
modeci-mdf

ModECI project

Mar 06, 2023

CONTENTS

1	ModECI Model Description Format (MDF)	3
2	Installation	7
3	ModECI contributors	11
4	Specification of ModECI v0.4	13
5	Model	15
6	Graph	17
7	Node	19
8	InputPort	21
9	Function	23
10	Parameter	25
11	ParameterCondition	27
12	OutputPort	29
13	Edge	31
14	Condition	33
15	ConditionSet	35
16	MDF Examples	37
17	Interactions between MDF and ACT-R	39
18	Interactions between NeuroML and MDF	41
19	ONNX MDF Converter	55
20	Interactions between PsyNeuLink and MDF	59
21	PyTorch and MDF	61
22	Interactions between MDF and Quantum computing technologies	73

23	MDF in WebGME	75
24	Specification of standard functions in ModECI v0.4	77
25	modeci_mdf	115
26	Indices and tables	225
	Python Module Index	227
	Index	229

MDF is an open source, community-supported standard and associated library of tools for expressing computational models in a form that allows them to be exchanged between diverse programming languages and execution environments. The overarching aim is to provide a common format for models across **computational neuroscience, cognitive science and machine learning**.

It consists of a specification for expressing models in serialized form (currently JSON, YAML or BSON representations, though others such as HDF5 are planned) and a set of Python tools for implementing a model described using MDF. The serialized formats can be used when importing a model into a supported target environment to execute it; and, conversely, when exporting a model built in a supported environment so that it can be re-used in other environments.

MODECI MODEL DESCRIPTION FORMAT (MDF)

[Click here for the full MDF documentation](#)

Note: MDF is still in development! See the [open issues related to the specification](#) or go [here](#) to get in contact regarding MDF. *The MDF format was first proposed following a meeting organised at Princeton in July 2019 by Russ Poldrack of the Center for Reproducible Neuroscience (CRN) at Stanford and the [Brain Imaging Data Standard \(BIDS\)](#) initiative. For more on the previous work in this area, see [here](#).*

1.1 Overview

MDF is an open source, community-supported standard and associated library of tools for expressing computational models in a form that allows them to be exchanged between diverse programming languages and execution environments. The overarching aim is to provide a common format for models across [computational neuroscience](#), [cognitive science](#) and [machine learning](#).

It consists of a specification for expressing models in serialized formats (currently [JSON](#), [YAML](#) and [BSON](#) representations are supported, though others such as [HDF5](#) are planned) and a set of Python tools for implementing a model described using MDF. The serialized formats can be used when importing a model into a supported target environment to execute it; and, conversely, when exporting a model built in a supported environment so that it can be re-used in other environments.

The MDF Python API can be used to create or load an MDF model for inspection and validation. It also includes a basic [execution engine](#) for simulating models in the format. However, this is not intended to provide a efficient, general-purpose simulation environment, nor is MDF intended as a programming language. Rather, the primary purpose of the Python API is to facilitate and validate the exchange of models between existing environments that serve different communities. Accordingly, these Python tools include bi-directional support for importing to and exporting from widely-used programming environments in a range of disciplines, and for easily extending these to other environments.

1.2 Development

The implementation and dissemination of the MDF language and associated tools is being carried out by the [Model Exchange and Convergence Initiative \(ModECI\)](#), which has been supported by the [NSF Convergence Accelerator Program](#) (Track D: AI-Driven Innovation via Data and Model Sharing), as a publicly accessible [open-source project](#). The initial design has been informed by a [series of workshops](#) involving developers of key software environments and other stakeholders in machine learning, cognitive science and neuroscience. Future workshops will address broadening of support to other domains in basic and applied science and technology development (e.g., population biology, medical informatics, structural and environmental monitoring, and complex systems control). Environments for which support is currently being developed include [PyTorch](#), [ONNX](#), [WebGME](#), [NeuroML](#), [PsyNeuLink](#), and [ACT-R](#).

Successful interfacing of MDF to existing disciplinary standards (such as [ONNX](#) in machine learning, and [NeuroML](#) in neuroscience) as well as general-purpose simulation environments (such as [WebGME](#)) will permit bridging between these environments, and translation to the broader set of environments supported by those standards (such as [Tensorflow](#) & [Keras](#) in the case of ONNX, and [The Virtual Brain](#) and [SONATA](#) in the case of NeuroML). Initial investigations have also taken place, in collaboration with projects in the NSF Accelerator Track C (Quantum Technology), to use MDF for facilitating the implementation of computational models on [quantum hardware](#).

1.2.1 The core elements of the MDF standard

Models The highest level construct in MDF is a model that consists of one or more **graphs** and model attributes. The former describe the operational features of the model (its structure and execution), while the latter provide additional information (metadata) useful for executing, evaluating, testing or visualizing it.

Graphs A graph specifies the structure and process flow of a **model**. The most fundamental element of a graph is a **node**, which specifies some unit of computation in terms of its **parameters** and **functions**. Nodes are connected to other nodes via directed **edges**, which, in the absence of additional **conditions**, define the computational flow of the model.

Nodes These define the core elements of computation in a **graph**, that receive and transmit information via their **input** and **output ports**. In general, ports represent points of contact between a **node** and the **edges** that connect it to other nodes.

Output Ports An output port is the starting point of the data transmission process. After processing the information in a **node**, an output port is used to begin the transmission of information to the next **node** through **edges**.

Edges These transmit information from the **output port** of one **node** to the **input port** of another, collectively defining a **graph's** topology. Edges may contain weights that can operate on the information they carry.

Input Ports An input port is the endpoint of the data transmission process. It receives the information transmitted through an **edge** and inputs it to the next **node** for further processing.

Conditions These are a core and distinctive element of the MDF specification, that complement other computational graph-based formats by providing a high-level set of descriptors for specifying conditional execution of **nodes**. This allows models with relatively complex execution requirements (e.g., containing cycles, branches, and/or temporal dependencies) to be expressed as **graphs** in a sufficiently abstract form that facilitates exchange among high-level modeling environments without requiring that they be “lowered” to and then recovered from more elaborated procedural descriptions.

Parameters Attributes that determine the configuration and operation of **nodes** and **edges**, can be defined in the MDF using parameters. In the case of parameters specifying large data structures (e.g., weight-matrices), arrays in widely used formats (e.g. numpy arrays, TensorFlow tensors) can be used, and serialisation in portable binary formats (e.g. BSON) is supported. Parameters can either be fixed values, which don't change when the **node** is executed, or can change over time (stateful parameters).

Functions A single value which is evaluated as a function of values on **input ports** and other functions and **parameters**. A key distinction with **parameters** is that a function is always stateless.

Model metadata There is the ability to add “metadata” to the **model**, **graph**, **nodes** and many of their sub elements which provide additional information about that element. While the metadata should not be essential to the mathematical description of the behavior/structure of the element, it could be useful for human interpretability of its function/purpose, or used when it is mapped to a specific application for simulation/visualization. Metadata can be added to the top level model to specify contact information, citations, acknowledgements, pointers to sample data and benchmark results, and environments in which the specified model was originally implemented and any that have been validated to support its execution.

1.3 Installation

1.3.1 Requirements

Requires Python ≥ 3.7

1.3.2 Quick start

```
pip install modeci-mdf
```

For more detailed installation instructions see [here](#).

For guidelines on contributing to the development of MDF, see [here](#).

1.4 Examples

To get started, follow the simple example in a Jupyter notebook [here](#)

Multiple examples of serialized MDF files, the Python scripts used to generate them, as well as mappings to target environments can be found [here](#).

INSTALLATION

2.1 Requirements

Python (≥ 3.7)

2.2 Quick start

Use pip to install the latest version of MDF (plus dependencies) from [PyPI](#):

```
pip install modeci_mdf
```

2.3 Installation from source

To install the MDF package from [source](#) and run it locally:

2.3.1 1) Create a virtual environment (e.g. called `mdf-env`)

```
pip install virtualenv
virtualenv mdf-env
```

2.3.2 2) Activate the virtual environment

```
source mdf-env/bin/activate
```

2.3.3 3) Clone this repository

```
git clone https://github.com/ModECI/MDF.git
```

2.3.4 4) Change to the directory

```
cd MDF
```

2.3.5 5) Install the package

```
pip install .
```

Alternatively, to install MDF plus all of the modules required for the export/import interfaces (e.g. PsyNeuLink, NeuroML):

```
pip install .[all]
```

2.4 Additional dependencies

To generate generate Graph images in MDF you require Graphviz which uses dot.

```
pip install graphviz
```

To render the generated DOT source code, you also need to install [Graphviz](#) ([download page](#), [installation procedure for Windows](#)).

Make sure that the directory containing the dot executable is on your system's PATH (sometimes done by the installer; setting PATH on [Linux](#), [Mac](#), and [Windows](#)).

2.5 Generating ModECI MDF documentation offline

The ModECI MDF Documentation can be found online [here](#). If you are working on MDF documentation or you make changes to the documentation, it is good practice to see if it is working as expected before pushing to the GitHub repository. Here is a walkthrough on how to generate the ModECI MDF documentation offline

2.6 Requirements

Python version-3.10 is ideally used for generating MDF documentation offline but if not working, use python version-3.9. The steps are the same except in creating a virtual environment.

The documentation is generated using [Sphinx](#). Make is also required. For **Windows** installation of Make, see [here](#). For **Mac** installation of Make, see [here](#)

2.6.1 1) Create a virtual environment with python

```
# install virtual environment

pip install virtualenv

# create & activate virtual environment for python 3.9

python3.9 -m virtualenv venv39
venv39\Scripts\activate

# or create & activate virtual environment for python 3.10

python3.10 -m virtualenv venv310
venv310\Scripts\activate
```

2.6.2 2) Clone MDF repository from GitHub into your local machine

```
git clone https://github.com/ModECI/MDF.git
```

2.6.3 3) Change into the MDF directory

```
cd MDF
```

2.6.4 4) Install all MDF package into the virtual environment

```
pip install .[all]
```

2.6.5 5) Change directory into sphinx folder

```
# for windows
cd docs\sphinx

# for Mac/Linux
cd docs/sphinx
```

2.6.6 6) Create offline documentation in sphinx folder

```
# To allow a fresh start when making the documentation
make clean

# To make the documentation
make html
```

2.6.7 7) Change directory into html folder and run the documentation offline

```
# for Windows go into build\html folder and double click on the index.html file, or:  
  
cd build\html  
index.html  
  
# for Mac, go into build/html folder and double click on the index.html file or:  
cd build/html  
open index.html
```

The documentation will open up in your browser automatically or right click on the file and open in any browser of your choice.

MODECI CONTRIBUTORS

This page lists names and Github profiles of contributors to the various ModECI repositories, listed in no particular order. This page is generated periodically, most recently on 2023-03-06.

- Padraig Gleeson (@pgleeson)
- David Turner (@davidt0x)
- Katherine Mantel (@kmantel)
- Ivy (@Ivy8127)
- (@mraunak)
- Shanka Subhra Mondal (@Shanka123)
- Onabajo Monsurat (@Monsurat-Onabajo)
- Parikshit Singh Rathore (@parikshit14)
- Patrick Stock (@patrickstock)
- Jeremy Lee (@jeremyr17)
- Raghavendra Pradyumna Pothukuchi (@rpradyumna)
- Marble Kusanele Mpofu (@kusanele)
- Somya Agrawal (@somyagr)
- (@jdcgni)
- Riya Saxena (@29riyasaxena)
- (@FatimaArshad-DS)
- Megha Bose (@Megha-Bose)
- Pranav Gokhale (@singular-value)
- (@sakshikaushik717)
- Esraa Abdelmaksoud (@esraa-abdelmaksoud)
- Shivani Rana (@shivani6320)
- (@vidhya-metacell)
- (@nicholwkprinceton)
- Matteo Cantarelli (@tarelli)
- Brian Broll (@brollb)

3.1 Repositories

- [MDF](#)
- [Website](#)
- [MDFTests](#)
- [modelspec](#)
- [PsyNeuLink](#)

SPECIFICATION OF MODECI V0.4

Note: the ModECI MDF specification is still in development! See [here](#) for ongoing discussions.

MODEL

The top level construct in MDF is Model, which may contain multiple *Graph* objects and model attribute(s)

Allowed parameters

Allowed field	Data Type	Description
metadata	Union[Any, NoneType]	Optional metadata field, an arbitrary dictionary of string keys and JSON serializable values.
id	str	A unique identifier for this Model
format	str	Information on the version of MDF used in this file
generat- ing_application	str	Information on what application generated/saved this file
onnx_opset_version	Union[str, None-Type]	The ONNX opset used for any ONNX functions in this model.

Allowed children

Allowed child	Data Type	Description
graphs	<i>Graph</i>	The collection of graphs that make up the MDF model.

GRAPH

A directed graph consisting of *Nodes* (with *Parameters* and *Functions* evaluated internally) connected via *Edges*.

Allowed parameters

Allowed field	Data Type	Description
metadata	Union[Any, NoneType]	Optional metadata field, an arbitrary dictionary of string keys and JSON serializable values.
id	str	A unique identifier for this Graph
parameters	Union[Any, NoneType]	Dictionary of global parameters for the Graph
conditions	Union[ConditionSet, NoneType]	The ConditionSet stored as dictionary for scheduling of the Graph

Allowed children

Allowed child	Data Type	Description
nodes	<i>Node</i>	One or more <i>Node(s)</i> present in the graph
edges	<i>Edge</i>	Zero or more <i>Edge(s)</i> present in the graph

NODE

A self contained unit of evaluation receiving input from other nodes on *InputPort(s)*. The values from these are processed via a number of *Function(s)* and one or more final values are calculated on the *OutputPort(s)*

Allowed parameters

Allowed field	Data Type	Description
metadata	Union[Any, None-Type]	Optional metadata field, an arbitrary dictionary of string keys and JSON serializable values.
id	str	A unique identifier for the node.

Allowed children

Allowed child	Data Type	Description
input_ports	<i>InputPort</i>	Dictionary of the <i>InputPort</i> objects in the Node
functions	<i>Function</i>	The <i>Function(s)</i> for computation the node
parameters	<i>Parameter</i>	Dictionary of <i>Parameter(s)</i> for the node
output_ports	<i>OutputPort</i>	The <i>OutputPort(s)</i> containing evaluated quantities from the node

INPUTPORT

The *InputPort* is an attribute of a Node which allows external information to be input to the Node

Allowed parameters

Allowed field	Data Type	Description
meta-data	Union[Any, NoneType]	Optional metadata field, an arbitrary dictionary of string keys and JSON serializable values.
id	str	The unique (for this Node) id of the input port,
shape	Union[Tuple[int, ...], NoneType]	The shape of the input port. This uses the same syntax as numpy ndarray shapes (e.g., numpy.zeros(shape) would produce an array with the correct shape
type	Union[str, None-Type]	The data type of the input received at a port.

FUNCTION

A single value which is evaluated as a function of values on *InputPort(s)* and other Functions

Allowed parameters

Al- lowed field	Data Type	Description
meta- data	Union[Any, NoneType]	Optional metadata field, an arbitrary dictionary of string keys and JSON serializable values.
id	str	The unique (for this Node) id of the function, which will be used in other <i>Functions</i> and the <i>OutputPorts</i> for its value
func- tion	Union[str, NoneType]	Which of the in-built MDF functions (linear , etc.). See supported functions: https://mdf.readthedocs.io/en/latest/api/MDF_function_specifications.html
args	Union[Any, NoneType]	Dictionary of values for each of the arguments for the Function, e.g. if the in-built function is linear(slope), the args here could be {"slope":3} or {"slope": "input_port_0 + 2"}
value	Union[EvaluableExpression, List, Dict, ndarray, int, float, str, NoneType]	If the function is a value expression, this attribute will contain the expression and the function and args attributes will be None.

PARAMETER

A parameter of the *Node*, which can be: 1) a specific fixed **value** (a constant (int/float) or an array) 2) a string expression for the **value** referencing other named *Parameter(s)*. which may be stateful (i.e. can change value over multiple executions of the *Node*); 3) be evaluated by an inbuilt **function** with **args**; 4) or change from a **default_initial_value** with a **time_derivative**.

Allowed parameters

Al- lowed field	Data Type	Description
meta- data	Union[Any, NoneType]	Optional metadata field, an arbitrary dictionary of string keys and JSON serializable values.
id	str	
value	Union[EvaluableExpression, List, Dict, ndarray, int, float, str, NoneType]	The next value of the parameter, in terms of the inputs, functions and PREVIOUS parameter values
de- fault_initial_value	Union[EvaluableExpression, List, Dict, ndarray, int, float, str, NoneType]	The initial value of the parameter, only used when parameter is stateful.
time_derivative	Union[str, NoneType]	How the parameter changes with time, i.e. ds/dt. Units of time are seconds.
func- tion	Union[str, NoneType]	Which of the in-build MDF functions (linear etc.) this uses, See
args	Union[Any, NoneType]	Dictionary of values for each of the arguments for the function of the parameter, e.g. if the in-build function is linear(slope) , the args here could be {"slope": 3} or {"slope": "input_port_0 + 2"}

Allowed children

Allowed child	Data Type	Description
conditions	<i>ParameterCondition</i>	Parameter specific conditions

PARAMETERCONDITION

A condition to test on a Node's parameters, which if true, sets the value of this Parameter

Allowed parameters

Allowed field	Data Type	Description
id	str	A unique identifier for the Parameter-Condition
test	Union[EvaluableExpression, List, Dict, ndarray, int, float, str, NoneType]	The boolean expression to evaluate
value	Union[EvaluableExpression, List, Dict, ndarray, int, float, str, NoneType]	The new value of the Parameter if the test is true

OUTPUTPORT

The *OutputPort* is an attribute of a *Node* which exports information to another *Node* connected by an *Edge*

Allowed parameters

Al- lowed field	Data Type	Description
meta- data	Union[Any, NoneType]	Optional metadata field, an arbitrary dictionary of string keys and JSON serializable values.
id	str	Unique identifier for the output port.
value	Union[str, None-Type]	The value of the <i>OutputPort</i> in terms of the <i>InputPort</i> , <i>Function</i> values, and <i>Parameter</i> values.
shape	Union[Tuple[int, ...], NoneType]	The shape of the output port. This uses the same syntax as numpy ndarray shapes (e.g., numpy.zeros(shape) would produce an array with the correct shape
type	Union[str, None-Type]	The data type of the output sent by a port.

EDGE

An *Edge* is an attribute of a *Graph* that transmits computational results from a sender's *OutputPort* to a receiver's *InputPort*.

Allowed parameters

Allowed field	Data Type	Description
metadata	Union[Any, None-Type]	Optional metadata field, an arbitrary dictionary of string keys and JSON serializable values.
id	str	A unique string identifier for this edge.
sender	str	The id of the <i>Node</i> which is the source of the edge.
receiver	str	The id of the <i>Node</i> which is the target of the edge.
sender_port	str	The id of the <i>OutputPort</i> on the sender <i>Node</i> , whose value should be sent to the receiver_port
receiver_port	str	The id of the <i>InputPort</i> on the receiver <i>Node</i>
parameters	Union[Any, None-Type]	Dictionary of parameters for the edge.

CONDITION

A set of descriptors which specifies conditional execution of Nodes to meet complex execution requirements.

Allowed parameters

Allowed field	Data Type	Description
metadata	Union[Any, None-Type]	Optional metadata field, an arbitrary dictionary of string keys and JSON serializable values.
type	str	The type of <i>Condition</i> from the library
kwargs	Union[Any, None-Type]	The dictionary of keyword arguments needed to evaluate the <i>Condition</i>

CONDITIONSET

Specifies the non-default pattern of execution of Nodes

Allowed parameters

Allowed field	Data Type	Description
metadata	Union[Any, None-Type]	Optional metadata field, an arbitrary dictionary of string keys and JSON serializable values.
node_specific	Union[Condition, NoneType]	A dictionary mapping nodes to any non-default run conditions
termination	Union[Condition, NoneType]	A dictionary mapping time scales of model execution to conditions indicating when they end

MDF EXAMPLES

Examples of [Python](#), [JSON](#) and [YAML](#) files to illustrate the structure and usage of MDF.

[Simple](#) | [ABCD](#) | [Arrays](#) | [States](#) | [Conditions](#) | [Parameters and Functions](#)

16.1 Simple example

[Python Source](#) | [JSON](#) | [YAML](#)

A simple example with 2 [Nodes](#) connected by an [Edge](#):

With more detail on [Nodes](#) (showing [Input Ports](#) (green), [Parameters](#) (blue) and [Output Ports](#) (red) and [Edges](#):

16.2 ABCD

[Python Source](#) | [JSON](#) | [YAML](#)

Another simple example with more [Nodes](#).

