
- **K** (K_constraint, single) – A 1-D tensor containing a single positive value corresponding to the number of top elements to retrieve

Node Outputs

- **Values** (T, single) – Tensor of shape [a_1, a_2, . . . , a_{axis-1}, k, a_{axis+1}, . . . a_n] containing top K values from the input tensor
- **Indices** (I, single) – Tensor of shape [a_1, a_2, . . . , a_{axis-1}, k, a_{axis+1}, . . . a_n] containing the corresponding input tensor indices for the top K values.

Type Constraints

- **T** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`
- **I** – `int64`

- **K_constraint** – `int64`

Parameters

- **name** – the name of the node.
- **axis** (`Optional` [`int`], default=-1) – Dimension on which to do the sort. Negative value means counting dimensions from the back. Accepted range is [-r, r-1] where r = rank(input).
- **largest** (`Optional` [`int`], default=1) – Whether to return the top-K largest or smallest elements.
- **sorted** (`Optional` [`int`], default=1) – Whether to return the elements in sorted order.

axis
Dimension on which to do the sort. Negative value means counting dimensions from the back. Accepted range is [-r, r-1] where r = rank(input).

largest
Whether to return the top-K largest or smallest elements.

sorted
Whether to return the elements in sorted order.

class ONNXTranspose(*name*, *, *perm=None*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

Node Inputs

- **data** (T, single) – An input tensor.

Node Outputs

- **transposed** (T, single) – Transposed output.

Type Constraints

- **T** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`, `bool_`, `complex64`, `complex128`

Parameters

- **name** – the name of the node.
- **perm** (`Optional` [`List` [`int`]], default=None) – A list of integers. By default, reverse the dimensions, otherwise permute the axes according to the values given.

perm
A list of integers. By default, reverse the dimensions, otherwise permute the axes according to the values given.

class ONNXTreeEnsembleClassifier(*name*, *, *base_values=None*, *class_ids=None*, *class_nodeids=None*, *class_treeids=None*, *class_weights=None*, *classlabels_int64s=None*, *classlabels_strings=None*, *nodes_falsenodeids=None*, *nodes_featureids=None*, *nodes_hitrates=None*, *nodes_missing_value_tracks_true=None*, *nodes_modes=None*, *nodes_nodeids=None*, *nodes_treeids=None*, *nodes_truenodeids=None*, *nodes_values=None*, *post_transform='NONE'*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Tree Ensemble classifier. Returns the top class for each of N inputs.
 The attributes named 'nodes_X' form a sequence of tuples, associated by index into the sequences, which must all be of equal length. These tuples define the nodes.
 Similarly, all fields prefixed with '**class_**' are tuples of votes at the leaves. A leaf may have multiple votes, where each vote is weighted by the associated class_weights index.
 One and only one of classlabels_strings or classlabels_int64s will be defined. The class_ids are indices into this list.

Node Inputs

- **X** (T1, single) – Input of shape [N,F]

Node Outputs

- **Y** (T2, single) – N, Top class for each point
- **Z** (Z_constraint, single) – The class score for each class, for each point, a tensor of shape [N,E].

Type Constraints

- **T1** – `float32`, `float64`, `int64`, `int32`
- **T2** – `int64`
- **Z_constraint** – `float32`

Parameters

- **`name`** – the name of the node.
- **`base_values`** (`Optional` [`List` [`float`]], default=None) – Base values for classification, added to final class score; the size must be the same as the classes or can be left unassigned (assumed 0)
- **`class_ids`** (`Optional` [`List` [`int`]], default=None) – The index of the class list that each weight is for.
- **`class_nodeids`** (`Optional` [`List` [`int`]], default=None) – node id that this weight is for.
- **`class_treeids`** (`Optional` [`List` [`int`]], default=None) – The id of the tree that this node is in.
- **`class_weights`** (`Optional` [`List` [`float`]], default=None) – The weight for the class in class_id.
- **`classlabels_int64s`** (`Optional` [`List` [`int`]], default=None) – Class labels if using integer labels.
One and only one of the 'classlabels_*' attributes must be defined.
- **`classlabels_strings`** (`Optional` [`List` [`str`]], default=None) – Class labels if using string labels.
One and only one of the 'classlabels_*' attributes must be defined.
- **`nodes_falsenodeids`** (`Optional` [`List` [`int`]], default=None) – Child node if expression is false.
- **`nodes_featureids`** (`Optional` [`List` [`int`]], default=None) – Feature id for each node.
- **`nodes_hitrates`** (`Optional` [`List` [`float`]], default=None) – Popularity of each node, used for performance and may be omitted.
- **`nodes_missing_value_tracks_true`** (`Optional` [`List` [`int`]], default=None) – For each node, define what to do in the presence of a missing value: if a value is missing

(NaN), use the 'true' or 'false' branch based on the value in this array.
This attribute may be left undefined, and the defalt value is false (0) for all nodes.