16.3 Arrays

[Python Source](#) | [JSON](#) | [YAML](#)

An example using arrays for [Parameters](#) and weights on [Edges](#).

16.4 States

[Python Source](#) | [JSON](#) | [YAML](#)

An example with [Nodes](#) containing persistent [States](#).

16.5 Conditions

[Python Source](#) | [JSON](#) | [YAML](#)

A simple 3 [Nodes](#) graph with scheduling [Conditions](#). For more examples of conditions see [here](#).

16.6 Parameters and Functions

[Python Source](#) | [JSON](#) | [YAML](#)

A simple [Node](#) with a number of different types of [Parameters](#) (in blue; fixed and **stateful**) and [Functions](#) (in purple; can be built in or ONNX based).

16.7 More examples

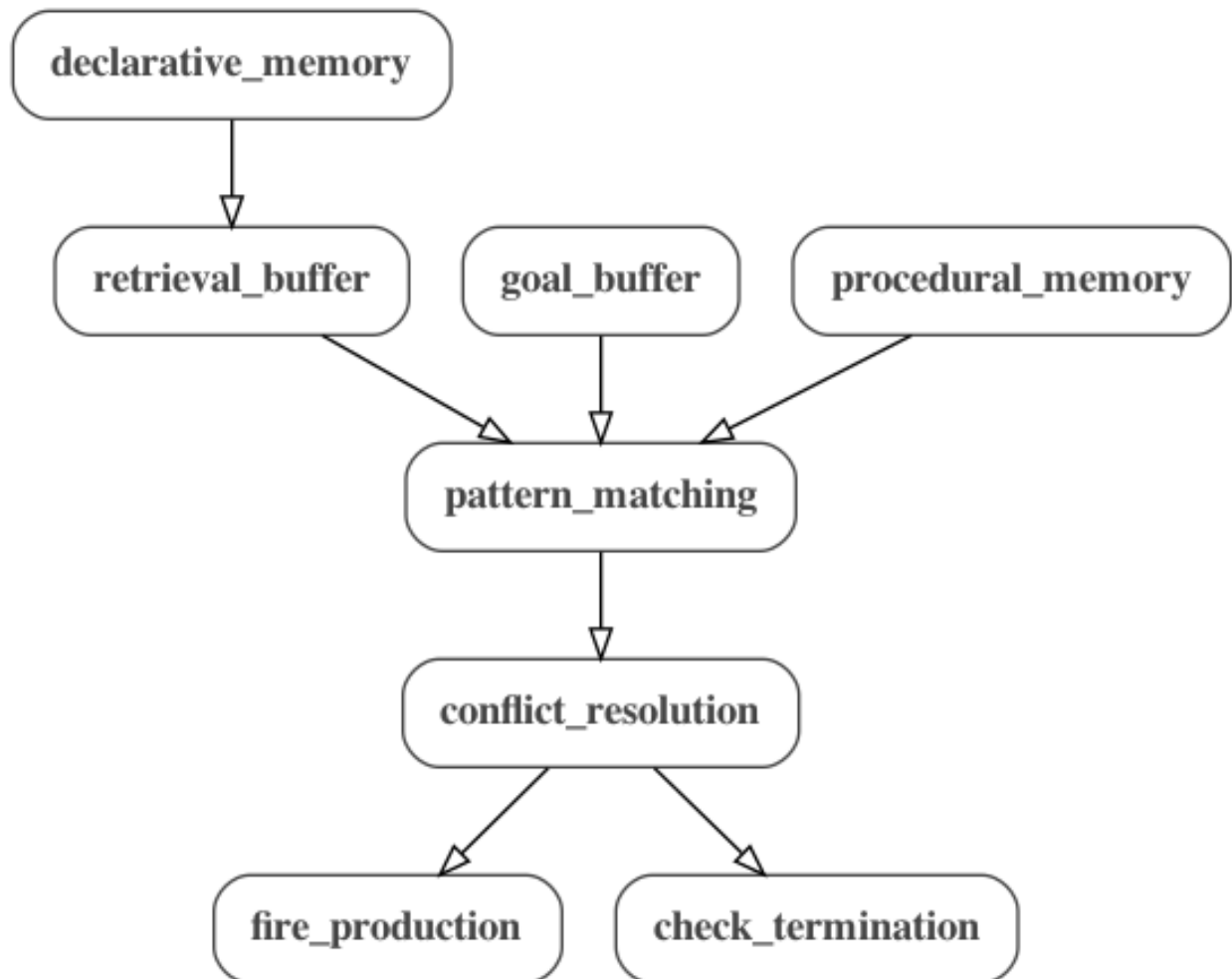
There are further examples under development, including of a Recurrent Neural Network (RNN), and an Integrate and Fire (IaF) neuron model in [this directory](#).

INTERACTIONS BETWEEN MDF AND ACT-R

This directory contains examples of ACT-R models converted to MDF. The ACT-R models `count.lisp` and `addition.lisp` are based on the [ACT-R tutorial](#).

The scripts `count.py` and `addition.py` can be run to create the MDF `.json` and `.yaml` files for the given example and execute it using the MDF scheduler.

The below graph represents the basic structure of all ACT-R models in MDF:



There are also more detailed graphs `count.png` and `addition.png` for each example.

17.1 Count Model

ACT-R | JSON | YAML | Python Script | Graph



The count model counts from a start value to an end value. The script `count.py` first reads the original ACT-R model file `count.lisp`, generates an MDF representation using the MDF ACT-R interface, and outputs the JSON and YAML files. It then executes the MDF model using the MDF scheduler and outputs the final goal set by the model once execution is finished. The final goal has the form:

```
{'name': 'first-goal', 'ISA': 'count-from', 'start': 'two', 'end': 'four',
'count': 'four'}
```

In this example, the start value is two, the end value is four, and the final value of count is four, indicating that the model counted from two to four. The start and end values can be modified in line 17 of `count.lisp`, which sets the initial goal of the model:

```
(first-goal ISA count-from start two end four)
```

The script can use any values specified in `count.lisp`, so the model can be modified and run multiple times in order to test different values. The count graph represents this example.

17.2 Addition Model

ACT-R | JSON | YAML | Python Script | Graph



The addition model computes the sum of two numbers. The script `addition.py` functions identically to the previous example, but uses the addition model instead. The final goal has the form:

```
{'name': 'second-goal', 'ISA': 'add', 'arg1': 'five', 'arg2': 'two', 'sum':
'seven', 'count': 'nil'}
```

In this case, the first argument is five, the second argument is two, and the model has calculated the sum, seven. Like the previous example, the arguments can be modified in line 23 of `addition.lisp` in order to test different values:

```
(second-goal ISA add arg1 five arg2 two)
```

The addition graph represents this example.

INTERACTIONS BETWEEN NEUROML AND MDF

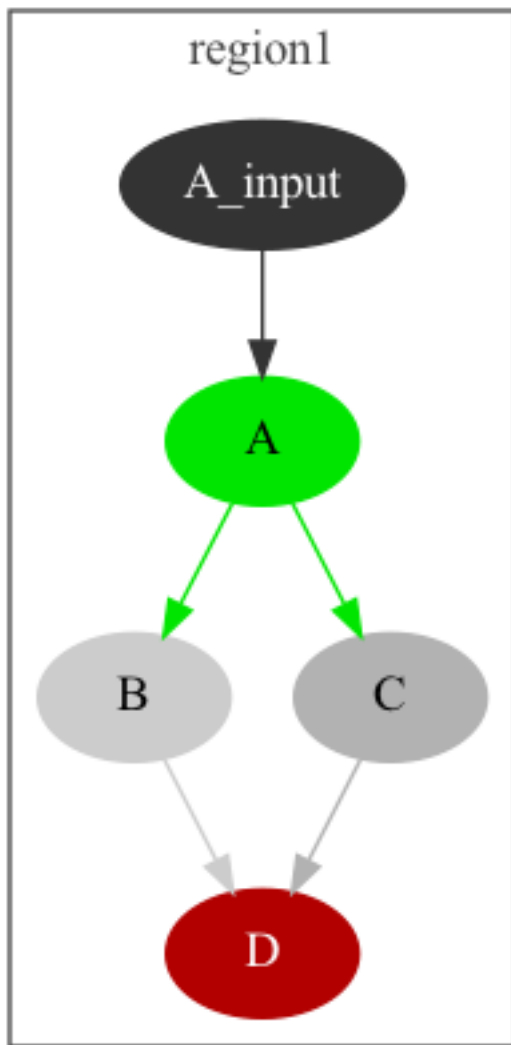
18.1 1) Converting NeuroML to MDF

18.2 1.1) Simple ABCD model

Summary: A model is created in NeuroML (using cell dynamics specified in LEMS and a network in NeuroMLlite) and converted to the equivalent model in MDF, which runs with the reference MDF execution engine.

18.2.1 1.1.1) ABCD - NeuroMLlite version

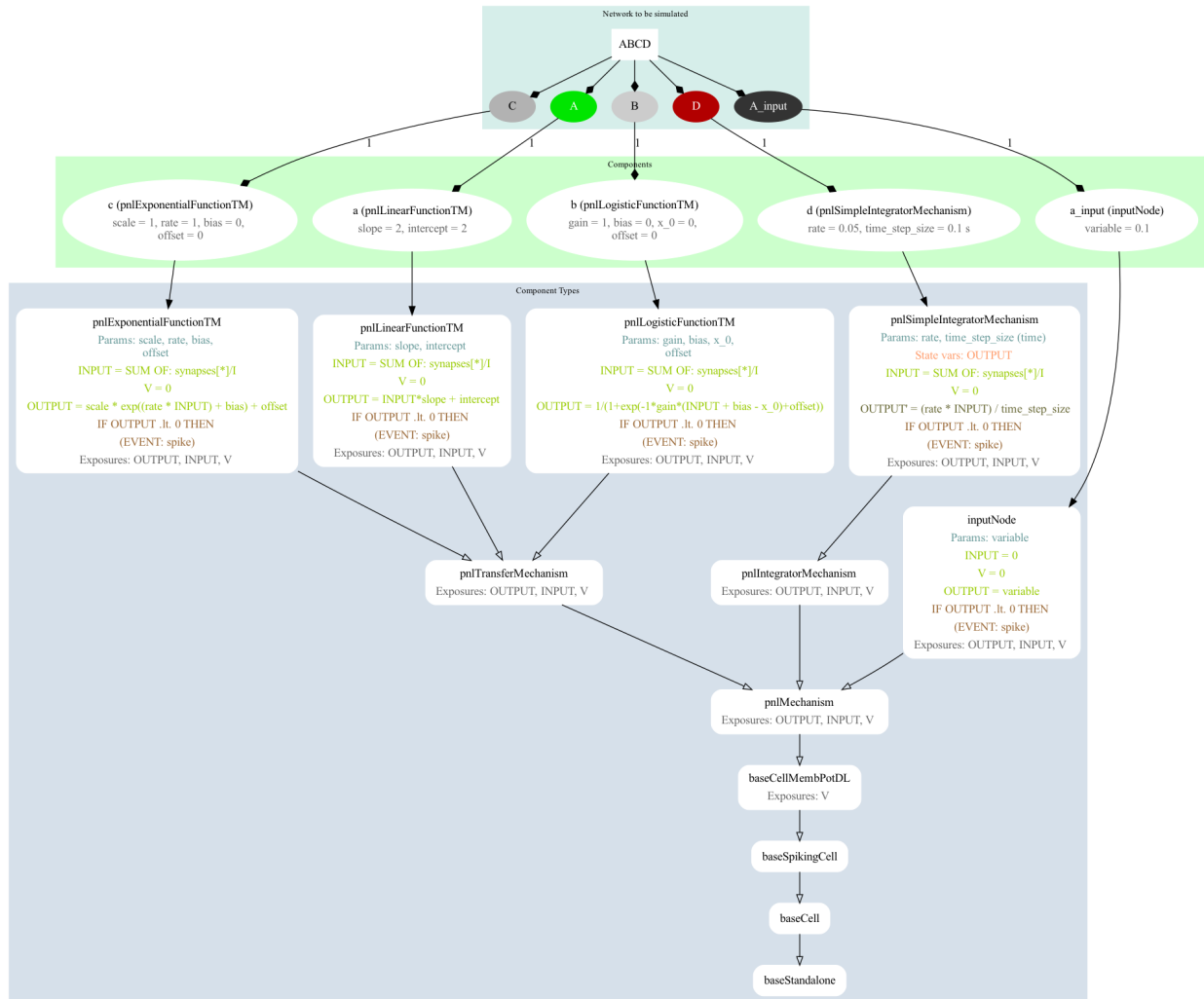
ABCD.py is a script using the [NeuroMLlite](#) package to create a simple network with 4 connected elements. The network built can be seen below (this can be generated with `python ABCD.py -graph2`):



18.2.2 1.1.2) ABCD - NeuroML2 version

A version of the network in NeuroML 2 can be generated with `python ABCD.py -nml`, or generated and executed with jNeuroML with `python ABCD.py -nml`. This will produce the NeuroML file: `ABCD.net.nml` (note though this is not valid, as not all the elements included are pure NeuroML). A [LEMS Simulation file](#) is generated for running the model in jNeuroML or pyNeuroML: `LEMS_SimABCD.xml`

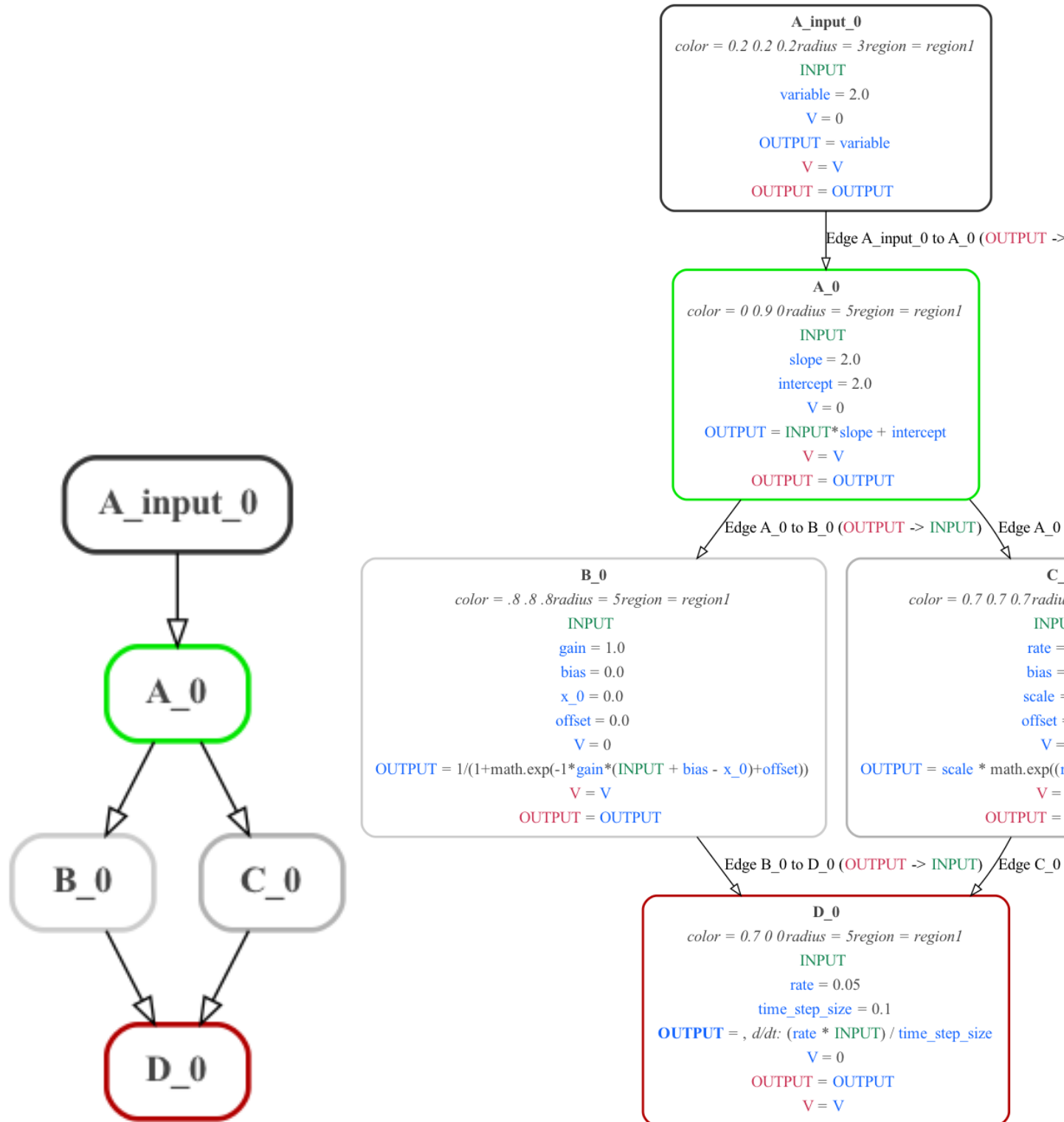
The definitions of the components used for A, B, etc. can be found in `PNL.xml`. This is a set of definitions of component types based on those present in PsyNeuLink. A graph depicting the definitions of the network elements can be generated with `pynml LEMS_SimABCD.xml -lems-graph`:



18.2.3 1.1.3) ABCD - MDF version

A version of the network in MDF can be generated from NeuroMLlite definition with: `python ABCD.py -mdf` producing `ABCD.mdf.yaml` and `ABCD.mdf.json`.

A graph of the structure of the MDF model can be generated with: `python -m modeci_mdf.interfaces.graphviz.exporter ABCD.mdf.yaml 1` (left below), or with more detail: `python -m modeci_mdf.interfaces.graphviz.exporter ABCD.mdf.yaml 3` (right below.)

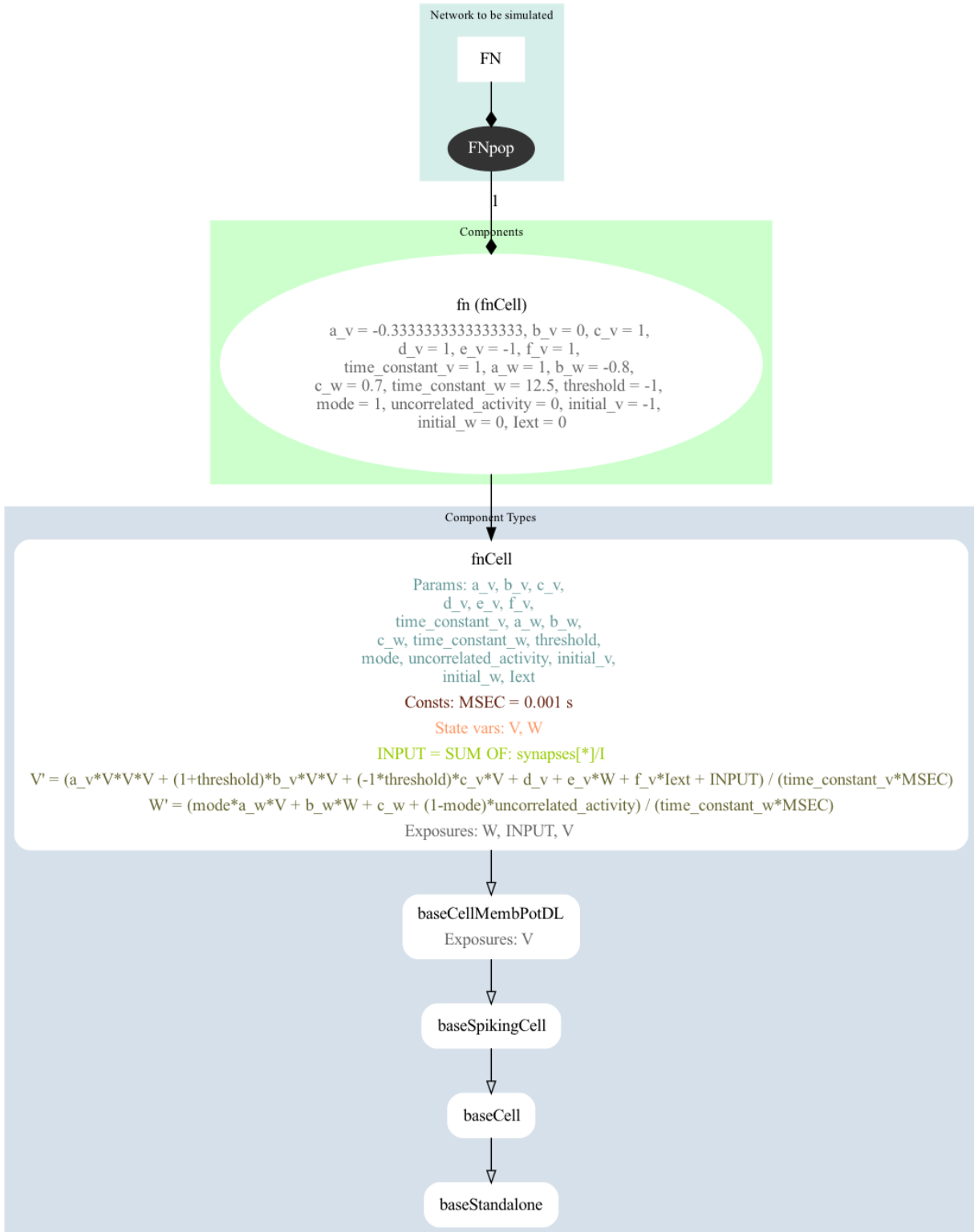


18.3 1.2) FitzHugh Nagumo cell models

18.3.1 1.2.1) FN - NeuroML version

A version of the FitzHugh Nagumo neuron model has been created using NeuroMLlite (FN.py) which generated LEMS (LEMS_SimFN.xml) which can simulate the NeuroML model (FN.net.nml).

A graphical representation of the LEMS is below:



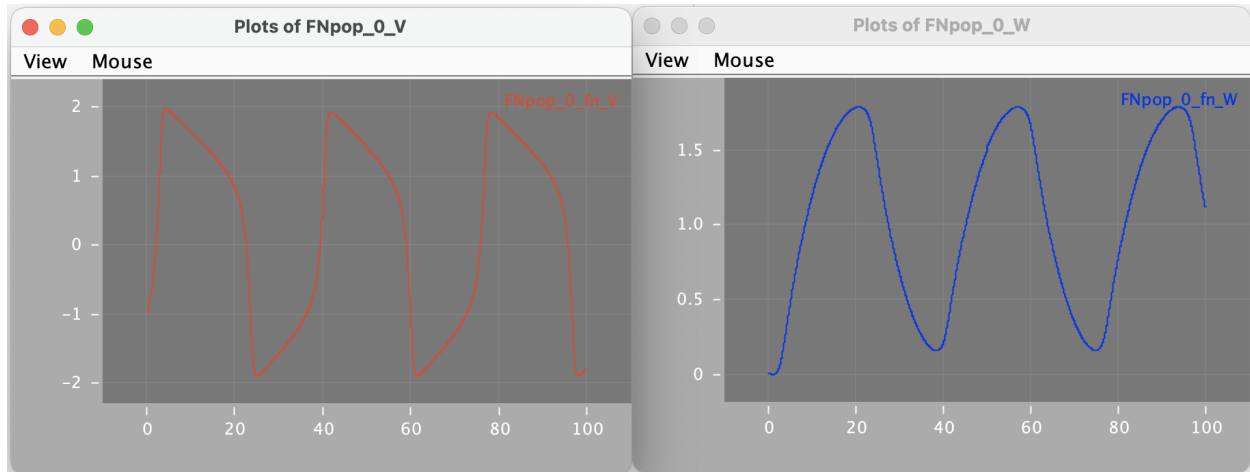
It can be run with:

```
python FN.py -jnm1 # Generate and run the LEMS file from the NeuroMLlite_
description
```

(continues on next page)

(continued from previous page)

```
pynml LEMS_SimFN.xml # Run the LEMS file using pyNeuroML
```



18.3.2 1.2.2) FN - MDF version

The NeuroMLlite version can also be used to generate MDF for the model:

```
python FN.py -mdf # Generate the MDF serializations (JSON and YAML) from the_
↪ NeuroMLlite description
```

These can be seen here: FN.mdf.json, FN.mdf.yaml, and a graphical version generated with:

```
python -m modeci_mdf.interfaces.graphviz.importer FN.mdf.yaml 3 # Generate graph_
↪ from MDF version
```

```

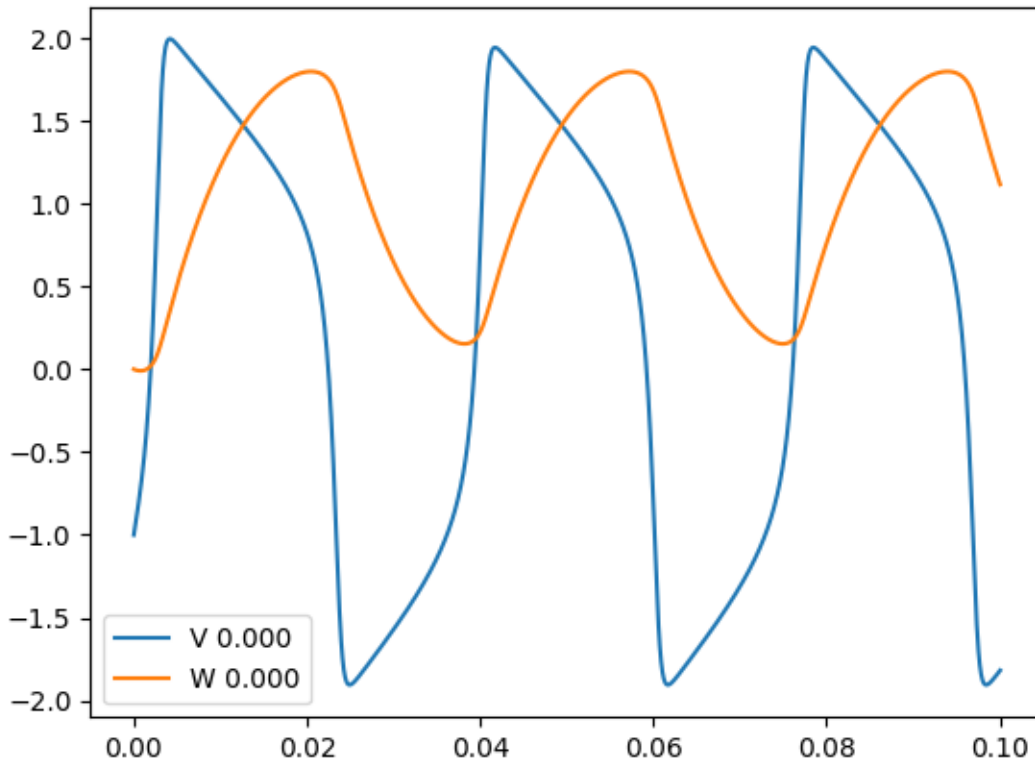
FNpop_0
color = 0.2 0.2 0.2 radius = 3 region = region1
INPUT
initial_w = 0.0
initial_v = -1.0
a_v = -0.3333333333333333
b_v = 0.0
c_v = 1.0
d_v = 1.0
e_v = -1.0
f_v = 1.0
time_constant_v = 1.0
a_w = 1.0
b_w = -0.8
c_w = 0.7
time_constant_w = 12.5
threshold = -1.0
mode = 1.0
uncorrelated_activity = 0.0
Iext = 0.0
MSEC = 0.001
V = def init value: initial_v, d/dt: (a_v*V*V*V + (1+threshold)*b_v*V*V + (-1*threshold)*c_v*V + d_v + e_v*W + f_v*Iext + INPUT) / (time_constant_v*MSEC)
W = def init value: initial_w, d/dt: (mode*a_w*V + b_w*W + c_w + (1-mode)*uncorrelated_activity) / (time_constant_w*MSEC)
V = V
W = W

```

18.3.3 1.2.3) FN - Execute model using MDF

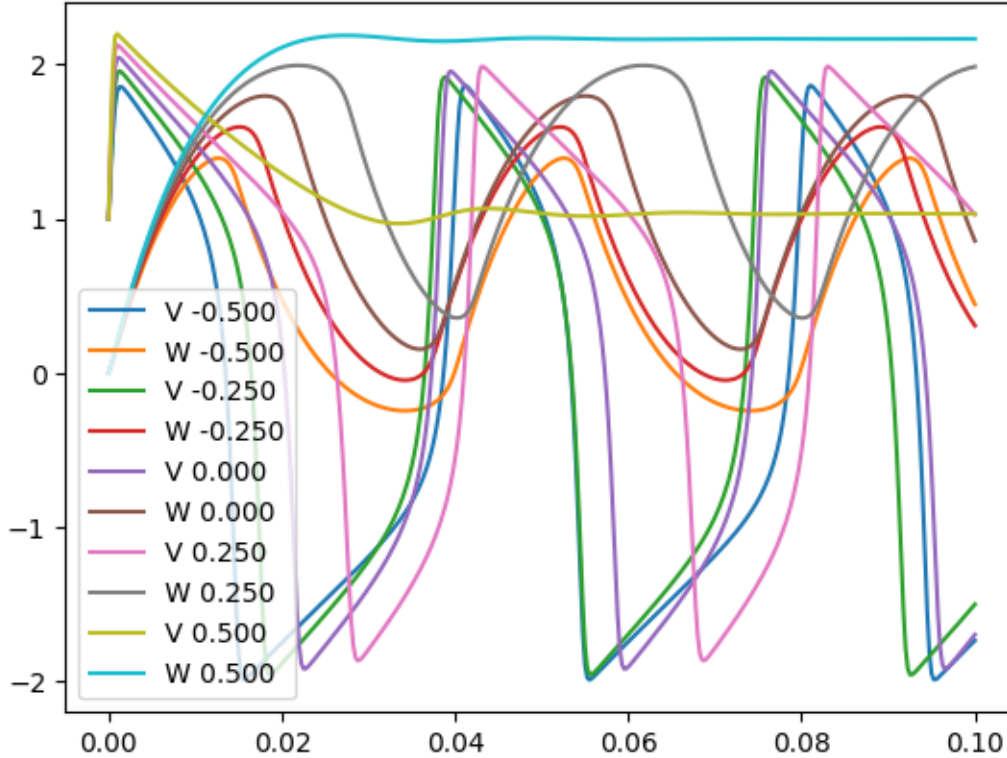
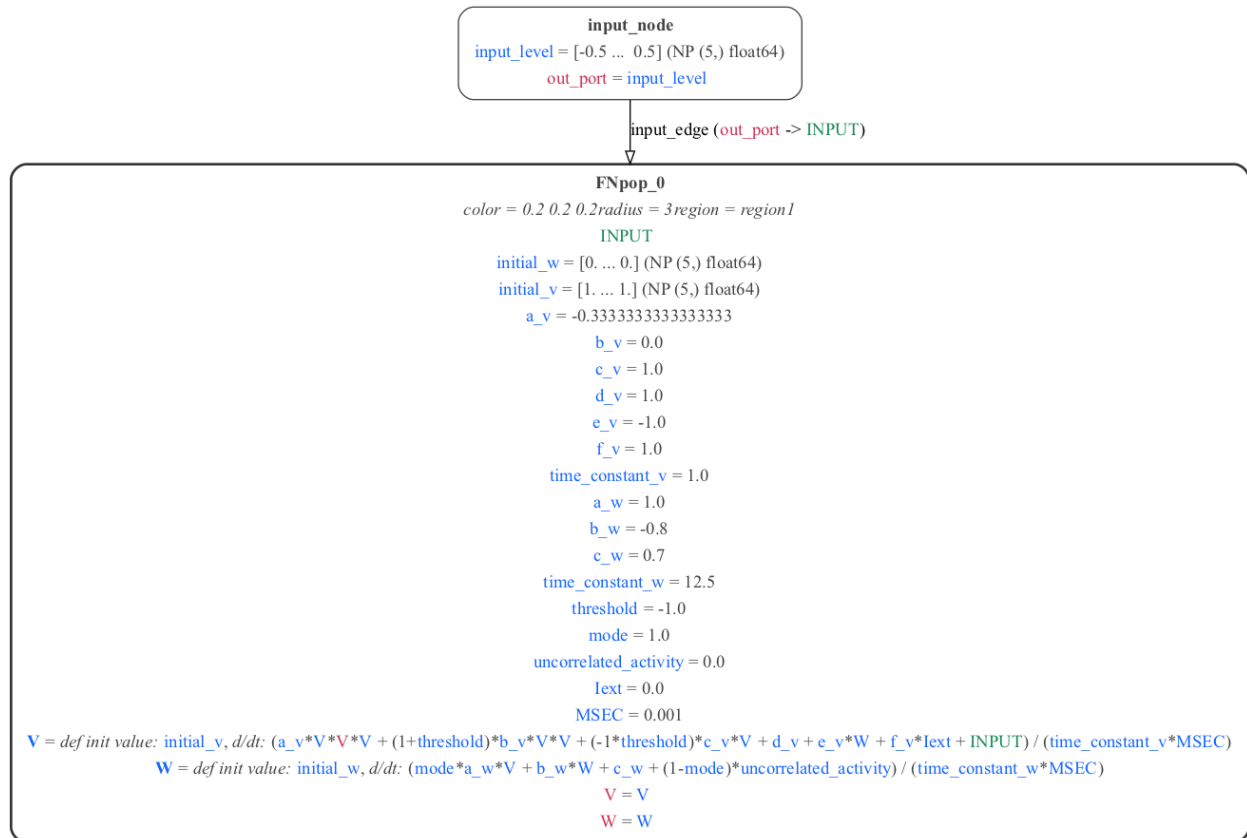
A script has been created (FNrun.py) where the model is loaded, run using the standard MDF execution engine, and plotted:

```
python FNrun.py      # Load FN model and run with MDF scheduler
```



Adding the option `-multi` to the Python script for running the FN example, modifies the graph to add an input node with an array of values, meaning multiple instances of the FN neuron will be simulated:

```
python FN.py -multi
```



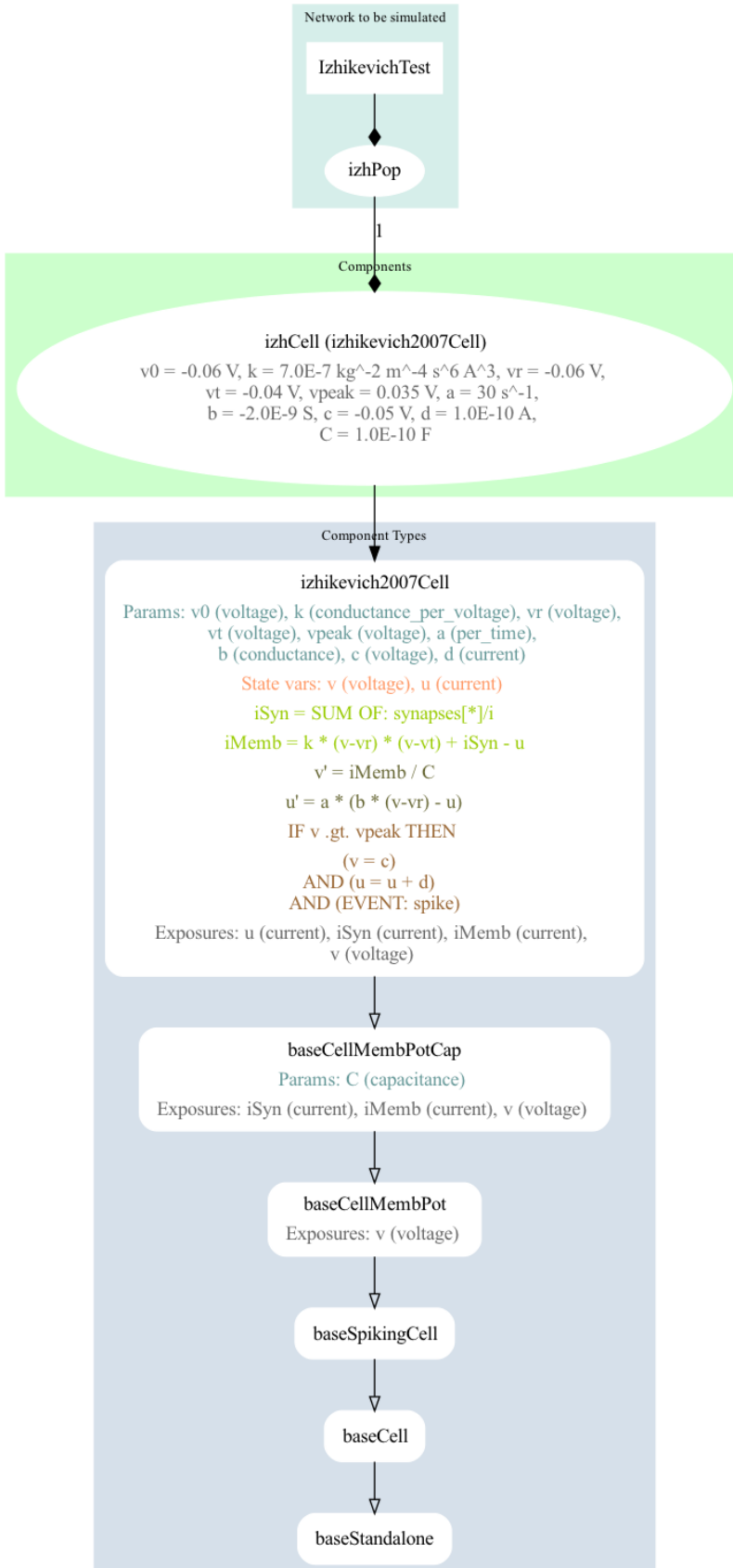
18.4 1.3) Izhikevich cell models

A version of the Izhikevich spiking neuron model has been created in NeuroML and can be exported to MDF and executed with the standard execution engine.

18.4.1 1.3.1) Izhikevich - NeuroML version

The single cell model has been created using NeuroMLlite (Izhikevich.py) which generated LEMS (LEMS_SimIzhikevichTest.xml) which can simulate the NeuroML model (IzhikevichTest.net.nml).