- **nodes_modes** (Optional [List [str]], default=None) – The node kind, that is, the comparison to make at the node. There is no comparison to make at a leaf node.
One of 'BRANCH_LEQ', 'BRANCH_LT', 'BRANCH_GTE', 'BRANCH_GT', 'BRANCH_EQ', 'BRANCH_NEQ', 'LEAF'
- **nodes_nodeids** (Optional [List [int]], default=None) – Node id for each node. Ids may restart at zero for each tree, but it not required to.
- **nodes_treeids** (Optional [List [int]], default=None) – Tree id for each node.
- **nodes_truenodeids** (Optional [List [int]], default=None) – Child node if expression is true.
- **nodes_values** (Optional [List [float]], default=None) – Thresholds to do the splitting on for each node.
- **post_transform** (Optional [str], default='NONE') – Indicates the transform to apply to the score.
 One of 'NONE,' 'SOFTMAX,' 'LOGISTIC,' 'SOFTMAX_ZERO,' or 'PROBIT.'

base_values
Base values for classification, added to final class score; the size must be the same as the classes or can be left unassigned (assumed 0)

class_ids
The index of the class list that each weight is for.

class_nodeids
node id that this weight is for.

class_treeids
The id of the tree that this node is in.

class_weights
The weight for the class in class_id.

classlabels_int64s
Class labels if using integer labels.
One and only one of the 'classlabels_*' attributes must be defined.

classlabels_strings
Class labels if using string labels.
One and only one of the 'classlabels_*' attributes must be defined.

nodes_falsenodeids
Child node if expression is false.

nodes_featureids
Feature id for each node.

nodes_hitrates
Popularity of each node, used for performance and may be omitted.

nodes_missing_value_tracks_true
For each node, define what to do in the presence of a missing value: if a value is missing (NaN), use the 'true' or 'false' branch based on the value in this array.
This attribute may be left undefined, and the defalt value is false (0) for all nodes.

nodes_modes
The node kind, that is, the comparison to make at the node. There is no comparison to make

at a leaf node.
One of 'BRANCH_LEQ', 'BRANCH_LT', 'BRANCH_GTE', 'BRANCH_GT', 'BRANCH_EQ', 'BRANCH_NEQ', 'LEAF'

nodes_nodeids
Node id for each node. Ids may restart at zero for each tree, but it not required to.

nodes_treeids
Tree id for each node.

nodes_truenodeids
Child node if expression is true.

nodes_values
Thresholds to do the splitting on for each node.

post_transform
Indicates the transform to apply to the score.
 One of 'NONE,' 'SOFTMAX,' 'LOGISTIC,' 'SOFTMAX_ZERO,' or 'PROBIT.'

class ONNXTreeEnsembleRegressor(*name*, ***, *aggregate_function='SUM'*, *base_values=None*, *n_targets=None*, *nodes_falsenodeids=None*, *nodes_featureids=None*, *nodes_hitrates=None*, *nodes_missing_value_tracks_true=None*, *nodes_modes=None*, *nodes_nodeids=None*, *nodes_treeids=None*, *nodes_truenodeids=None*, *nodes_values=None*, *post_transform='NONE'*, *target_ids=None*, *target_nodeids=None*, *target_treeids=None*, *target_weights=None*)

Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Tree Ensemble regressor. Returns the regressed values for each input in N.
 All args with **nodes_** are fields of a tuple of tree nodes, and it is assumed they are the same length, and an index i will decode the tuple across these inputs. Each node id can appear only once for each tree id.
 All fields prefixed with **target_** are tuples of votes at the leaves.
 A leaf may have multiple votes, where each vote is weighted by the associated target_weights index.
 All trees must have their node ids start at 0 and increment by 1.
 Mode enum is BRANCH_LEQ, BRANCH_LT, BRANCH_GTE, BRANCH_GT, BRANCH_EQ, BRANCH_NEQ, LEAF

Node Inputs

- **X** (T, single) – Input of shape [N,F]