A graphical representation of the LEMS is below:



It can be run with:

```
python Izhikevich.py -jnml          # Generate and run the LEMS file from the NeuroMLlite_
↪description
pynml LEMS_SimIzhikevichTest.xml    # Run the LEMS file using pyNeuroML
```

18.4.2 1.3.2) Izhikevich - MDF version

The NeuroMLlite version can also be used to generate MDF for the model:

```
python Izhikevich.py -mdf          # Generate the MDF serializations (JSON and YAML) from_
↪the NeuroMLlite description
```

These can be seen here: IzhikevichTest.mdf.json, IzhikevichTest.mdf.yaml, and a graphical version generated with:

```
python -m modeci_mdf.interfaces.graphviz.importer IzhikevichTest.mdf.yaml 3      # 
↪Generate graph from MDF version
```



```

    izePop_0
    synapses_i
    v0 = -0.06
    C = 1e-10
    k = 7e-07
    vr = -0.06
    vt = -0.04
    vpeak = 0.035
    a = 30.0
    b = -2e-09
    c = -0.05
    d = 1e-10

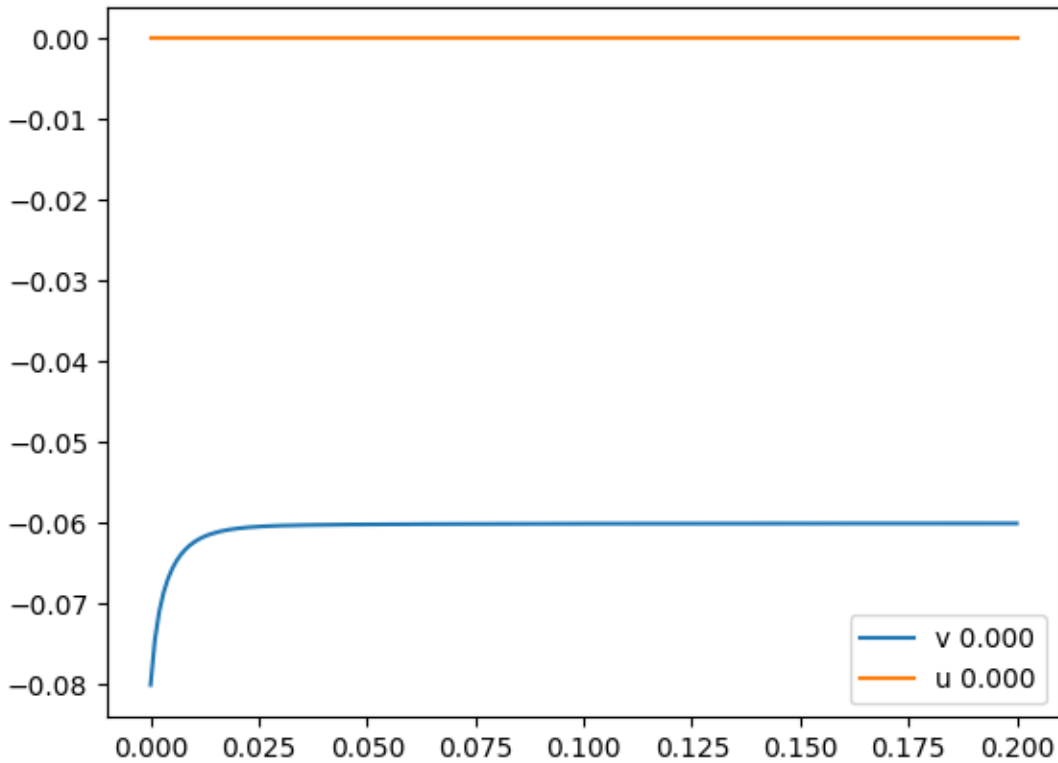
    v = def init value: v0, d/dt: iMemb / C
    condition_0: IF v > vpeak THEN v=c
    u = def init value: 0, d/dt: a * (b * (v-vr) - u)
    condition_0: IF v > vpeak THEN u=u + d
    iSyn = synapses_i
    iMemb = k * (v-vr) * (v-vt) + iSyn - u
    v = v
    u = u
    iMemb = iMemb

```

18.4.3 1.3.3) Izhikevich - Execute model using MDF

A script has been created (Izh_run.py) where the model is loaded, run using the standard MDF execution engine, and plotted:

```
python Izh_run.py      # Load Izh model and run with MDF scheduler
```



18.5 2) Converting MDF to NeuroML/LEMS

It is also possible to convert MDF models into equivalents in NeuroML/LEMS:

```
cd ../MDF # convert some of the examples in the examples/MDF directory

python -m modeci_mdf.interfaces.neuroml.exporter Simple.json -run
python -m modeci_mdf.interfaces.neuroml.exporter ABCD.json -run
python -m modeci_mdf.interfaces.neuroml.exporter States.json -run
```

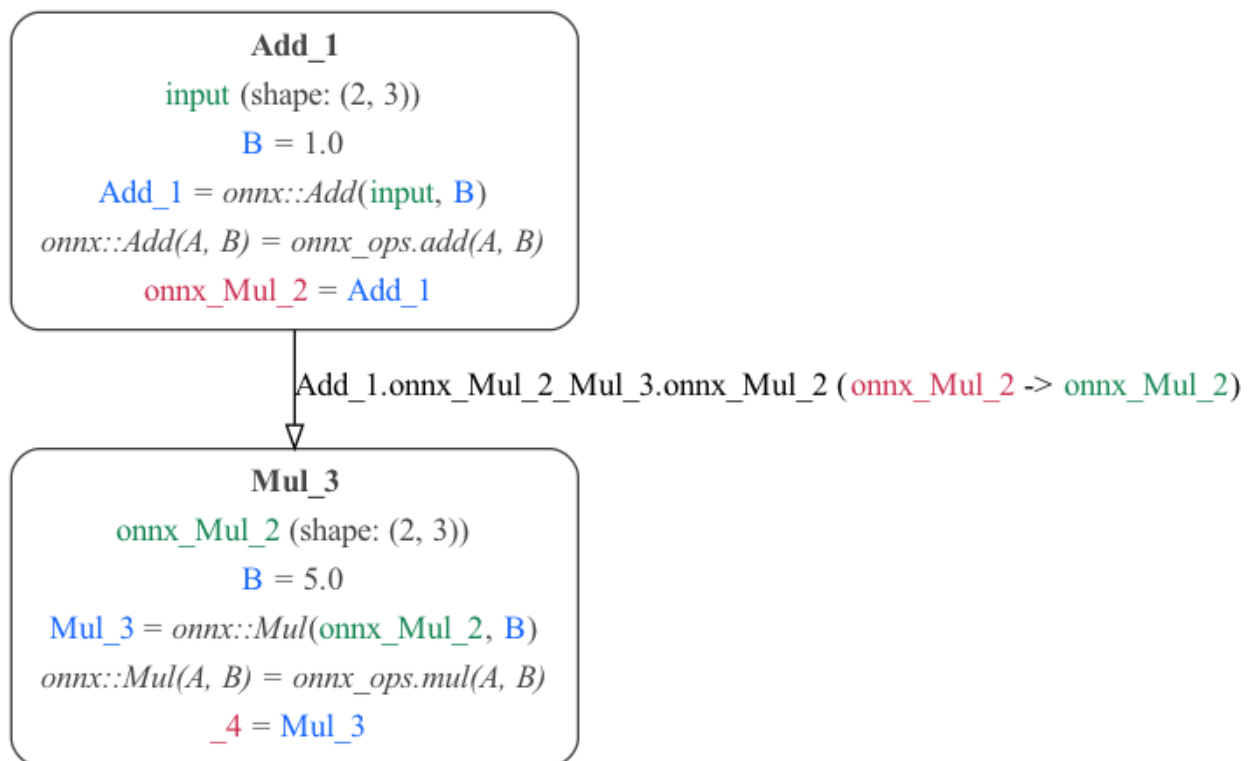
ONNX MDF CONVERTER

19.1 ONNX to MDF

19.1.1 AB Sequential Model - 2 nodes

Python source | JSON | YAML

This is an example of a PyTorch model with 2 nodes. First, the script saves the PyTorch model as ONNX and then converts this to MDF. The graphical view of the generated MDF is shown below.



19.1.2 ABC Sequential Model with Loop

Python source | JSON | YAML

Note: Example still in development!

This is an example of a PyTorch model that is implemented in `onnx_mdf/examples/simple_abc.py`. The model code is very simple:

```
import torch

class A(torch.nn.Module):
    def forward(self, x):
        return x + 1

@torch.jit.script
def loop_b(x, y):
    for i in range(int(y)):
        x = x / 10
    return x

class B(torch.nn.Module):
    def forward(self, x, y):
        return loop_b(x, y)

class C(torch.nn.Module):
    def forward(self, x):
        return x * 100

class ABC(torch.nn.Module):
    def __init__(self):
        super(ABC, self).__init__()
        self.A = A()
        self.B = B()
        self.C = C()

    def forward(self, x, B_loop_count):
        return self.C(self.B(self.A(x), B_loop_count))
```

This implements a PyTorch model with three modules. The modules process the input sequentially, and the inner B module has a loop construct.



It is exported to ONNX via a combination of tracing and scripting.

19.1.3 ABCD Branching Conditional Model

Python source | JSON | YAML

Note: Example still in development!

This is an example of a PyTorch model that have four components (A, B, C, D). We loop over the whole model 10 iterations. A is executed only on the first iteration, B is executed every iteration, C is executed every 5 times B is executed, and D is executed every 10 times B is executed. A, B, C, and D are each simple stateless linear functions. This type of conditional execution specification is common in PsyNeuLink. The PyTorch code for the model is fairly straightforward:

```
class Linear(torch.nn.Module):
    def __init__(self, slope=1.0, intercept=0.0):
        super(Linear, self).__init__()
        self.slope = slope
        self.intercept = intercept

    def forward(self, x):
        return self.slope*x + self.intercept

class ABCD(torch.nn.Module):
    def __init__(self, A, B, C, D):
        super(ABCD, self).__init__()
        self.A = A
        self.B = B
        self.C = C
        self.D = D

    def forward(self, x):

        # Since we are implementing conditions that reference the number of calls
        # to A and B, we need to keep track of this.
        num_A_calls = 0
        num_B_calls = 0

        # We need to initialize outputs, torchscript jit complains if c and d
        # are not defined in the FALSE branches of our conditionals.
        a = torch.zeros_like(x)
        b = torch.zeros_like(x)
        c = torch.zeros_like(x)
        d = torch.zeros_like(x)

        for i in range(10):

            # A: pnl.AtNCalls(A, 0),
            if num_A_calls == 0:
                a = self.A(x)
                num_A_calls = num_A_calls + 1

            # B: pnl.Always()
            b = self.B(a)
            num_B_calls = num_B_calls + 1

            # C: pnl.EveryNCalls(B, 5),
            if num_B_calls % 5 == 0:
                c = self.C(b)
```

(continues on next page)

(continued from previous page)

```
# D: pnl.EveryNCalls(B, 10)
if num_B_calls % 10 == 0:
    d = self.D(b)

return c, d
```

The ONNX IR representation of this model is shown below. The small computation sub-graphs contained in the if and else body attributes are not shown. These are either a simple multiplication and addition or an identity.

INTERACTIONS BETWEEN PSYNEULINK AND MDF

20.1 Simple

20.1.1 ABCD

[Python source](#) | [JSON](#) | [Reconstructed source](#)

An example with four Nodes, as in other environments.

20.1.2 SimpleLinear

SimpleLinear-conditional

[Python source](#) | [JSON](#) | [Reconstructed source](#)

A three-Node example with Conditions.

SimpleLinear-timing

[Python source](#) | [JSON](#) | [Reconstructed source](#)

The same model as in SimpleLinear-conditional with Conditions for timeline scheduling. Note: these conditions are still not fully implemented by the scheduler.

20.2 Nested

20.2.1 Nested without scheduling

[Python source](#) | [JSON](#) | [Reconstructed source](#)

A model with several Nodes in two Graphs, one of which contains the other.

20.2.2 Nested with scheduling

[Python source](#) | [JSON](#) | [Reconstructed source](#)

A similar model as in Nested without scheduling with Conditions.

20.3 SimpleFN

[Python source](#) | [JSON](#) | [Reconstructed source](#)

An example with a single Node using the PsyNeuLink implementation of the [FitzHugh–Nagumo model](#).

20.3.1 SimpleFN-timing

[Python source](#) | [JSON](#) | [Reconstructed source](#)

The same model as in SimpleFN with Conditions for timeline scheduling. Note: these conditions are still not fully implemented by the scheduler.

20.3.2 SimpleFN-conditional

[Python source](#) | [JSON](#) | [Reconstructed source](#)

The same model in SimpleFN with scheduling Conditions that mimic the behavior in SimpleFN-timing.

20.4 Stroop

[Python source](#) | [JSON](#) | [Reconstructed source](#)

A model representing the [Stroop effect](#) with conflict monitoring that uses Conditions.

PYTORCH AND MDF

1. MDF to Pytorch
2. Pytorch to MDF

21.1 MDF to PyTorch

To export an MDF model to PyTorch, provide an MDF model as an input to the `mdf_to_pytorch()` function.

The output of `mdf_to_pytorch` is a PyTorch model.

```
mdf_to_pytorch(  
    mdf_model: model in MDF format  
    eval_models: Set Evaluation of model to True or False  
    version: MDF version  
    model_input: input file name  
)
```

It returns a dictionary where `key = model name` and `value = PyTorch model object`.

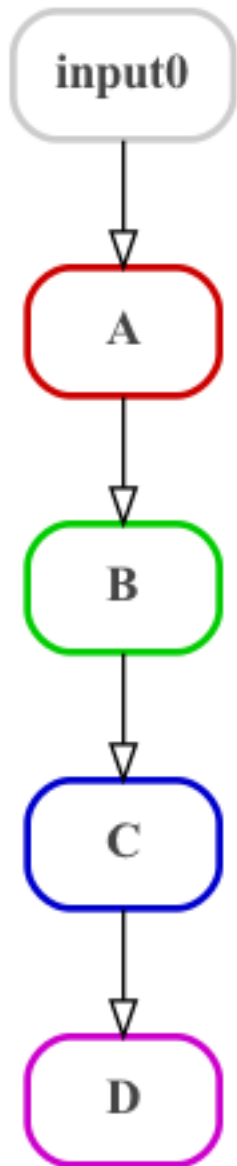
A test script demonstrating conversion of MDF model to PyTorch is at `MDF_to_PyTorch.py`. This converts multiple MDF models to their respective PyTorch models. The converted models are available in folder: `MDF_PyTorch`.

21.1.1 Examples

Below are some working examples of this functionality.

1) Simple ABCD example

We convert one of the sample MDF examples `ABCD.json`:



This is converted to PyTorch and can be seen here: [ABCD_pytorch.py](#).

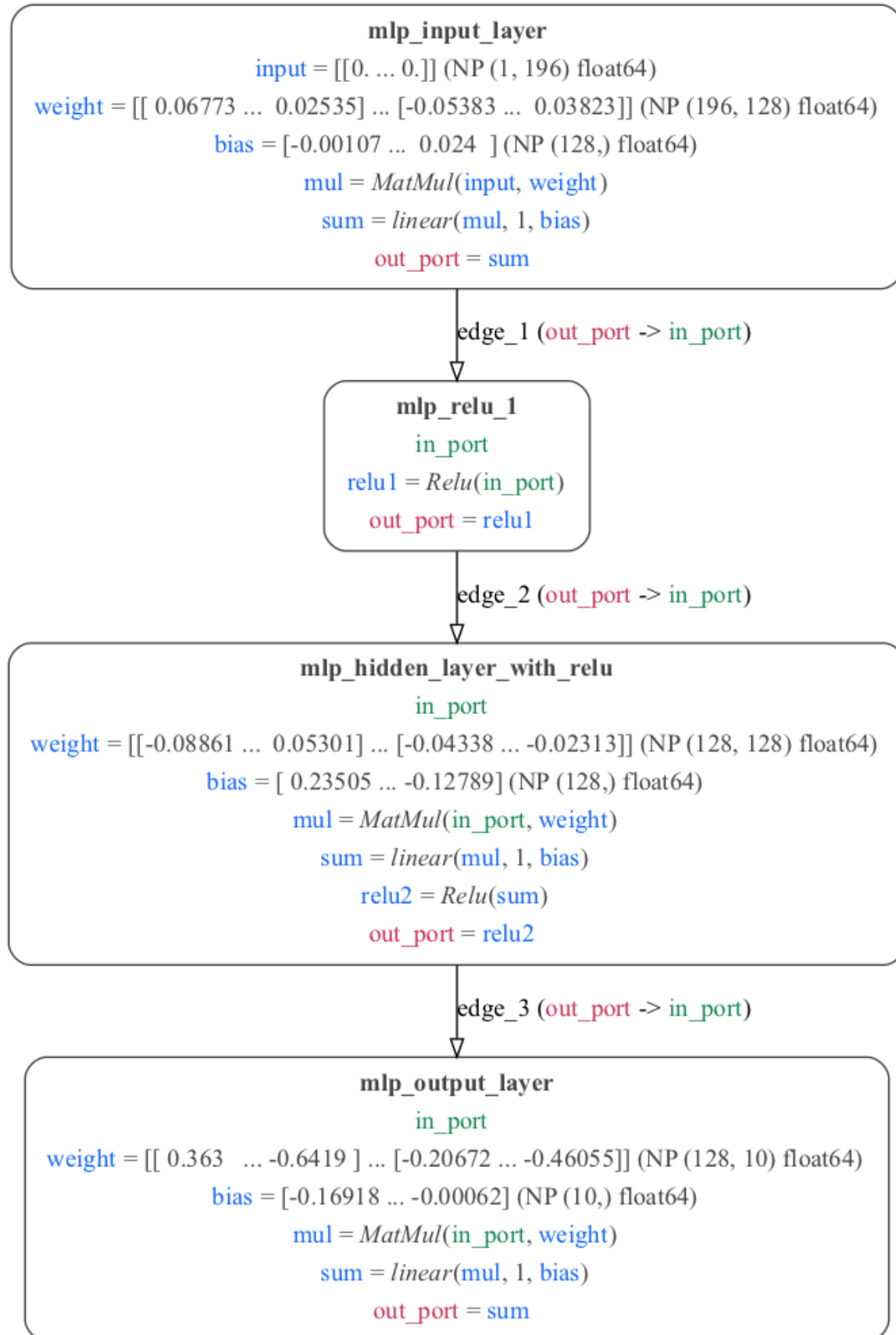
The PyTorch model is further converted to ONNX `ABCD.onnx`. An image of the contents of the ONNX model (visualized using [NETRON](#)) is below.

2) Multi-Layer Perceptron MDF to PyTorch Conversion:

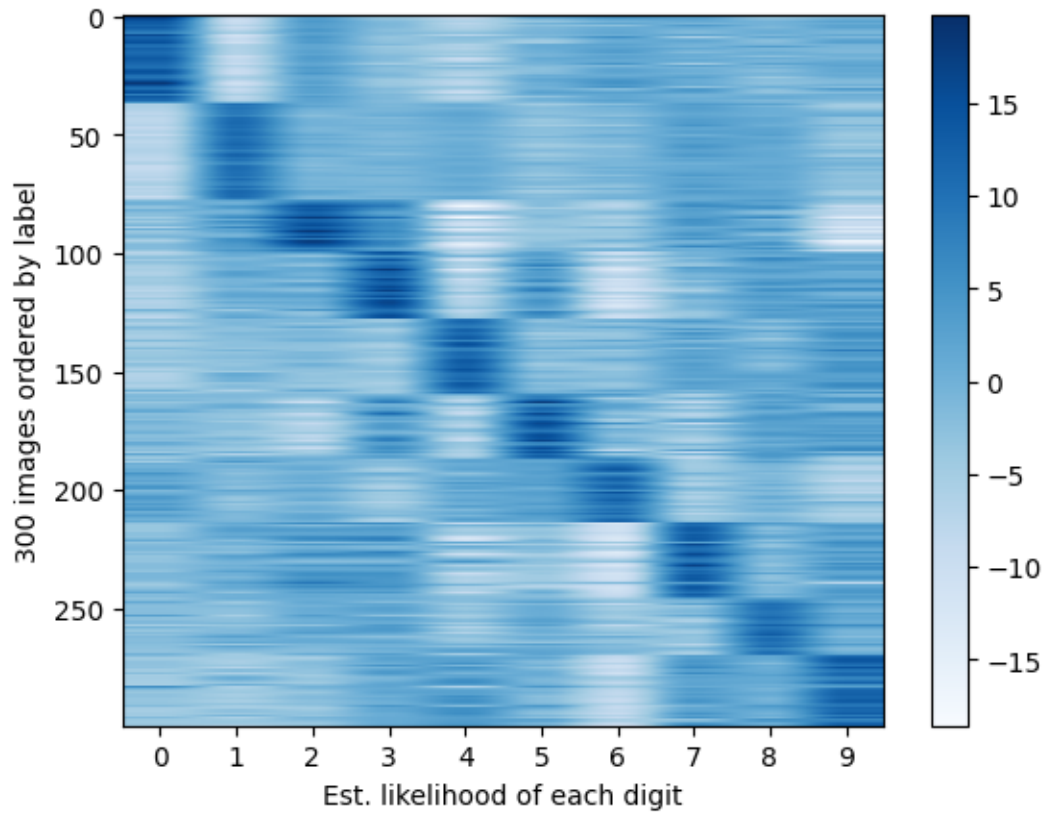
To run an example where a simple Multi-Layer Perceptron (MLP) created using the MDF specification and executed using sample digit-recognition data, run:

```
python mlp_pure_mdf.py
```

A graph of the network can be created with `python mlp_pure_mdf.py -graph`:



The network can be run against images from the MNIST database with: `python mlp_pure_mdf.py -run`, and produce 98% accuracy. The image below shows the results of 300 images:



21.2 PyTorch to MDF

The current implementation of our PyTorch to MDF conversion functionality is built on top of the TorchScript infrastructure provided by PyTorch. PyTorch models that can be translated to TorchScript (via `torch.jit.script` or `torch.jit.trace`) should then be able to be converted to their MDF representation automatically. Below are several working examples of this functionality.

To perform an PyTorch to MDF conversion, provide a PyTorch model as an input to the `pytorch_to_mdf()` function which is available in `importer.py`. The output of `pytorch_to_mdf()` is an MDF model.

```
pytorch_to_mdf(
    model: The model to translate into MDF.
    args: The input arguments for this model. If a nn.Module is passed then the
    ↪ model will be traced with these
        inputs. If a ScriptModule is passed, they are still needed to determine
    ↪ input shapes.
    trace: Force the use of tracing to compile the model. The default is to use
    ↪ torch.jit.script
    use_onnx_ops: Use ONNX ops when possible, fallback to ATEN ops when not
    ↪ available. Default is True. If False,
        use only ATEN ops.
)
```

Returns a translated MDF model.

21.2.1 Examples of usage

1) Simple PyTorch To MDF

This is a simple fully-connected neural network model example consisting of input image of $224 * 224 * 3$ and resulting in two classes as the output To run an example of converting a PyTorch model written in PyTorch to its MDF representation simply run:

```
python simple_pytorch_to_mdf.py
```

Code is present in `simple_pytorch_to_mdf.py` The graph representation of the ONNX model can be generated with:


```
python simple_pytorch_to_mdf.py -graph-onnx
```

NOTE: This command will run the NETRON python server on the local host where we can export the graph as svg/png

The graph representation of the MDF model can be generated with:

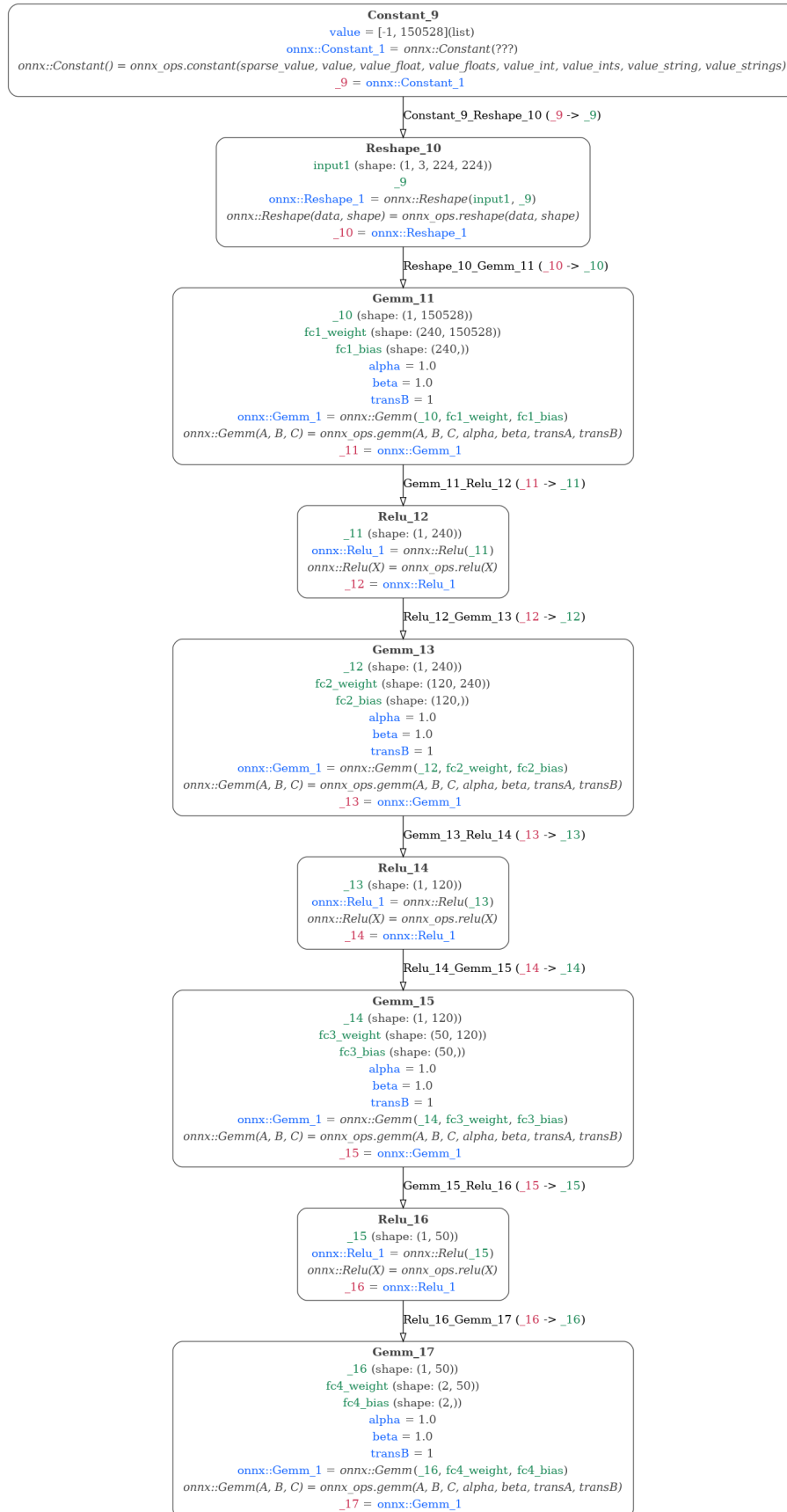
```
python simple_pytorch_to_mdf.py -graph
```

Graphical export from MDF level 1:



api/export_format/PyTorch/simple_pytorch_to_mdf.1.png

Graphical export from MDF level 3:



To visualize the PyTorch model:

```
python simple_pytorch_to_mdf.py -graph-torch
```


The MDF for this model is the written to `simple_pytorch_to_mdf.json`. The model is then executed via the MDF scheduler and the results are compared to the native execution in PyTorch.

2) Inception Blocks Model

To run an example of converting a PyTorch InceptionV3 like model written in PyTorch to its MDF representation simply run:

```
python inception.py
```

Code is present in `inception.py` This will define the model in PyTorch, invoke the TorchScript tracing compiler, convert the underlying IR representation of the model to MDF. The MDF for this model is the written to `inception.json`. The model is then executed via the MDF scheduler and the results are compared to the native execution in PyTorch.

The graph representation of the MDF model can be generated with:

```
python inception.py -graph
```



INTERACTIONS BETWEEN MDF AND QUANTUM COMPUTING TECHNOLOGIES

Starting summer 2021, we will develop tools for interfacing between MDF and quantum computers. This interface is motivated by expectations that quantum hardware will provide speedups for solving Ising-type MDF problems. We will address both gate- and annealing- based quantum computers:

- for gate-based quantum computers, we will bridge from MDF to [OpenQASM](#), the leading quantum Intermediate Representation.
- for annealing-based quantum computers, we will target platforms such as [D-Wave Ocean](#).

Our work will be agnostic to the exact quantum algorithm/solver used, though we will provide sample implementations using Variational Quantum Eigensolver ([VQE](#)) and [Quantum Approximate Optimization Algorithm](#).

As a first step, we have begun developing implementations targeting quantum hardware for the key computations in several cognitive models as listed below. Next, we will extend MDF so that quantum implementations such as the ones we develop, can be expressed in it.

MDF IN WEBGME

This contains a tool for converting the [MDF specification](#) into JSON compatible with [JSON importer](#). This allows us to programmatically create a metamodel and, as a result, use WebGME as a design environment for MDF.

23.1 Quick Start

23.1.1 Starting WebGME app

First, install the `mdf_gme` following:

- [NodeJS](#) (LTS recommended)
- [MongoDB](#)

Second, start `mongodb` locally by running the `mongod` executable in your `mongodb` installation (you may need to create a data directory or set `--dbpath`).

Then, run `webgme start` from the project root to start . Finally, navigate to `http://localhost:8888` to start using `mdf_gme`!

23.1.2 Loading the spec into WebGME

First, install dependencies with `npm install`. Then convert the MDF specification using

```
node spec_to_gme.js path/to/MDF/spec.json
```

Finally, import the JSON into WebGME just like the [examples](#) (suffixed with “_meta”)!

23.1.3 Loading instances to and from WebGME importable JSON and MDF

```
node bin/instance_converter path/to/MDForGME/instance.json
```


SPECIFICATION OF STANDARD FUNCTIONS IN MODECI V0.4

Note: the ModECI MDF specification is still in development! See [here](https://github.com/ModECI/MDF/blob/main/src/modeci_mdf/standard_functions.py) for ongoing discussions. These functions are defined in https://github.com/ModECI/MDF/blob/main/src/modeci_mdf/standard_functions.py

24.1 All of MDF functions:

| MatMul | Relu | change_goal | check_termination | conflict_resolution_function | cos | cosh | exponential | linear | logistic | onnx::Abs | onnx::Acos | onnx::Acosh | onnx::Add | onnx::And | onnx::ArgMax | onnx::ArgMin | onnx::Asin | onnx::Asinh | onnx::Atan | onnx::Atanh | onnx::AveragePool | onnx::BatchNormalization | onnx::Bernoulli | onnx::BitShift | onnx::Cast | onnx::CastLike | onnx::Ceil | onnx::Celu | onnx::Clip | onnx::Compress | onnx::Concat | onnx::ConcatFromSequence | onnx::Constant | onnx::ConstantOfShape | onnx::Conv | onnx::ConvInteger | onnx::ConvTranspose | onnx::Cos | onnx::Cosh | onnx::CumSum | onnx::DepthToSpace | onnx::DequantizeLinear | onnx::Det | onnx::Div | onnx::Dropout | onnx::DynamicQuantizeLinear | onnx::Einsum | onnx::Elu | onnx::Equal | onnx::Erf | onnx::Exp | onnx::Expand | onnx::EyeLike | onnx::Flatten | onnx::Floor | onnx::GRU | onnx::Gather | onnx::GatherElements | onnx::GatherND | onnx::Gemm | onnx::GlobalAveragePool | onnx::GlobalLpPool | onnx::GlobalMaxPool | onnx::Greater | onnx::GreaterOrEqual | onnx::HardSigmoid | onnx::HardSwish | onnx::Hardmax | onnx::Identity | onnx::If | onnx::InstanceNormalization | onnx::IsInf | onnx::IsNaN | onnx::LRN | onnx::LSTM | onnx::LeakyRelu | onnx::Less | onnx::LessOrEqual | onnx::Log | onnx::LogSoftmax | onnx::Loop | onnx::LpNormalization | onnx::LpPool | onnx::MatMul | onnx::MatMulInteger | onnx::Max | onnx::MaxPool | onnx::MaxRoiPool | onnx::MaxUnpool | onnx::Mean | onnx::MeanVarianceNormalization | onnx::Min | onnx::Mod | onnx::Mul | onnx::Multinomial | onnx::Neg | onnx::NegativeLogLikelihoodLoss | onnx::NonMaxSuppression | onnx::NonZero | onnx::Not | onnx::OneHot | onnx::Optional | onnx::OptionalGetElement | onnx::OptionalHasElement | onnx::Or | onnx::PRelu | onnx::Pad | onnx::Pow | onnx::QLinearConv | onnx::QLinearMatMul | onnx::QuantizeLinear | onnx::RNN | onnx::RandomNormal | onnx::RandomNormalLike | onnx::RandomUniform | onnx::RandomUniformLike | onnx::Range | onnx::Reciprocal | onnx::ReduceL1 | onnx::ReduceL2 | onnx::ReduceLogSum | onnx::ReduceLogSumExp | onnx::ReduceMax | onnx::ReduceMean | onnx::ReduceMin | onnx::ReduceProd | onnx::ReduceSum | onnx::ReduceSumSquare | onnx::Relu | onnx::Reshape | onnx::Resize | onnx::ReverseSequence | onnx::RoiAlign | onnx::Round | onnx::Scan | onnx::Scatter | onnx::ScatterElements | onnx::ScatterND | onnx::Selu | onnx::SequenceAt | onnx::SequenceConstruct | onnx::SequenceEmpty | onnx::SequenceErase | onnx::SequenceInsert | onnx::SequenceLength | onnx::Shape | onnx::Shrink | onnx::Sigmoid | onnx::Sign | onnx::Sin | onnx::Sinh | onnx::Size | onnx::Slice | onnx::Softmax | onnx::SoftmaxCrossEntropyLoss | onnx::Softplus | onnx::Softsign | onnx::SpaceToDepth | onnx::Split | onnx::SplitToSequence | onnx::Sqrt | onnx::Squeeze | onnx::StringNormalizer | onnx::Sub | onnx::Sum | onnx::Tan | onnx::Tanh | onnx::TfidfVectorizer | onnx::ThresholdedRelu | onnx::Tile | onnx::TopK | onnx::Transpose | onnx::Trilu | onnx::Unique | onnx::Unsqueeze | onnx::Upsample | onnx::Where | onnx::Xor | pattern_matching_function | retrieve_chunk | sin | sinh | tan | tanh | update_goal | update_retrieval |

24.2 MatMul

24.3 Relu

24.4 change_goal

24.5 check_termination

24.6 conflict_resolution_function

24.7 cos

24.8 cosh

24.9 exponential

24.10 linear

24.11 logistic

24.12 onnx::Abs

24.13 onnx::Acos

24.14 onnx::Acosh

24.15 onnx::Add

This operator supports **multidirectional (i.e., Numpy-style) broadcasting**; for more details please check the doc. (Opset 14 change): Extend supported types to include uint8, int8, uint16, and int16.

24.16 onnx::And

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

24.17 onnx::ArgMax

24.18 onnx::ArgMin

24.19 onnx::Asin

24.20 onnx::Asinh

24.21 onnx::Atan

24.22 onnx::Atanh

24.23 onnx::AveragePool

* 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).

24.24 onnx::BatchNormalization

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

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

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

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

where:

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

Notice that ReduceVar refers to the population variance, and it equals to

(continues on next page)

(continued from previous page)

```
sum(sqrt(x_i - x_avg)) / N
where N is the population size (this formula does not use sample size N - 1).
```

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

When training_mode=False:

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

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

24.25 onnx::Bernoulli

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

24.26 onnx::BitShift

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.

24.27 onnx::Cast

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.

24.28 onnx::CastLike**24.29 onnx::Ceil****24.30 onnx::Celu**

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

24.31 onnx::Clip**24.32 onnx::Compress****24.33 onnx::Concat****24.34 onnx::ConcatFromSequence****24.35 onnx::Constant****24.36 onnx::ConstantOfShape****24.37 onnx::Conv****24.38 onnx::ConvInteger****24.39 onnx::ConvTranspose**

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

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

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

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

```
</i></p>
```

24.40 onnx::Cos

24.41 onnx::Cosh

24.42 onnx::CumSum

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]
```

24.43 onnx::DepthToSpace

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])

24.44 onnx::DequantizeLinear

24.45 onnx::Det

24.46 onnx::Div

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

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

24.47 onnx::Dropout

24.48 onnx::DynamicQuantizeLinear

24.49 onnx::Einsum

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

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

The Einsum operator evaluates algebraic tensor operations on a sequence of tensors, using the Einstein summation convention. The equation string contains a comma-separated sequence of lower case letters. 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.

24.50 onnx::Elu

24.51 onnx::Equal

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

24.52 onnx::Erf

24.53 onnx::Exp

24.54 onnx::Expand

24.55 onnx::EyeLike

24.56 onnx::Flatten

24.57 onnx::Floor

24.58 onnx::GRU

Notations:

X - input tensor

z - update gate

r - reset gate

h - hidden gate

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

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

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

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

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

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

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

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

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

H - Hidden state

num_directions - 2 if direction == bidirectional else 1

Activation functions:

Relu(x) - $\max(0, x)$

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

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

(NOTE: Below are optional)

Affine(x) - $\alpha * x + \beta$

LeakyRelu(x) - x if $x \geq 0$ else $\alpha * x$

ThresholdedRelu(x) - x if $x \geq \alpha$ else 0

ScaledTanh(x) - $\alpha \tanh(\beta x)$

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

Elu(x) - x if $x \geq 0$ else $\alpha(e^x - 1)$

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

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

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

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

24.59 onnx::Gather

axis = 0 :

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

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

axis = 1 :

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

```

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

```

24.60 onnx::GatherElements

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

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

```

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

```

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

Example 1:

```

data = [
    [1, 2],
    [3, 4],
]
indices = [
    [0, 0],
    [1, 0],
]
axis = 1
output = [
    [1, 1],
    [4, 3],
]

```

Example 2:

```

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

```

(continues on next page)

(continued from previous page)

```

]
indices = [
    [1, 2, 0],
    [2, 0, 0],
]
axis = 0
output = [
    [4, 8, 3],
    [7, 2, 3],
]

```

24.61 onnx::GatherND

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

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

Some salient points about the inputs' rank and shape:

- 1) $r \geq 1$ and $q \geq 1$ are to be honored. There is no dependency condition to be met between ranks r and q
- 2) The first b dimensions of the shape of `indices` tensor and `data` tensor must be equal.
- 3) $b < \min(q, r)$ is to be honored.
- 4) The `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]
```

24.62 onnx::Gemm

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

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

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

24.63 onnx::GlobalAveragePool

24.64 onnx::GlobalLpPool

24.65 onnx::GlobalMaxPool

24.66 onnx::Greater

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

24.67 onnx::GreaterOrEqual

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

24.68 onnx::HardSigmoid

24.69 onnx::HardSwish

24.70 onnx::Hardmax

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

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

24.71 onnx::Identity

24.72 onnx::If

24.73 onnx::InstanceNormalization

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

24.74 onnx::IsInf

24.75 onnx::IsNaN

24.76 onnx::LRN

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

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

24.77 onnx::LSTM

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 $\text{direction} == \text{bidirectional}$ else 1

Activation functions:

$\text{Relu}(x) - \max(0, x)$

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

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

(NOTE: Below are optional)

$\text{Affine}(x) - \alpha * x + \beta$

LeakyRelu(x) - x if $x \geq 0$ else $\alpha * x$

ThresholdedRelu(x) - x if $x \geq \alpha$ else 0

ScaledTanh(x) - $\alpha \tanh(\beta x)$

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

Elu(x) - x if $x \geq 0$ else $\alpha * (e^x - 1)$

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

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

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

- $it = f(X_t * (W_i^T) + H_{t-1} * (R_i^T) + P_i) \cdot C_{t-1} + W_{bi} + R_{bi}$
- $ft = f(X_t * (W_f^T) + H_{t-1} * (R_f^T) + P_f) \cdot C_{t-1} + W_{bf} + R_{bf}$
- $ct = g(X_t * (W_c^T) + H_{t-1} * (R_c^T) + W_{bc} + R_{bc})$
- $C_t = ft \cdot C_{t-1} + it \cdot ct$
- $ot = f(X_t * (W_o^T) + H_{t-1} * (R_o^T) + P_o) \cdot C_t + W_{bo} + R_{bo}$
- $H_t = ot \cdot h(C_t)$ This operator has **optional** inputs/outputs. See the doc for more details about the representation of optional arguments. An empty string may be used in the place of an actual argument's name to indicate a missing argument. Trailing optional arguments (those not followed by an argument that is present) may also be simply omitted.

24.78 onnx::LeakyRelu

24.79 onnx::Less

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

24.80 onnx::LessOrEqual

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

24.81 onnx::Log

24.82 onnx::LogSoftmax

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

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

24.83 onnx::Loop

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

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

```
Operator inputs defined as (max_trip_count, condition_var).

input ("", ""):
    for (int i=0; ; ++i) {
        cond = ... // Note this value is ignored, but is required in the body
    }

input ("", cond) // Note this is analogous to a while loop
    bool cond = ...;
    for (int i=0; cond; ++i) {
        cond = ...;
    }

input ("", 1) // Note this is analogous to a do-while loop
    bool cond = true
    for (int i=0; cond; ++i) {
        cond = ...;
    }

input (trip_count, "") // Note this is analogous to a for loop
    int trip_count = ...
    for (int i=0; i < trip_count; ++i) {
        cond = ...; // ignored
    }

input (trip_count, cond)
    int trip_count = ...;
    bool cond = ...;
    for (int i=0; i < trip_count && cond; ++i) {
        cond = ...;
    }
```

Sample usage - cond as well as trip count

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

graph body-net (
```

(continues on next page)

(continued from previous page)

```

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

```

Sample equivalent C code

```

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

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

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

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

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

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

```

There are several things of note in this code snippet:

- 1) Values from the enclosing scope (i.e. variable “a” here) are in scope and can be referenced in the inputs of the loop.
- 2) Any values computed in the loop body that needs to be used in a subsequent iteration or after the loop are modelled using a pair of variables in the loop-body, consisting of an input variable (eg., b_in) and an output variable (eg., b_out). These are referred to as loop-carried dependences. The loop operation node supplies

the input value of the input variable for the first iteration, and returns the output value of the output variable produced by the final iteration.

- 3) Scan_output variables are used to implicitly concatenate values computed across all the iterations. In the above example, the value of user_defined_val computed over all iterations are concatenated and returned as the value of user_defined_vals after the loop.
- 4) Values created in the body cannot be accessed in the enclosing scope, except using the mechanism described above.

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

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

24.84 onnx::LpNormalization

24.85 onnx::LpPool

24.86 onnx::MatMul

24.87 onnx::MatMulInteger

24.88 onnx::Max

24.89 onnx::MaxPool

```
* 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.

24.90 onnx::MaxRoiPool

24.91 onnx::MaxUnpool

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

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

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

24.92 onnx::Mean

24.93 onnx::MeanVarianceNormalization

24.94 onnx::Min

24.95 onnx::Mod

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.