Node Outputs

- **Y** (Y_constraint, single) – N classes

Type Constraints

- **T** – `float32`, `float64`, `int64`, `int32`
- **Y_constraint** – `float32`

Parameters

- **name** – the name of the node.
- **aggregate_function** (`Optional [str]`, default='SUM') – Defines how to aggregate leaf values within a target.
One of 'AVERAGE,' 'SUM,' 'MIN,' 'MAX.'
- **base_values** (`Optional [List [float]]`, default=None) – Base values for classification, added to final class score; the size must be the same as the classes or can be left unassigned (assumed 0)

- **n_targets** (Optional [int], default=None) – The total number of targets.
- **nodes_falsenodeids** (Optional [List [int]], default=None) – Child node if expression is false
- **nodes_featureids** (Optional [List [int]], default=None) – Feature id for each node.
- **nodes_hitrates** (Optional [List [float]], default=None) – Popularity of each node, used for performance and may be omitted.
- **nodes_missing_value_tracks_true** (Optional [List [int]], default=None) – For each node, define what to do in the presence of a NaN: use the 'true' (if the attribute value is 1) or 'false' (if the attribute value is 0) branch based on the value in this array.
This attribute may be left undefined and the defalt value is false (0) for all nodes.
- **nodes_modes** (Optional [List [str]], default=None) – The node kind, that is, the comparison to make at the node. There is no comparison to make at a leaf node.
One of 'BRANCH_LEQ', 'BRANCH_LT', 'BRANCH_GTE', 'BRANCH_GT', 'BRANCH_EQ', 'BRANCH_NEQ', 'LEAF'
- **nodes_nodeids** (Optional [List [int]], default=None) – Node id for each node. Node ids must restart at zero for each tree and increase sequentially.
- **nodes_treeids** (Optional [List [int]], default=None) – Tree id for each node.
- **nodes_truenodeids** (Optional [List [int]], default=None) – Child node if expression is true
- **nodes_values** (Optional [List [float]], default=None) – Thresholds to do the splitting on for each node.
- **post_transform** (Optional [str], default='NONE') – Indicates the transform to apply to the score.
One of 'NONE,' 'SOFTMAX,' 'LOGISTIC,' 'SOFTMAX_ZERO,' or 'PROBIT'
- **target_ids** (Optional [List [int]], default=None) – The index of the target that each weight is for
- **target_nodeids** (Optional [List [int]], default=None) – The node id of each weight
- **target_treeids** (Optional [List [int]], default=None) – The id of the tree that each node is in.
- **target_weights** (Optional [List [float]], default=None) – The weight for each target

aggregate_function
Defines how to aggregate leaf values within a target.
One of 'AVERAGE,' 'SUM,' 'MIN,' 'MAX.'

base_values
Base values for classification, added to final class score; the size must be the same as the classes or can be left unassigned (assumed 0)

n_targets
The total number of targets.

nodes_falsenodeids
Child node if expression is false

nodes_featureids
Feature id for each node.

nodes_hitrates
Popularity of each node, used for performance and may be omitted.

nodes_missing_value_tracks_true
For each node, define what to do in the presence of a NaN: use the 'true' (if the attribute value is 1) or 'false' (if the attribute value is 0) branch based on the value in this array.
This attribute may be left undefined and the defalt value is false (0) for all nodes.

nodes_modes
The node kind, that is, the comparison to make at the node. There is no comparison to make at a leaf node.
One of 'BRANCH_LEQ', 'BRANCH_LT', 'BRANCH_GTE', 'BRANCH_GT', 'BRANCH_EQ', 'BRANCH_NEQ', 'LEAF'

nodes_nodeids
Node id for each node. Node ids must restart at zero for each tree and increase sequentially.

nodes_treeids
Tree id for each node.

nodes_truenodeids
Child node if expression is true

nodes_values
Thresholds to do the splitting on for each node.

post_transform
Indicates the transform to apply to the score.
One of 'NONE,' 'SOFTMAX,' 'LOGISTIC,' 'SOFTMAX_ZERO,' or 'PROBIT'

target_ids
The index of the target that each weight is for

target_nodeids
The node id of each weight

target_treeids
The id of the tree that each node is in.

target_weights
The weight for each target

class ONNXUnique(*name*, *, *axis=None*, *sorted=1*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Find the unique elements of a tensor. When an optional attribute 'axis' is provided, unique subtensors sliced along the 'axis' are returned. Otherwise the input tensor is flattened and unique values of the flattened tensor are returned.

This operator returns the unique values or sliced unique subtensors of the input tensor and three optional outputs. The first output tensor 'Y' contains all unique values or subtensors of the input. The second optional output tensor 'indices' contains indices of 'Y' elements' first occurance in 'X'.. The third optional output tensor 'inverse_indices' contains, for elements of 'X', its corresponding indices in 'Y'. ". The fourth optional output tensor 'counts' contains the count of each element of 'Y' in the input.

Outputs are either sorted in ascending order or optionally in the order of the first occurrence of the values in the input.

https://docs.scipy.org/doc/numpy/reference/generated/numpy.unique.html

Example 1: input_X = [2, 1, 1, 3, 4, 3] attribute_sorted = 0 attribute_axis = None output_Y = [2, 1, 3, 4] output_indices = [0, 1, 3, 4] output_inverse_indices = [0, 1, 1, 2, 3, 2] output_counts = [1, 2, 2, 1]

Example 2: input_X = [[1, 3], [2, 3]] attribute_sorted = 1 attribute_axis = None output_Y = [1, 2, 3] output_indices = [0, 2, 1] output_inverse_indices = [0, 2, 1, 2] output_counts = [1, 1, 2]

Example 3: input_X = [[1, 0, 0], [1, 0, 0], [2, 3, 4]] attribute_sorted = 1 attribute_axis = 0 output_Y = [[1, 0, 0], [2, 3, 4]] output_indices = [0, 2] output_inverse_indices = [0, 0, 1] output_counts = [2, 1]

Example 4:

input_x = [[[1., 1.], [0., 1.], [2., 1.], [0., 1.]], [[1., 1.], [0., 1.], [2., 1.], [0., 1.]]]

attribute_sorted = 1 attribute_axis = 1

intermediate data are presented below for better understanding:

there are 4 subtensors sliced along axis 1 of input_x (shape = (2, 4, 2)): A: [[1, 1], [1, 1]],

[[0, 1], [0, 1]], [[2, 1], [2, 1]], [[0, 1], [0, 1]].

there are 3 unique subtensors: [[1, 1], [1, 1]], [[0, 1], [0, 1]], [[2, 1], [2, 1]].

sorted unique subtensors: B: [[0, 1], [0, 1]],

[[1, 1], [1, 1]], [[2, 1], [2, 1]].

output_Y is constructed from B: [[[0. 1.], [1. 1.], [2. 1.]],

[[0. 1.], [1. 1.], [2. 1.]]]

output_indices is to map from B to A: [1, 0, 2]

output_inverse_indices is to map from A to B: [1, 0, 2, 0]

output_counts = [2 1 1]

Node Inputs

- **X** (T, single) – A N-D input tensor that is to be processed.

Node Outputs

- **Y** (T, single) – A tensor of the same type as 'X' containing all the unique values or subtensors sliced along a provided 'axis' in 'X', either sorted or maintained in the same order they occur in input 'X'
- **indices** (indices_constraint, optional) – A 1-D INT64 tensor containing indices of 'Y' elements' first occurance in 'X'. When 'axis' is provided, it contains indices to subtensors in input 'X' on the 'axis'. When 'axis' is not provided, it contains indices to values in the flattened input tensor.
- **inverse_indices** (inverse_indices_constraint, optional) – A 1-D INT64 tensor containing, for elements of 'X', its corresponding indices in 'Y'. When 'axis' is provided, it contains indices to subtensors in output 'Y' on the 'axis'. When 'axis' is not provided, it contains indices to values in output 'Y'.
- **counts** (counts_constraint, optional) – A 1-D INT64 tensor containing the count of each element of 'Y' in input 'X'

Type Constraints

- **T** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`, `bool_`, `complex64`, `complex128`
- **indices_constraint** – `int64`
- **inverse_indices_constraint** – `int64`
- **counts_constraint** – `int64`

Parameters

- **name** – the name of the node.
- **axis** (Optional [int], default=None) – (Optional) The dimension to apply unique. If not specified, the unique elements of the flattened input are returned. Negative value means counting dimensions from the back. Accepted range is [-r, r-1] where r = rank(input).
- **sorted** (Optional [int], default=1) – (Optional) Whether to sort the unique elements in ascending order before returning as output. Must be one of 0, or 1 (default).

axis
(Optional) The dimension to apply unique. If not specified, the unique elements of the flattened input are returned. Negative value means counting dimensions from the back. Accepted range is [-r, r-1] where r = rank(input).

sorted
(Optional) Whether to sort the unique elements in ascending order before returning as output. Must be one of 0, or 1 (default).

class ONNXUnsqueeze(*name*, *, *axes*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Insert single-dimensional entries to the shape of an input tensor (*data*). Takes one required argument *axes* - which contains a list of dimension indices and this operator will insert a dimension of value *1* into the corresponding index of the output tensor (*expanded*).