24.96 onnx::Mul

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

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

24.97 onnx::Multinomial

24.98 onnx::Neg

24.99 onnx::NegativeLogLikelihoodLoss

```
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]
```

(continues on next page)

(continued from previous page)

```

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

```

24.100 onnx::NonMaxSuppression**24.101 onnx::NonZero****24.102 onnx::Not****24.103 onnx::OneHot**

```

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.

```

24.104 onnx::Optional

24.105 onnx::OptionalGetElement

24.106 onnx::OptionalHasElement

24.107 onnx::Or

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

24.108 onnx::PRelu

24.109 onnx::Pad

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, empty string, or False)
- 2) `reflect` - pads with the reflection of the vector mirrored on the first and last values of the vector along each axis
- 3) `edge` - pads with the edge values of array

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], ]
```

24.110 onnx::Pow

24.111 onnx::QLinearConv

24.112 onnx::QLinearMatMul

24.113 onnx::QuantizeLinear

24.114 onnx::RNN

Notations:

X - input tensor

i - input gate

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

W_i - W parameter weight matrix for input gate

R_i - R recurrence weight matrix for input gate

W_{bi} - W parameter bias vector for input gate

R_{bi} - R parameter bias vector for input gate

W_{Bi} - W parameter weight matrix for backward input gate

R_{Bi} - R recurrence weight matrix for backward input gate

W_{Bbi} - W parameter bias vectors for backward input gate

R_{Bbi} - R parameter bias vectors for backward input gate

H - Hidden state

$num_directions$ - 2 if $direction == bidirectional$ else 1

Activation functions:

$Relu(x) = \max(0, x)$

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

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

(NOTE: Below are optional)

$Affine(x) = \alpha * x + \beta$

$LeakyRelu(x) = x$ if $x \geq 0$ else $\alpha * x$

$ThresholdedRelu(x) = x$ if $x \geq \alpha$ else 0

$ScaledTanh(x) = \alpha * Tanh(\beta * x)$

$HardSigmoid(x) = \min(\max(\alpha * x + \beta, 0), 1)$

$Elu(x) = x$ if $x \geq 0$ else $\alpha * (e^x - 1)$

$Softsign(x) = x / (1 + |x|)$

$Softplus(x) = \log(1 + e^x)$

Equations (Default: f=Tanh):

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

24.115 onnx::RandomNormal

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.

24.116 onnx::RandomNormalLike

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.

24.117 onnx::RandomUniform

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.

24.118 onnx::RandomUniformLike

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.

24.119 onnx::Range

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]

24.120 onnx::Reciprocal

24.121 onnx::ReduceL1

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

24.122 onnx::ReduceL2

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

24.123 onnx::ReduceLogSum

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

24.124 onnx::ReduceLogSumExp

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

24.125 onnx::ReduceMax

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

24.126 onnx::ReduceMean

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

24.127 onnx::ReduceMin

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

24.128 onnx::ReduceProd

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

24.129 onnx::ReduceSum

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

24.130 onnx::ReduceSumSquare

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

24.131 onnx::Relu

24.132 onnx::Reshape

24.133 onnx::Resize

24.134 onnx::ReverseSequence

For each slice i iterating on batch axis, the operator reverses the first $\text{sequence_lens}[i]$ elements on time axis, and copies elements whose index's beyond $\text{sequence_lens}[i]$ to the output. So the output slice i contains reversed sequences on the first $\text{sequence_lens}[i]$ elements, then have original values copied for the other elements.

Example 1: input = [[0.0, 4.0, 8.0, 12.0], [1.0, 5.0, 9.0, 13.0], [2.0, 6.0, 10.0, 14.0], [3.0, 7.0, 11.0, 15.0]] sequence_lens = [4, 3, 2, 1] time_axis = 0 batch_axis = 1

output = [[3.0, 6.0, 9.0, 12.0], [2.0, 5.0, 8.0, 13.0], [1.0, 4.0, 10.0, 14.0], [0.0, 7.0, 11.0, 15.0]]

Example 2: input = [[0.0, 1.0, 2.0, 3.0], [4.0, 5.0, 6.0, 7.0], [8.0, 9.0, 10.0, 11.0], [12.0, 13.0, 14.0, 15.0]] sequence_lens = [1, 2, 3, 4] time_axis = 1 batch_axis = 0

output = [[0.0, 1.0, 2.0, 3.0], [5.0, 4.0, 6.0, 7.0], [10.0, 9.0, 8.0, 11.0], [15.0, 14.0, 13.0, 12.0]]

24.135 onnx::RoiAlign

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.

24.136 onnx::Round

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]
```

24.137 onnx::Scan

The attribute body must be a graph, specifying the computation to be performed in every iteration. It takes as input the current values of the state_variables and the current iterated element of the scan_inputs. It must return the (updated) values of the state_variables and zero or more scan_output_element tensors. The values of the scan_output_element tensors are concatenated over all the iterations to produce the scan_output values of the scan construct (similar to the concatenated intermediate hidden-state values of RNN-like constructs). All the output tensors (state_variables as well as scan_output_element tensors) are required to have the same shape in each iteration of the loop (a restriction imposed to enable efficient memory allocation).

Note that the iterated element passed to the body subgraph does not have a sequence axis. It will have a rank one less than the rank of the corresponding scan_input.

The scan operation returns the final values of the state_variables as well as the scan_outputs.

The optional attribute scan_input_directions specifies the direction (forward or backward) for each scan input. If this attribute is omitted, all sequences are scanned in the forward direction. A bidirectional scan may be performed by specifying the same tensor input twice in the scan_inputs, once with a forward direction, and once with a backward direction.

The scan_output of the operation is produced by concatenating the scan_output_element values produced by the body in each iteration. The optional attribute scan_output_directions specifies the direction in which scan_output is constructed (by appending or prepending the scan_output_element to scan_output in each iteration) for each scan_output. If this attribute is omitted, the scan_output_element is appended to the scan_output in each iteration.

The optional attribute scan_input_axes specifies the axis to be scanned for each scan_input. If omitted, every scan_input will be scanned in axis 0. For example, if axis 0 is the batch axis and axis 1 is the time axis (to be scanned), specify an axis value of 1. Note that scanning a non-zero axis may be less efficient than scanning axis zero.

The optional attribute scan_output_axes specifies the axis along which the scan_outputs are accumulated for each scan_output. For example, if axis 1 is the time axis (to be scanned) for both inputs and outputs, specify a scan_input axis and scan_output axis value of 1.

Note that because of the ONNX restriction that only the last parameter of an operator can be variadic, the initial-states and scan-inputs are listed together as one input parameter. Similarly, the final-states and scan-outputs are listed together as one output parameter. The attribute num_scan_inputs indicates the number M of scan-inputs.

The behavior of

```
Scan <
  num_scan_inputs = m,
  body = loop-body,
  scan_input_axes = [axis_1, ..., axis_m]
> (init_1, ..., init_n, scan_1, ..., scan_m)
```

is equivalent to the following pseudo-code:

```
// scan_i.shape[axis_i] denotes the (max) sequence-length of scan_i
// scan_i.shape[axis_i] is required to be equal to scan_j.shape[axis_j] for all i,j.
sequence_length = scan_1.shape[axis_1];

// initialize state-variables
st_1 = init_1; ... st_n = init_n;
// initialize scan-output variables: [] denotes an empty tensor
scan_out_1 = []; ...; scan_out_k = [];
// identify number of iterations:

// execute loop
```

(continues on next page)

(continued from previous page)

```

for (int t = 0; t < sequence_length; ++t) {
    // generate the scan-input elements: the notation T<axis=k>[t] indicates the sub-
    ↪ tensor
    // of rank one less than T obtained by indexing T at position t along axis k.
    si_1 = scan_1<axis=axis_1>[t];
    ... ;
    si_m = scan_m<axis=axis_m>[t];
    // execute loop-body
    st_1, ..., st_n, so_1, ..., so_k = loop-body(st_1, ..., st_n, si_1, ..., si_m)
    // accumulate the scan-output elements
    scan_out_1 = Concat<axis=0>(scan_out_1, so_1); ... ; scan_out_k = Concat<axis=0>
    ↪ (scan_out_k, so_k);
}

return st_1, ..., st_n, scan_out_1, ..., scan_out_k;

```

Sample usage: Encoding RNN using a Scan

The following example shows how a simple RNN over an input tensor %X, with weight tensor %Wi, recurrence weight tensor %Ri, bias tensors %Wbi and %Rbi, and initial hidden-state %H_0 can be encoded as a ScanLoop. Note that the loop-body is a nested graph, and it directly computes

values are computed in the outer graph, they need to be passed in as extra state_variables.

```

graph rnn-encoding {
    %H_0 = ...
    %X = ...
    %Y_h, %Y = Scan[body = <graph rnn-cell-1>, num_scan_inputs=1](%H_0, %X)
    return %Y, %Y_h
}

graph rnn-cell-1 (
    %H_tminus1[FLOAT, tensor]
    %X_t[FLOAT, tensor]
) {
    %Wi = ...
    %Ri = ...
    %Wbi = ...
    %Rbi = ...
    %t1 = X_t * (Wi^T)
    %t2 = H_tminus1 * (Ri^T)
    %t3 = Add(%t1, %t2)
    %t4 = Add(%t3, %Wbi)
    %t5 = Add(%t4, %Rbi)
    %Ht = Tanh(%t5)
    %Accumulate = Identity(%Ht)
    return %Ht, %Accumulate
}

```

24.138 onnx::Scatter

Scatter takes three inputs `data`, `updates`, and `indices` of the same rank $r \geq 1$ and an optional attribute `axis` that identifies an axis of `data` (by default, the outer-most axis, that is axis 0). The output of the operation is produced by creating a copy of the input `data`, and then updating its value to values specified by `updates` at specific index positions specified by `indices`. Its output shape is the same as the shape of `data`.

For each entry in `updates`, the target index in `data` is obtained by combining the corresponding entry in `indices` with the index of the entry itself: the index-value for dimension = `axis` is obtained from the value of the corresponding entry in `indices` and the index-value for dimension \neq `axis` is obtained from the index of the entry itself.

For instance, in a 2-D tensor case, the update corresponding to the `[i][j]` entry is performed as below:

```
output[indices[i][j]][j] = updates[i][j] if axis = 0,
output[i][indices[i][j]] = updates[i][j] if axis = 1,
```

This operator is the inverse of `GatherElements`. It is similar to Torch's `Scatter` operation.

Example 1:

```
data = [
  [0.0, 0.0, 0.0],
  [0.0, 0.0, 0.0],
  [0.0, 0.0, 0.0],
]
indices = [
  [1, 0, 2],
  [0, 2, 1],
]
updates = [
  [1.0, 1.1, 1.2],
  [2.0, 2.1, 2.2],
]
output = [
  [2.0, 1.1, 0.0]
  [1.0, 0.0, 2.2]
  [0.0, 2.1, 1.2]
]
```

Example 2:

```
data = [[1.0, 2.0, 3.0, 4.0, 5.0]]
indices = [[1, 3]]
updates = [[1.1, 2.1]]
axis = 1
output = [[1.0, 1.1, 3.0, 2.1, 5.0]]
```

24.139 onnx::ScatterElements

For each entry in `updates`, the target index in `data` is obtained by combining the corresponding entry in `indices` with the index of the entry itself: the index-value for dimension = `axis` is obtained from the value of the corresponding entry in `indices` and the index-value for dimension \neq `axis` is obtained from the index of the entry itself.

For instance, in a 2-D tensor case, the update corresponding to the `[i][j]` entry is performed as below:

```
output[indices[i][j]][j] = updates[i][j] if axis = 0,
output[i][indices[i][j]] = updates[i][j] if axis = 1,
```

This operator is the inverse of `GatherElements`. It is similar to Torch's `Scatter` operation.

Example 1:

```
data = [
    [0.0, 0.0, 0.0],
    [0.0, 0.0, 0.0],
    [0.0, 0.0, 0.0],
]
indices = [
    [1, 0, 2],
    [0, 2, 1],
]
updates = [
    [1.0, 1.1, 1.2],
    [2.0, 2.1, 2.2],
]
output = [
    [2.0, 1.1, 0.0]
    [1.0, 0.0, 2.2]
    [0.0, 2.1, 1.2]
]
```

Example 2:

```
data = [[1.0, 2.0, 3.0, 4.0, 5.0]]
indices = [[1, 3]]
updates = [[1.1, 2.1]]
axis = 1
output = [[1.0, 1.1, 3.0, 2.1, 5.0]]
```