For example: Given an input tensor (*data*) of shape [3, 4, 5], then Unsqueeze(data, axes=[0, 4]) outputs a tensor (*expanded*) containing same data as *data* but with shape [1, 3, 4, 5, 1].

The attribute *axes* should not contain any duplicate entries. It is an error if it contains duplicates. The rank of the output tensor (*output_rank*) is the rank of the input tensor (*data*) plus the number of values in *axes*. Each value in *axes* should be within the (inclusive) range [-output_rank , output_rank - 1]. The order of values in *axes* does not matter and can come in any order.

Node Inputs

- **data** (T, single) – Original tensor

Node Outputs

- **expanded** (T, single) – Reshaped tensor with same data as input.

Type Constraints

- **T** – uint8, uint16, uint32, uint64, int8, int16, int32, int64, float16, float32, float64, bool_, complex64, complex128

Parameters

- **name** – the name of the node.
- **axes** (List [int]) – List of integers indicating the dimensions to be inserted. Negative value means counting dimensions from the back. Accepted range is [-r, r-1] where r = rank(expanded).

axes
List of integers indicating the dimensions to be inserted. Negative value means counting dimensions from the back. Accepted range is [-r, r-1] where r = rank(expanded).

class ONNXUpsample(*name*, *, *mode='nearest'*)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Upsample the input tensor. Each dimension value of the output tensor is:

output_dimension = floor(input_dimension * scale).

Node Inputs

- **X** (T, single) – N-D tensor
- **scales** (scales_constraint, single) – The scale array along each dimension. It takes value greater than or equal to 1. The number of elements of 'scales' should be the same as the rank of input 'X'.

Node Outputs

- **Y** (T, single) – N-D tensor after resizing

Type Constraints

- **T** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`, `bool_`, `complex64`, `complex128`
- **scales_constraint** – `float32`

Parameters

- **name** – the name of the node.
- **mode** (`Optional [str]`, default='nearest') – Two interpolation modes: nearest (default), and linear (including bilinear, trilinear, etc)

mode
Two interpolation modes: nearest (default), and linear (including bilinear, trilinear, etc)

class ONNXWhere(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

Return elements, either from X or Y, depending on condition (with Numpy-style broadcasting support). Where behaves like numpy.where with three parameters: https://docs.scipy.org/doc/numpy/reference/generated/numpy.where.html

Node Inputs

- **condition** (B, single) – When True (nonzero), yield X, otherwise yield Y
- **X** (T, single) – values selected at indices where condition is True
- **Y** (T, single) – values selected at indices where condition is False

Node Outputs

- **output** (T, single) – Tensor of shape equal to the broadcasted shape of condition, X, and Y.

Type Constraints

- **B** – `bool_`
- **T** – `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float16`, `float32`, `float64`, `bool_`, `complex64`, `complex128`

Parameters **name** – the name of the node.

class ONNXXor(*name*, *)
Bases: *daceml.onnx.nodes.onnx_op.ONNXOp*

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

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

Node Inputs

- **A** (T, single) – First input operand for the logical operator.
- **B** (T, single) – Second input operand for the logical operator.

Node Outputs

- **C** (T1, single) – Result tensor.

Type Constraints

- **T** – `bool_`
- **T1** – `bool_`

Parameters **`name`** – the name of the node.

CHAPTER

NINE

DACEML.PYTORCH

class DaceModule(*module*, *dummy_inputs=None*, *cuda=None*, *training=False*, *backward=False*, *apply_strict=True*, *auto_optimize=True*, *debug_transients=False*, *sdfg_name=None*)

Bases: `torch.nn.modules.module.Module`

A wrapper that converts a PyTorch `nn.Module` to a PyTorch compatible data-centric `nn.Module`.

Parameters

- **module** (`Module`) – the model to wrap.
- **dummy_inputs** (`Optional[Tuple[Tensor]]`) – a tuple of tensors to use as input when tracing `model`.
- **cuda** (`Optional[bool]`) – if `True`, the module will execute using CUDA. If `None`, it will be detected from the `module`.
- **training** (`bool`) – whether to use train mode when tracing `model`.
- **backward** – whether to enable the backward pass.
- **apply_strict** (`bool`) – whether to apply strict transforms after conversion (this generally improves performance, but can be slow).
- **sdfg_name** (`Optional[str]`) – the name to give to the sdfg (defaults to `dace_model`).
- **auto_optimize** (`bool`) – whether to apply automatic optimizations.
- **debug_transients** (`bool`) – if True, the module will have all transients as outputs.

Example

```
>>> from daceml.pytorch import DaceModule
>>> class MyModule(nn.Module):
...     def forward(self, x):
...         x = torch.log(x)
...         x = torch.sqrt(x)
...         return x
>>> module = MyModule()
>>> module(torch.ones(2))
tensor([0., 0.])
>>> dace_module = DaceModule(module)
>>> dace_module(torch.ones(2))
tensor([0., 0.])
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

append_post_autodiff_hook(*name*, *func*)

Parameters

- **name** (*str*) –
- **func** (*Callable[[dace.sdfg.sdfg.SDFG, dace.sdfg.sdfg.SDFG], None]*) –

append_post_compile_hook(*name*, *func*)

Parameters

- **name** (*str*) –
- **func** (*Callable[[dace.codegen.compiled_sdfg.CompiledSDFG], None]*) –

append_post_onnx_hook(*name*, *func*)

Parameters

- **name** (*str*) –
- **func** (*Callable[[DaceModule], None]*) –

forward(**actual_inputs*)
Execute the forward pass using the traced `module`.

post_autodiff_hooks: OrderedDict[str, Callable[[dace.SDFG, dace.SDFG], None]]
hooks that are executed after the backpropagation sdfg has been created

post_compile_hooks: OrderedDict[str, Callable[[compiled_sdfg.CompiledSDFG], None]]
hooks that are executed after the sdfg is compiled

post_onnx_hooks: OrderedDict[str, Callable[[*DaceModule*], None]]
hooks that are executed after onnx graph is imported to an SDFG

prepend_post_autodiff_hook(*name*, *func*)

Parameters

- **name** (*str*) –
- **func** (*Callable[[dace.sdfg.sdfg.SDFG, dace.sdfg.sdfg.SDFG], None]*) –

prepend_post_compile_hook(*name*, *func*)

Parameters

- **name** (*str*) –
- **func** (*Callable[[dace.codegen.compiled_sdfg.CompiledSDFG], None]*) –

prepend_post_onnx_hook(*name*, *func*)

Parameters

- **name** (*str*) –
- **func** (*Callable[[DaceModule], None]*) –

reset_sdfg()
Clear the sdfg so that optimizations are reapplied.

training: bool

@dace_module(*moduleclass*, *dummy_inputs=None*, *cuda=None*, *training=False*, *backward=False*, *apply_strict=True*, *auto_optimize=True*, *sdfg_name=None*, *debug_transients=False*)

Decorator to apply on a definition of a `torch.nn.Module` to convert it to a data-centric module upon construction.

Example

```
>>> from daceml.pytorch import dace_module
>>> @dace_module
... class MyDecoratedModule(nn.Module):
...     def forward(self, x):
...         x = torch.log(x)
...         x = torch.sqrt(x)
...         return x
>>> module = MyDecoratedModule()
>>> module(torch.ones(2))
tensor([0., 0.])
```

Parameters

- **moduleclass** – the model to wrap.
- **dummy_inputs** (`Optional[Tuple[Tensor]]`) – a tuple of tensors to use as input when tracing `model`.
- **cuda** (`Optional[bool]`) – if `True`, the module will execute using CUDA. If `None`, it will be detected from the `module`.
- **training** (`bool`) – whether to use train mode when tracing `model`.
- **backward** – whether to enable the backward pass.
- **apply_strict** (`bool`) – whether to apply strict transforms after conversion (this generally improves performance, but can be slow).
- **auto_optimize** (`bool`) – whether to apply automatic optimizations.
- **sdfg_name** (`Optional[str]`) – the name to give to the sdfg (defaults to `dace_model`).
- **debug_transients** (`bool`) – if True, the module will have all transients as outputs.

Return type `Type[DaceModule]`

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex

PYTHON MODULE INDEX

d

INDEX

Symbols

A

B

C

D

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z