24.140 onnx::ScatterND

`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 $\text{idx1} \neq \text{idx2}$, then $\text{indices}[\text{idx1}] \neq \text{indices}[\text{idx2}]$. 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]]]
```

24.141 onnx::Selu**24.142 onnx::SequenceAt****24.143 onnx::SequenceConstruct****24.144 onnx::SequenceEmpty****24.145 onnx::SequenceErase****24.146 onnx::SequenceInsert****24.147 onnx::SequenceLength****24.148 onnx::Shape**

For example: Input tensor with shape: [2, 3, 4] No attributes specified. Output: [2, 3, 4]

Input tensor with shape: [2, 3, 4] start: -1 Output: [4]

Input tensor with shape: [2, 3, 4] end: -1 Output: [2, 3]

Input tensor with shape: [2, 3, 4] start: 1 end: 2 Output: [3]

24.149 onnx::Shrink**24.150 onnx::Sigmoid****24.151 onnx::Sign****24.152 onnx::Sin****24.153 onnx::Sinh****24.154 onnx::Size****24.155 onnx::Slice**

Slice uses the `starts`, `ends`, `axes` and `steps` inputs to select a sub-tensor of its input data tensor.

An effective `start[i]`, `end[i]`, and `step[i]` must be computed for each `i` in `[0, ... r-1]` where `r = rank(input)` as follows:

If `axes` are omitted, they are set to `[0, ..., r-1]`. If `steps` are omitted, they are set to `[1, ..., 1]` of length `len(starts)`

The effective values are initialized as `start[i] = 0, end[i] = dims[i]` where `dims` are the dimensions of input and `step[i] = 1`.

All negative elements of `axes` are made non-negative by adding `r` to them, where `r = rank(input)`.

All negative values in `starts[i]` and `ends[i]` have `dims[axes[i]]` added to them, where `dims` are the dimensions of input. Then `start[axes[i]]` is the adjusted `starts[i]` is clamped into the range `[0, dims[axes[i]]]` for positive stepping and `[0, dims[axes[i]]-1]` for negative stepping.

The clamping for the adjusted `ends[i]` depends on the sign of `steps[i]` and must accommodate copying 0 through `dims[axes[i]]` elements, so for positive stepping `end[axes[i]]` is clamped to `[0, dims[axes[i]]]`, while for negative stepping it is clamped to `[-1, dims[axes[i]]-1]`.

Finally, `step[axes[i]] = steps[i]`.

For slicing to the end of a dimension with unknown size, it is recommended to pass in `INT_MAX` when slicing forward and `'INT_MIN'` when slicing backward.

Example 1: `data = [[1, 2, 3, 4], [5, 6, 7, 8],] axes = [0, 1] starts = [1, 0] ends = [2, 3] steps = [1, 2] result = [[5, 7],]`

Example 2: `data = [[1, 2, 3, 4], [5, 6, 7, 8],] starts = [0, 1] ends = [-1, 1000] result = [[2, 3, 4],]`

24.156 onnx::Softmax

`Softmax(input, axis) = Exp(input) / ReduceSum(Exp(input), axis=axis, keepdims=1)`

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

24.157 onnx::SoftmaxCrossEntropyLoss

`shape(scores)`: (N, C) where C is the number of classes, or (N, C, D1, D2, ..., Dk), with $K \geq 1$ in case of K-dimensional loss. `shape(labels)`: (N) where each value is $0 \leq \text{labels}[i] \leq C-1$, or (N, D1, D2, ..., Dk), with $K \geq 1$ in case of K-dimensional loss.

The loss for one sample, `l_i`, can be calculated as follows: `l[i][d1][d2]...[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]]`.

24.158 onnx::Softplus

24.159 onnx::Softsign

24.160 onnx::SpaceToDepth

24.161 onnx::Split

24.162 onnx::SplitToSequence

24.163 onnx::Sqrt

24.164 onnx::Squeeze

24.165 onnx::StringNormalizer

24.166 onnx::Sub

This operator supports **multidirectional (i.e., Numpy-style) broadcasting**; for more details please check the doc. (Opset 14 change): Extend supported types to include uint8, int8, uint16, and int16.

24.167 onnx::Sum

24.168 onnx::Tan

24.169 onnx::Tanh

24.170 onnx::TfidfVectorizer

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.

24.171 onnx::ThresholdedRelu

24.172 onnx::Tile

24.173 onnx::TopK

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.

24.174 onnx::Transpose

24.175 onnx::Trilu

24.176 onnx::Unique

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 occurrence in ‘X’.. The third optional output tensor ‘inverse_indices’ contains, for elements of ‘X’, its corresponding indices in ‘Y’. “. The fourth optional output tensor ‘counts’ contains the count of each element of ‘Y’ in the input.

Outputs are either sorted in ascending order or optionally in the order of the first occurrence of the values in the input.

<https://docs.scipy.org/doc/numpy/reference/generated/numpy.unique.html>

Example 1: input_X = [2, 1, 1, 3, 4, 3] attribute_sorted = 0 attribute_axis = None output_Y = [2, 1, 3, 4] output_indices = [0, 1, 3, 4] output_inverse_indices = [0, 1, 1, 2, 3, 2] output_counts = [1, 2, 2, 1]

Example 2: input_X = [[1, 3], [2, 3]] attribute_sorted = 1 attribute_axis = None output_Y = [1, 2, 3] output_indices = [0, 2, 1] output_inverse_indices = [0, 2, 1, 2] output_counts = [1, 1, 2]

Example 3: input_X = [[1, 0, 0], [1, 0, 0], [2, 3, 4]] attribute_sorted = 1 attribute_axis = 0 output_Y = [[1, 0, 0], [2, 3, 4]] output_indices = [0, 2] output_inverse_indices = [0, 0, 1] output_counts = [2, 1]

Example 4: input_x = [[[1., 1.], [0., 1.], [2., 1.], [0., 1.]], [[1., 1.], [0., 1.], [2., 1.], [0., 1.]]] attribute_sorted = 1 attribute_axis = 1

intermediate data are presented below for better understanding:

there are 4 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]`

24.177 onnx::Unsqueeze

For example: Given an input tensor (`data`) of shape `[3, 4, 5]`, then `Unsqueeze(data, axes=[0, 4])` outputs a tensor (`expanded`) containing same data as `data` but with shape `[1, 3, 4, 5, 1]`.

The input `axes` should not contain any duplicate entries. It is an error if it contains duplicates. The rank of the output tensor (`output_rank`) is the rank of the input tensor (`data`) plus the number of values in `axes`. Each value in `axes` should be within the (inclusive) range `[-output_rank, output_rank - 1]`. The order of values in `axes` does not matter and can come in any order.

24.178 onnx::Upsample

24.179 onnx::Where

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

24.180 onnx::Xor

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

24.181 pattern_matching_function

24.182 retrieve_chunk

24.183 sin

24.184 sinh

24.185 tan

24.186 tanh

24.187 update_goal

24.188 update_retrieval

MODECI_MDF

MDF is intended to be an open source, community-supported standard and associated library of tools for expressing computational models in a form that allows them to be exchanged between diverse programming languages and execution environments. The MDF Python API can be used to create or load an MDF model for inspection and validation. It also includes a basic execution engine for simulating models in the format. However, this is not intended as a general purpose simulation environment, nor is MDF intended as a programming language. Rather, the primary purpose of the Python API is to facilitate and validate the exchange of models between existing environments that serve different communities. Accordingly, these Python tools include bi-directional support for importing to and exporting from widely-used programming environments in a range of disciplines, and for easily extending these to other environments.

<code>modeci_mdf.execution_engine</code>	The reference implementation of the MDF execution engine; allows for executing <code>Graph</code> objects in Python.
<code>modeci_mdf.full_translator</code>	
<code>modeci_mdf.functions</code>	Specifies and implements the MDF the function ontology; a collection of builtin functions that can be used in <code>MDF Function</code> and <code>Parameter</code> objects.
<code>modeci_mdf.interfaces</code>	Implementations of importers and exporters for supported environments; fulfilling the hub and spoke model of MDF by allowing exchange between different modeling environments via MDF.
<code>modeci_mdf.mdf</code>	The main object-oriented implementation of the MDF schema, with each core component of the MDF specification implemented as a <code>class</code> .
<code>modeci_mdf.utils</code>	Useful utility functions for dealing with MDF objects.

25.1 modeci_mdf.execution_engine

The reference implementation of the MDF execution engine; allows for executing `Graph` objects in Python.

This module implements a set of classes for executing loaded MDF models in Python. The implementation is organized such that each class present in `mdf` has a corresponding `Evaluable` version of the class. Each of these classes implements the execution of these components and tracks their state during execution. The organization of the entire execution of the model is implemented at the top-level `evaluate()` method of the `EvaluableGraph` class. The external library [graph-scheduler](#) is used to implement the scheduling of nodes under declarative conditional constraints.

Functions

<code>evaluate_expr([expr, func_params, ...])</code>	Evaluates an expression given in string format and a dict of parameters.
<code>evaluate_onnx_expr(expr, base_parameters, ...)</code>	Evaluates a simple expression in string format representing an ONNX function call
<code>get_required_variables_from_expression(expr)</code>	Produces a list containing variable symbols in expr
<code>main(example_file[, array_format, verbose])</code>	Main entry point for execution engine.

25.1.1 modeci_mdf.execution_engine.evaluate_expr

`modeci_mdf.execution_engine.evaluate_expr` (*expr*: Union[*str*, List[*str*], *numpy.ndarray*, *tf.tensor*] = None, *func_params*: Dict[*str*, Any] = None, *array_format*: *str* = 'numpy', *allow_strings_returned*: Optional[bool] = False, *verbose*: Optional[bool] = False) → *numpy.ndarray*

Evaluates an expression given in string format and a dict of parameters.

Parameters

- **expr** – Expression or list of expressions to be evaluated
- **func_params** – A dict of parameters (e.g. {'weight': 2})
- **array_format** – It can be a n-dimensional array or a tensor
- **allow_strings_returned** – Don't throw an error if the expression evaluates to a string
- **verbose** – If set to True provides in-depth information else verbose message is not displayed

Returns n-dimensional array

25.1.2 modeci_mdf.execution_engine.evaluate_onnx_expr

`modeci_mdf.execution_engine.evaluate_onnx_expr` (*expr*: *str*, *base_parameters*: Dict[*str*, Any], *evaluated_parameters*: Dict, *verbose*: bool = False) → Any

Evaluates a simple expression in string format representing an ONNX function call

Parameters

- **expr** (*str*) – Expression to be evaluated
- **base_parameters** (Dict[*str*, Any]) – A dict of parameters that may contain variables
- **evaluated_parameters** (Dict) – A dict mapping variables used in **base_parameters** to actual values
- **verbose** (*bool*, *optional*) – If set to True provides in-depth information else verbose message is not displayed. Defaults to False.

Returns the return value of **expr**

Return type Any

25.1.3 modeci_mdf.execution_engine.get_required_variables_from_expression

`modeci_mdf.execution_engine.get_required_variables_from_expression` (*expr*: *str*) → *List[str]*

Produces a list containing variable symbols in **expr**

25.1.4 modeci_mdf.execution_engine.main

`modeci_mdf.execution_engine.main` (*example_file*: *str*, *array_format*: *str* = 'numpy', *verbose*: *bool* = False)

Main entry point for execution engine.

Parameters

- **example_file** – The MDF file to execute.
- **array_format** – The format of arrays to use. Allowed values: 'numpy' or 'tensorflow'.
- **verbose** – Whether to print output to standard out during execution.

Classes

<i>EvaluableFunction</i> (function, verbose)	Evaluates a <i>Function</i> value during MDF graph execution.
<i>EvaluableGraph</i> (graph, verbose)	Evaluates a <i>Graph</i> with the MDF execution engine.
<i>EvaluableInput</i> (input_port, verbose)	Evaluates input value at the <i>InputPort</i> of the node during MDF graph execution.
<i>EvaluableNode</i> (node, verbose)	Evaluates a <i>Node</i> during MDF graph execution.
<i>EvaluableOutput</i> (output_port, verbose)	Evaluates the current value of an <i>OutputPort</i> during MDF graph execution.
<i>EvaluableParameter</i> (parameter, verbose)	Evaluates the current value of a <i>Parameter</i> during the MDF graph execution.

25.1.5 modeci_mdf.execution_engine.EvaluableFunction

class `modeci_mdf.execution_engine.EvaluableFunction` (*function*: *mod-eci_mdf.mdf.Function* = False, *verbose*: *Optional[bool]* = False)

Bases: `object`

Evaluates a *Function* value during MDF graph execution.

Parameters

- **function** – *Function()* to be evaluated e.g. mdf standard function
- **verbose** – If set to True Provides in-depth information else verbose message is not displayed

Methods

<code>evaluate([parameters, array_format])</code>	Performs evaluation on the basis of given parameters and array_format
---	---

evaluate (*parameters: Optional[Dict[str, Any]] = None, array_format: str = 'numpy'*) → Dict[str, Any]
 Performs evaluation on the basis of given parameters and array_format

Parameters

- **parameters** – A dictionary of function parameters, e.g. logistic, parameters={ 'gain': 2, 'bias': 3, 'offset': 1 }
- **array_format** – It can be a n-dimensional array or a tensor

Returns value of function after evaluation in Dictionary

25.1.6 modeci_mdf.execution_engine.EvaluableGraph

class modeci_mdf.execution_engine.**EvaluableGraph** (*graph: modeci_mdf.mdf.Graph, verbose: Optional[bool] = False*)

Bases: `object`

Evaluates a *Graph* with the MDF execution engine. This is the top-level interface to the execution engine.

Parameters

- **graph** – A directed graph consisting of *Node*(s) connected via *Edge*(s)
- **verbose** – If set to True Provides in-depth information else verbose message is not displayed

Methods

<code>evaluate([time_increment, array_format, ...])</code>	Evaluates a <i>Graph</i> .
<code>evaluate_edge(edge[, time_increment, ...])</code>	Evaluates edges in graph
<code>parse_condition(condition)</code>	Convert the condition in a specific format

evaluate (*time_increment: Optional[Union[int, float]] = None, array_format: str = 'numpy', initializer: Optional[Dict[str, Any]] = None*)
 Evaluates a *Graph*. This is the top-level interface to the execution engine.

Parameters

- **time_increment** – Time step for next execution
- **array_format** – A n-dimensional array
- **initializer** – sets the initial value of parameters of the node

evaluate_edge (*edge: modeci_mdf.mdf.Edge, time_increment: Optional[Union[int, float]] = None, array_format: str = 'numpy'*)
 Evaluates edges in graph

Parameters

- **time_increment** – Time step for next execution

- **array_format** – A n-dimensional array

parse_condition (*condition*: *Union[modeci_mdf.mdf.Condition, Dict]*) → *graph_scheduler.condition.Condition*
 Convert the condition in a specific format

Parameters **condition** – Specify the condition under which a Component should be allowed to execute

Returns Condition in specific format

25.1.7 modeci_mdf.execution_engine.EvaluableInput

class modeci_mdf.execution_engine.**EvaluableInput** (*input_port*: *mod-eci_mdf.mdf.InputPort*, *verbose*: *Optional[bool] = False*)

Bases: *object*

Evaluates input value at the *InputPort* of the node during MDF graph execution.

Parameters

- **input_port** – The *InputPort* is an attribute of a Node which imports information to the *Node*
- **verbose** – If set to True Provides in-depth information else verbose message is not displayed

Methods

<i>evaluate</i> ([parameters, array_format])	Evaluates value at Input port based on parameters and array_format
<i>set_input_value</i> (value)	Set a new value at input port

set_input_value (*value*: *Union[str, int, numpy.ndarray]*)
 Set a new value at input port

Parameters **value** – Value to be set at Input Port

evaluate (*parameters*: *Optional[Dict[str, Any]] = None*, *array_format*: *str = 'numpy'*) → *Union[int, numpy.ndarray]*
 Evaluates value at Input port based on parameters and array_format

Parameters

- **parameters** – Dictionary of parameters
- **array_format** – It is a n-dimensional array

Returns value at Input port

25.1.8 modeci_mdf.execution_engine.EvaluableNode

class modeci_mdf.execution_engine.**EvaluableNode** (*node*: modeci_mdf.mdf.Node, *verbose*: *Optional[bool]* = *False*)

Bases: `object`

Evaluates a *Node* during MDF graph execution.

Parameters

- **node** – A self contained unit of evaluation receiving input from other *Node*(s) on *InputPort*(s).
- **verbose** – If set to True Provides in-depth information else verbose message is not displayed

Methods

<code>evaluate</code> ([time_increment, array_format])	Evaluate the Node for one time-step
<code>get_output</code> ([id])	Get value at output port for given output port's id

evaluate (*time_increment*: *Optional[Union[int, float]]* = *None*, *array_format*: *str* = 'numpy')

Evaluate the Node for one time-step

Parameters

- **time_increment** – The time-increment to use for this evaluation.
- **array_format** – The format to use for arrays.

get_output (*id*: *Optional[str]* = *None*) → Union[int, numpy.ndarray, Tuple]

Get value at output port for given output port's id

Parameters *id* – Unique identifier of the output port. If None, return a tuple for all output ports.

Returns value at the output port. If id is None, return all outputs as a tuple. If there is only one output, return just its value.

25.1.9 modeci_mdf.execution_engine.EvaluableOutput

class modeci_mdf.execution_engine.**EvaluableOutput** (*output_port*: modeci_mdf.mdf.OutputPort, *verbose*: *Optional[bool]* = *False*)

Bases: `object`

Evaluates the current value of an *OutputPort* during MDF graph execution.

Parameters

- **output_port** – Attribute of a Node which exports information to the dependent Node object
- **verbose** – If set to True Provides in-depth information else verbose message is not displayed

Methods

<code>evaluate([parameters, array_format])</code>	Evaluate the value at the output port on the basis of parameters and array_format
---	---

evaluate (*parameters: Optional[Dict[str, Any]] = None, array_format: str = 'numpy'*) → Union[int, numpy.ndarray]

Evaluate the value at the output port on the basis of parameters and array_format

Parameters

- **parameters** – Dictionary of global parameters of the Output Port
- **array_format** – It is a n-dimensional array

Returns value at output port

25.1.10 modeci_mdf.execution_engine.EvaluableParameter

class modeci_mdf.execution_engine.**EvaluableParameter** (*parameter: mod-eci_mdf.mdf.Parameter, verbose: bool = False*)

Bases: object

Evaluates the current value of a *Parameter* during the MDF graph execution.

Parameters

- **parameter** – The parameter to evaluate during execution.
- **verbose** – Whether to print output of parameter calculations.

Methods

<code>evaluate(parameters[, time_increment, ...])</code>	Evaluate the parameter and store the result in the <code>curr_value</code> attribute.
<code>get_current_value(parameters[, array_format])</code>	Get the current value of the parameter; evaluates the expression if the current value has not yet been set.

get_current_value (*parameters: Dict[str, Any], array_format: str = 'numpy'*) → Any

Get the current value of the parameter; evaluates the expression if the current value has not yet been set. Note: this is different from 'evaluate', as calling that method multiple times can change the state of the parameter, but calling this should not reevaluate the parameter if it has a current value.

Parameters

- **parameters** – a dictionary of parameters and their values that may or may not be needed to evaluate this parameter.
- **array_format** – The array format to use (either 'numpy' or tensorflow').

Returns The evaluated value of the parameter.

evaluate (*parameters: Dict[str, Any], time_increment: Optional[float] = None, array_format: str = 'numpy'*) → Any

Evaluate the parameter and store the result in the `curr_value` attribute.

Parameters

- **parameters** – a dictionary of parameters and their values that may or may not be needed to evaluate this parameter.
- **time_increment** – a floating point value specifying the timestep size, only used for `time_derivative` parameters
- **array_format** – The array format to use (either 'numpy' or tensorflow').

Returns The current value of the parameter.

25.2 modeci_mdf.full_translator

Functions

<code>convert_states_to_stateful_parameters([.T])</code>	Translates json file if with states to json file with stateful_parameters, otherwise unchanged :param file_path: File in Json Format
--	--

25.2.1 modeci_mdf.full_translator.convert_states_to_stateful_parameters

`modeci_mdf.full_translator.convert_states_to_stateful_parameters` (*file_path:* *Optional[str]*
= *None*,
dt=5e-05)

Translates json file if with states to json file with stateful_parameters, otherwise unchanged :param file_path: File in Json Format

Returns file in json format

25.3 modeci_mdf.functions

Specifies and implements the MDF the function ontology; a collection of builtin functions that can be used in MDF `Function` and `Parameter` objects. Code for registering standard functions are with the ontology is implemented in *standard*. If you want to add functions to the ontology that can be used during execution see how this has been done for ONNX (*onnx*) and ACT-R (*actr*)

<code>modeci_mdf.functions.actr</code>	Contains implementations of ACT-R functions using the ccm library.
<code>modeci_mdf.functions.onnx</code>	Programmatically defines every ONNX operation as a python callable function.
<code>modeci_mdf.functions.standard</code>	Implementation of core MDF function ontology.

25.3.1 modeci_mdf.functions.actr

Contains implementations of ACT-R functions using the ccm library.

Functions

<code>change_goal(pattern, curr_goal)</code>	Modifies the current goal buffer using the given pattern.
<code>check_termination(production)</code>	Function used to check if no production was selected.
<code>chunk_to_string(chunk)</code>	Converts a chunk dictionary to a string format.
<code>conflict_resolution_function(productions)</code>	ACT-R conflict resolution function.
<code>get_actr_functions()</code>	Creates a list of all the ACT-R functions as MDF specifications.
<code>match_production(production, context)</code>	Returns True if the production's left hand side matches the given context and adds the matching bindings to the production.
<code>pattern_matching_function(productions, goal, ...)</code>	Returns the productions that match the given goal and retrieval buffers.
<code>pattern_to_string(pattern)</code>	Converts a pattern dictionary to a string format.
<code>retrieve_chunk(pattern, dm_chunks, types)</code>	Retrieve a chunk from declarative memory given a pattern.
<code>update_buffer(production, buffer)</code>	Returns a pattern to update the given buffer with.
<code>update_goal(production)</code>	Returns a pattern to update the goal buffer with.
<code>update_retrieval(production)</code>	Returns a pattern to update the retrieval buffer with.

modeci_mdf.functions.actr.change_goal

`modeci_mdf.functions.actr.change_goal` (*pattern: Dict[str, str], curr_goal: Dict[str, str]*)

Modifies the current goal buffer using the given pattern.

Parameters

- **pattern** – A dict representing a pattern.
- **curr_goal** – A dict representing the current goal pattern.

Returns The current goal updated with the data in pattern.

modeci_mdf.functions.actr.check_termination

`modeci_mdf.functions.actr.check_termination` (*production: Dict[str, Any]*)

Function used to check if no production was selected.

Parameters **production** – A production dict that was selected.

Returns True if the production is empty.

modeci_mdf.functions.actr.chunk_to_string

`modeci_mdf.functions.actr.chunk_to_string(chunk: Dict[str, str]) → str`

Converts a chunk dictionary to a string format.

Parameters `chunk` – A dict representing a chunk.

Returns A string representation of the chunk.

modeci_mdf.functions.actr.conflict_resolution_function

`modeci_mdf.functions.actr.conflict_resolution_function(productions: List[Dict[str, Any]]) → Dict[str, Any]`

ACT-R conflict resolution function. Currently selects a production at random from the already matched productions, since utility values and learning are not implemented yet.

Parameters `productions` – A list of productions as dicts.

Returns The selected production from the list.

modeci_mdf.functions.actr.get_actr_functions

`modeci_mdf.functions.actr.get_actr_functions() → List[Dict[str, Any]]`

Creates a list of all the ACT-R functions as MDF specifications.

Returns A list of MDF function specifications.

modeci_mdf.functions.actr.match_production

`modeci_mdf.functions.actr.match_production(production: Dict[str, Any], context: Dict[str, Dict[str, str]]) → bool`

Returns True if the production's left hand side matches the given context and adds the matching bindings to the production.

Parameters

- **production** – A dict representing a production.
- **context** – A dict with the contents of the goal and retrieval buffers.

Returns True if the production's left hand side matches the context.

modeci_mdf.functions.actr.pattern_matching_function

`modeci_mdf.functions.actr.pattern_matching_function(productions: List[Dict[str, Any]], goal: Dict[str, str], retrieval: Dict[str, str]) → List[Dict[str, Any]]`

Returns the productions that match the given goal and retrieval buffers.

Parameters

- **productions** – A list of all productions as dicts.
- **goal** – The current value of the goal buffer as a dict.
- **retrieval** – The chunk dict retrieved from declarative memory.

Returns A list of productions that match the buffers.

modeci_mdf.functions.actr.pattern_to_string

`modeci_mdf.functions.actr.pattern_to_string (pattern: Dict[str, str]) → str`

Converts a pattern dictionary to a string format.

Parameters **chunk** – A dict representing a pattern.

Returns A string representation of the pattern.

modeci_mdf.functions.actr.retrieve_chunk

`modeci_mdf.functions.actr.retrieve_chunk (pattern: Dict[str, str], dm_chunks: List[Dict[str, str]], types: Dict[str, List[str]]) → Dict[str, str]`

Retrieve a chunk from declarative memory given a pattern.

Parameters

- **pattern** – A dict representing the pattern to match.
- **dm_chunks** – A list of dicts, each representing a chunk in declarative memory.
- **types** – A dict containing each possible chunk type.

Returns The chunk in declarative memory that matches the pattern.

modeci_mdf.functions.actr.update_buffer

`modeci_mdf.functions.actr.update_buffer (production: Dict[str, Any], buffer: str) → Dict[str, str]`

Returns a pattern to update the given buffer with.

Parameters

- **production** – The production dict that specifies the buffer update.
- **buffer** – The name of the buffer to update.

Returns A pattern that the buffer will be updated with.

modeci_mdf.functions.actr.update_goal

`modeci_mdf.functions.actr.update_goal (production: Dict[str, Any]) → Dict[str, str]`

Returns a pattern to update the goal buffer with.

Parameters **production** – The production dict that specifies the goal buffer update.

Returns A pattern that the goal buffer will be updated with.

modeci_mdf.functions.actr.update_retrieval

`modeci_mdf.functions.actr.update_retrieval (production: Dict[str, Any]) → Dict[str, str]`

Returns a pattern to update the retrieval buffer with.

Parameters **production** – The production dict that specifies the retrieval buffer update.

Returns A pattern that the retrieval buffer will be updated with.

modeci-mdf

modeci_mdf.functions.actr.ccm

modeci_mdf.functions.actr.ccm

modeci_mdf.functions.actr.ccm.buffer

modeci_mdf.functions.actr.ccm.dm

modeci_mdf.functions.actr.ccm.logger

modeci_mdf.functions.actr.ccm.model

modeci_mdf.functions.actr.ccm.pattern

*modeci_mdf.functions.actr.ccm.
scheduler*

modeci_mdf.functions.actr.ccm.buffer

Classes

Buffer()

Chunk(contents[, bound])

modeci_mdf.functions.actr.ccm.buffer.Buffer

class modeci_mdf.functions.actr.ccm.buffer.**Buffer**
Bases: *modeci_mdf.functions.actr.ccm.model.Model*

Methods

clear()

isEmpty()

*modify(**args)*

set(chunk)

modeci_mdf.functions.actr.ccm.buffer.Chunk

class modeci_mdf.functions.actr.ccm.buffer.**Chunk** (*contents, bound=None*)
 Bases: `collections.UserDict`

Methods**modeci_mdf.functions.actr.ccm.dm****Classes**

Associated(a, b)

BlendingMemory(buffer[, latency, threshold, ...])

DMAssociate(memory, buffer[, weight, decay, ...])

DMBaseLevel(memory[, decay, limit])

DMFixed(memory[, default])

DMInhibition(memory[, decayScale, timeScale])

DMNoise(memory[, noise, baseNoise])

DMSalience(memory)

DMSpacing(memory[, decayScale, decayIntercept])

DMSpreading(memory, *buffers)

First(parent[, size, time])

Memory(buffer[, latency, threshold, ...])

MemorySubModule(parent)

Partial(memory[, strength, limit])

modeci_mdf.functions.actr.ccm.dm.Associated

class modeci_mdf.functions.actr.ccm.dm.**Associated**(*a, b*)
Bases: *object*

Methods

modeci_mdf.functions.actr.ccm.dm.BlendingMemory

class modeci_mdf.functions.actr.ccm.dm.**BlendingMemory**(*buffer, latency=0.05, threshold=0, maximum_time=10.0, first_size=4, first_time=3.0*)
Bases: *modeci_mdf.functions.actr.ccm.dm.Memory*

Methods

recall(chunk, matches, request_number)

modeci_mdf.functions.actr.ccm.dm.DMAssociate

class modeci_mdf.functions.actr.ccm.dm.**DMAssociate**(*memory, buffer, weight=1, decay=0.5, limit=None*)
Bases: *modeci_mdf.functions.actr.ccm.dm.MemorySubModule*

Methods

activation(chunk)

recalled(chunk)

set_association(pre, post, baselevel)

modeci_mdf.functions.actr.ccm.dm.DMBaseLevel

class modeci_mdf.functions.actr.ccm.dm.**DMBaseLevel** (*memory, decay=0.5, limit=None*)
Bases: *modeci_mdf.functions.actr.ccm.dm.MemorySubModule*

Methods

activation(chunk)

create(chunk[, time, baselevel])

merge(chunk[, time, baselevel])

modeci_mdf.functions.actr.ccm.dm.DMFixed

class modeci_mdf.functions.actr.ccm.dm.**DMFixed** (*memory, default=0*)
Bases: *modeci_mdf.functions.actr.ccm.dm.MemorySubModule*

Methods

activation(chunk)

create(chunk[, fixed])

merge(chunk[, fixed])

modeci_mdf.functions.actr.ccm.dm.DMIInhibition

class modeci_mdf.functions.actr.ccm.dm.**DMIInhibition** (*memory, decayScale=1.0, timeScale=5.0*)
Bases: *modeci_mdf.functions.actr.ccm.dm.MemorySubModule*

Methods

activation(chunk)

create(chunk[, time])

merge(chunk[, time])

modeci_mdf.functions.actr.ccm.dm.DMNoise

class modeci_mdf.functions.actr.ccm.dm.**DMNoise** (*memory, noise=0.3, baseNoise=0.0*)
Bases: *modeci_mdf.functions.actr.ccm.dm.MemorySubModule*

Methods

activation(chunk)

create(chunk, **keys)

logisticNoise(s)

modeci_mdf.functions.actr.ccm.dm.DMSalience

class modeci_mdf.functions.actr.ccm.dm.**DMSalience** (*memory*)
Bases: *modeci_mdf.functions.actr.ccm.dm.MemorySubModule*

Methods

activation(chunk)

context(pattern)

weights(**weights)

modeci_mdf.functions.actr.ccm.dm.DMSpacing

class modeci_mdf.functions.actr.ccm.dm.**DMSpacing** (*memory, decayScale=0.0, decayIntercept=0.5*)
Bases: *modeci_mdf.functions.actr.ccm.dm.MemorySubModule*

Methods

activation(chunk)

create(chunk[, time])

merge(chunk[, time])

modeci_mdf.functions.actr.ccm.dm.DMSpreading

```
class modeci_mdf.functions.actr.ccm.dm.DMSpreading(memory, *buffers)  
    Bases: modeci_mdf.functions.actr.ccm.dm.MemorySubModule
```

Methods

```
activation(chunk)
```

```
create(chunk, **keys)
```

modeci_mdf.functions.actr.ccm.dm.Finst

```
class modeci_mdf.functions.actr.ccm.dm.Finst(parent, size=4, time=3.0)  
    Bases: object
```

Methods

```
add(o)
```

```
contains(o)
```

```
remove(o)
```

modeci_mdf.functions.actr.ccm.dm.Memory

```
class modeci_mdf.functions.actr.ccm.dm.Memory(buffer, latency=0.05, threshold=0,  
                                              maximum_time=10.0, finst_size=4,  
                                              finst_time=3.0)  
    Bases: modeci_mdf.functions.actr.ccm.model.Model
```

Methods

```
add(chunk[, record])
```

```
add_adaptor(a)
```

```
clear()
```

```
fail(request_number)
```

```
find_matching_chunks(pattern[, threshold])
```

continues on next page

Table 30 – continued from previous page

`get_activation(chunk)`

`recall(chunk, matches, request_number)`

`request(pattern[, partial, require_new])`

modeci_mdf.functions.actr.ccm.dm.MemorySubModule

```
class modeci_mdf.functions.actr.ccm.dm.MemorySubModule (parent)
    Bases: object
```

Methods

`activation(chunk)`

`create(chunk, **keys)`

`matched(chunks)`

`merge(chunk, **keys)`

`now()`

`recalled(chunk)`

modeci_mdf.functions.actr.ccm.dm.Partial

```
class modeci_mdf.functions.actr.ccm.dm.Partial (memory, strength=1.0, limit=- 1.0)
    Bases: object
```

Methods

`match(key, a, b)`

`request(pattern)`

`similarity(a, b, value)`

modeci_mdf.functions.actr.ccm.logger**Functions**

file_exists(filename)

finished([flush])

log([screen, html, data, summary, directory])

modeci_mdf.functions.actr.ccm.logger.file_exists

modeci_mdf.functions.actr.ccm.logger.**file_exists** (filename)

modeci_mdf.functions.actr.ccm.logger.finished

modeci_mdf.functions.actr.ccm.logger.**finished** (flush=True)

modeci_mdf.functions.actr.ccm.logger.log

modeci_mdf.functions.actr.ccm.logger.**log** (screen=None, html=None, data=None, summary=None, directory=None)

Classes

DummyLog()

Log()

LogProxy(log[, prefix])

Trace()

modeci_mdf.functions.actr.ccm.logger.DummyLog

class modeci_mdf.functions.actr.ccm.logger.**DummyLog**
Bases: `object`

Methods

set(key, value)

modeci_mdf.functions.actr.ccm.logger.Log

class modeci_mdf.functions.actr.ccm.logger.**Log**
Bases: `object`

Methods

display_all()

display_value(key, value)

ensure_directory_exists()

get_time_code()

reset()

set(key, value)

use_directory(dir)

modeci_mdf.functions.actr.ccm.logger.LogProxy

class modeci_mdf.functions.actr.ccm.logger.**LogProxy** (*log, prefix=""*)
Bases: `object`

Methods

—

modeci_mdf.functions.actr.ccm.logger.Trace

class modeci_mdf.functions.actr.ccm.logger.**Trace**
Bases: `object`

Methods

`add(key, value)`

`fixed_keys()`

`get_at(name, time)`

`get_final(key)`

`get_pts(vars)`

`group_pts(pts, key)`

`keys()`

`merge_pts(pts, key)`

modeci_mdf.functions.actr.ccm.model**Functions**

`log_everything(model[, log])`

modeci_mdf.functions.actr.ccm.model.log_everything

modeci_mdf.functions.actr.ccm.model.**log_everything** (*model*, *log=None*)

Classes

`MethodGeneratorWrapper`(obj, func, name)

`MethodWrapper`(obj, func, name)

`Model`([log])

modeci_mdf.functions.actr.ccm.model.MethodGeneratorWrapper

```
class modeci_mdf.functions.actr.ccm.model.MethodGeneratorWrapper (obj, func,  
                                                                name)  
    Bases: modeci_mdf.functions.actr.ccm.model.MethodWrapper
```

Methods

```
__call__ (*args, **keys)  
    Call self as a function.
```

modeci_mdf.functions.actr.ccm.model.MethodWrapper

```
class modeci_mdf.functions.actr.ccm.model.MethodWrapper (obj, func, name)  
    Bases: object
```

Methods

```
__call__ (*args, **keys)  
    Call self as a function.
```

modeci_mdf.functions.actr.ccm.model.Model

```
class modeci_mdf.functions.actr.ccm.model.Model (log=None, **keys)  
    Bases: object
```

Methods

```
get_children()
```

```
now()
```

```
run([limit, func])
```

```
start()
```

```
stop()
```

modeci_mdf.functions.actr.ccm.pattern

Functions

get(obj, name, key)

parse(patterns[, bound])

partialmatch(obj, name, key, b, value)

modeci_mdf.functions.actr.ccm.pattern.get

modeci_mdf.functions.actr.ccm.pattern.**get** (*obj, name, key*)

modeci_mdf.functions.actr.ccm.pattern.parse

modeci_mdf.functions.actr.ccm.pattern.**parse** (*patterns, bound=None*)

modeci_mdf.functions.actr.ccm.pattern.partialmatch

modeci_mdf.functions.actr.ccm.pattern.**partialmatch** (*obj, name, key, b, value*)

Classes

Pattern(patterns[, bound, partial])

modeci_mdf.functions.actr.ccm.pattern.Pattern

class modeci_mdf.functions.actr.ccm.pattern.**Pattern** (*patterns, bound=None, partial=None*)

Bases: *object*

Methods

match(obj)

Exceptions

PatternException

modeci_mdf.functions.actr.ccm.pattern.PatternException

exception modeci_mdf.functions.actr.ccm.pattern.**PatternException**

modeci_mdf.functions.actr.ccm.scheduler

Classes

Event(func, time[, args, keys, priority])

Scheduler()

Trigger([name])

modeci_mdf.functions.actr.ccm.scheduler.Event

class modeci_mdf.functions.actr.ccm.scheduler.**Event** (*func, time, args=[], keys={}, priority=0*)
Bases: *object*

Methods

modeci_mdf.functions.actr.ccm.scheduler.Scheduler

class modeci_mdf.functions.actr.ccm.scheduler.**Scheduler**
Bases: *object*

Methods

add(func[, delay, args, keys, priority, ...])

add_event(event)

do_event(event)

extend(other)

continues on next page

Table 50 – continued from previous page

<code>handle_result(result, event)</code>
<code>run()</code>
<code>stop()</code>
<code>trigger(key[, priority])</code>

modeci_mdf.functions.actr.ccm.scheduler.Trigger

class modeci_mdf.functions.actr.ccm.scheduler.**Trigger** (*name=""*)
 Bases: `object`

Methods

—

Exceptions

SchedulerError

modeci_mdf.functions.actr.ccm.scheduler.SchedulerError

exception modeci_mdf.functions.actr.ccm.scheduler.**SchedulerError**

25.3.2 modeci_mdf.functions.onnx

Programmatically defines every ONNX operation as a python callable function. Executing ONNX graphs in this way somewhat defeats the performance purposes of ONNX since the overhead for each operation will be high. However, this allows us to test the MDF scheduler (which invokes Python functions) on any MDF model defined over ONNX operations. In the future, the MDF should probably just compile to ONNX (or some other IR) for execution.

Functions

<code>abs(*args, **kwargs)</code>	Absolute takes one input data (Tensor<T>) and produces one output data (Tensor<T>) where the absolute is, $y = \text{abs}(x)$, is applied to the tensor elementwise.
<code>acos(*args, **kwargs)</code>	Calculates the arccosine (inverse of cosine) of the given input tensor, element-wise.
<code>acosh(*args, **kwargs)</code>	Calculates the hyperbolic arccosine of the given input tensor element-wise.
<code>add(*args, **kwargs)</code>	Performs element-wise binary addition (with Numpy-style broadcasting support).

continues on next page

Table 53 – continued from previous page

<code>and(*args, **kwargs)</code>	Returns the tensor resulted from performing the <i>and</i> logical operation elementwise on the input tensors <i>A</i> and <i>B</i> (with Numpy-style broadcasting support).
<code>argmax(*args, **kwargs)</code>	Computes the indices of the max elements of the input tensor's element along the provided axis.
<code>argmin(*args, **kwargs)</code>	Computes the indices of the min elements of the input tensor's element along the provided axis.
<code>asin(*args, **kwargs)</code>	Calculates the arcsine (inverse of sine) of the given input tensor, element-wise.
<code>asinh(*args, **kwargs)</code>	Calculates the hyperbolic arcsine of the given input tensor element-wise.
<code>atan(*args, **kwargs)</code>	Calculates the arctangent (inverse of tangent) of the given input tensor, element-wise.
<code>atanh(*args, **kwargs)</code>	Calculates the hyperbolic arctangent of the given input tensor element-wise.
<code>averagepool(*args, **kwargs)</code>	AveragePool consumes an input tensor <i>X</i> and applies average pooling across the tensor according to kernel sizes, stride sizes, and pad lengths.
<code>batchnormalization(*args, **kwargs)</code>	Carries out batch normalization as described in the paper https://arxiv.org/abs/1502.03167 .
<code>bernoulli(*args, **kwargs)</code>	Draws binary random numbers (0 or 1) from a Bernoulli distribution.
<code>bitshift(*args, **kwargs)</code>	Bitwise shift operator performs element-wise operation. For each input element, if the
<code>cast(*args, **kwargs)</code>	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.
<code>castlike(*args, **kwargs)</code>	The operator casts the elements of a given input tensor (the first input) to the same data type as the elements of the second input tensor.
<code>ceil(*args, **kwargs)</code>	Ceil takes one input data (Tensor<T>) and produces one output data (Tensor<T>) where the ceil is, $y = \text{ceil}(x)$, is applied to the tensor elementwise.
<code>celu(*args, **kwargs)</code>	Continuously Differentiable Exponential Linear Units: Perform the linear unit element-wise on the input tensor <i>X</i> using formula:
<code>clip(*args, **kwargs)</code>	Clip operator limits the given input within an interval.
<code>compress(*args, **kwargs)</code>	Selects slices from an input tensor along a given axis where condition evaluates to True for each axis index.
<code>concat(*args, **kwargs)</code>	Concatenate a list of tensors into a single tensor.
<code>concatfromsequence(*args, **kwargs)</code>	Concatenate a sequence of tensors into a single tensor.
<code>constant(*args, **kwargs)</code>	This operator produces a constant tensor.
<code>constantofshape(*args, **kwargs)</code>	Generate a tensor with given value and shape.
<code>conv(*args, **kwargs)</code>	The convolution operator consumes an input tensor and a filter, and computes the output.
<code>convert_type(v)</code>	Helper function to convert types to ONNX compatible types.
<code>convinteger(*args, **kwargs)</code>	The integer convolution operator consumes an input tensor, its zero-point, a filter, and its zero-point, and computes the output.

continues on next page

Table 53 – continued from previous page

<code>convttranspose(*args, **kwargs)</code>	The convolution transpose operator consumes an input tensor and a filter, and computes the output.
<code>cos(*args, **kwargs)</code>	Calculates the cosine of the given input tensor, element-wise.
<code>cosh(*args, **kwargs)</code>	Calculates the hyperbolic cosine of the given input tensor element-wise.
<code>cumsum(*args, **kwargs)</code>	Performs cumulative sum of the input elements along the given axis.
<code>depthtospace(*args, **kwargs)</code>	DepthToSpace rearranges (permutes) data from depth into blocks of spatial data.
<code>dequantizelinear(*args, **kwargs)</code>	The linear dequantization operator.
<code>det(*args, **kwargs)</code>	Det calculates determinant of a square matrix or batches of square matrices.
<code>div(*args, **kwargs)</code>	Performs element-wise binary division (with Numpy-style broadcasting support).
<code>dropout(*args, **kwargs)</code>	Dropout takes an input floating-point tensor, an optional input ratio (floating-point scalar) and an optional input training_mode (boolean scalar).
<code>dynamicquantizelinear(*args, **kwargs)</code>	A Function to fuse calculation for Scale, Zero Point and FP32->8Bit conversion of FP32 Input data. Outputs Scale, ZeroPoint and Quantized Input for a given FP32 Input. Scale is calculated as: $y_scale = (max(x) - min(x)) / (qmax - qmin)$ * 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: $intermediate_zero_point = qmin - min(x) / y_scale$ $y_zero_point = cast(round(saturate(intermediate_zero_point)))$ * where qmax and qmin are max and min values for quantization range .i.e [0, 255] in case of uint8 * for saturation, it saturates to [0, 255] if it's uint8, or [-127, 127] if it's int8. Right now only uint8 is supported. * rounding to nearest ties to even. `Data quantization formula is: $y = saturate(round(x / y_scale) + y_zero_point)$ * for saturation, it saturates to [0, 255] if it's uint8, or [-127, 127] if it's int8. Right now only uint8 is supported. * rounding to nearest ties to even. `.
<code>einsum(*args, **kwargs)</code>	An einsum of the form `term1, term2 -> output-term` produces an output tensor using the following equation
<code>elu(*args, **kwargs)</code>	Elu takes one input data (Tensor<T>) and produces one output data (Tensor<T>) where the function $f(x) = \alpha * (exp(x) - 1.)$ for $x < 0$, $f(x) = x$ for $x \geq 0$., is applied to the tensor elementwise.

continues on next page

Table 53 – continued from previous page

<code>equal(*args, **kwargs)</code>	Returns the tensor resulted from performing the <i>equal</i> logical operation elementwise on the input tensors <i>A</i> and <i>B</i> (with Numpy-style broadcasting support).
<code>erf(*args, **kwargs)</code>	Computes the error function of the given input tensor element-wise.
<code>exp(*args, **kwargs)</code>	Calculates the exponential of the given input tensor, element-wise.
<code>expand(*args, **kwargs)</code>	Broadcast the input tensor following the given shape and the broadcast rule.
<code>eyelike(*args, **kwargs)</code>	Generate a 2D tensor (matrix) with ones on the diagonal and zeros everywhere else.
<code>flatten(*args, **kwargs)</code>	Flattens the input tensor into a 2D matrix.
<code>floor(*args, **kwargs)</code>	Floor takes one input data (Tensor<T>) and produces one output data (Tensor<T>) where the floor is, $y = \text{floor}(x)$, is applied to the tensor elementwise.
<code>gather(*args, **kwargs)</code>	Given <i>data</i> tensor of rank $r \geq 1$, and <i>indices</i> tensor of rank q , gather entries of the axis dimension of <i>data</i> (by default outer-most one as $\text{axis}=0$) indexed by <i>indices</i> , and concatenates them in an output tensor of rank $q + (r - 1)$.
<code>gatherelements(*args, **kwargs)</code>	GatherElements takes two inputs <i>data</i> and <i>indices</i> of the same rank $r \geq 1$ and an optional attribute <i>axis</i> that identifies an axis of <i>data</i> (by default, the outer-most axis, that is axis 0).
<code>gathernd(*args, **kwargs)</code>	Given <i>data</i> tensor of rank $r \geq 1$, <i>indices</i> tensor of rank $q \geq 1$, and <i>batch_dims</i> integer b , this operator gathers slices of <i>data</i> into an output tensor of rank $q + r - \text{indices_shape}[-1] - 1 - b$.
<code>gemm(*args, **kwargs)</code>	General Matrix multiplication: https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms#Level_3
<code>get_all_schemas_version(max_version)</code>	Enumerate all the OpSchemas available from ONNX.
<code>get_onnx_ops([opset_version])</code>	Enumerate all available ONNX operations and generate MDF function specifications for each one.
<code>get_onnx_schema(func_name[, opset_version])</code>	Return the ONNX schema corresponding to a generated ONNX python function with name func_name
<code>globalaveragepool(*args, **kwargs)</code>	GlobalAveragePool consumes an input tensor <i>X</i> and applies average pooling across the values in the same channel.
<code>globallppool(*args, **kwargs)</code>	GlobalLpPool consumes an input tensor <i>X</i> and applies lp pool pooling across the values in the same channel.
<code>globalmaxpool(*args, **kwargs)</code>	GlobalMaxPool consumes an input tensor <i>X</i> and applies max pooling across the values in the same channel.
<code>greater(*args, **kwargs)</code>	Returns the tensor resulted from performing the <i>greater</i> logical operation elementwise on the input tensors <i>A</i> and <i>B</i> (with Numpy-style broadcasting support).
<code>greaterorequal(*args, **kwargs)</code>	Returns the tensor resulted from performing the <i>greater_equal</i> logical operation elementwise on the input tensors <i>A</i> and <i>B</i> (with Numpy-style broadcasting support).
<code>gru(*args, **kwargs)</code>	Computes an one-layer GRU.

continues on next page

Table 53 – continued from previous page

<i>hardmax</i> (*args, **kwargs)	The operator computes the hardmax values for the given input:
<i>hardsigmoid</i> (*args, **kwargs)	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.
<i>hardswish</i> (*args, **kwargs)	HardSwish takes one input data (Tensor<T>) and produces one output data (Tensor<T>) where the HardSwish function, $y = x * \max(0, \min(1, \alpha * x + \beta)) = x * \text{HardSigmoid}(\alpha, \beta)(x)$, where $\alpha = 1/6$ and $\beta = 0.5$, is applied to the tensor elementwise.
<i>identity</i> (*args, **kwargs)	Identity operator
<i>if</i> (*args, **kwargs)	If conditional
<i>import_class</i> (name)	Import from a module specified by a string
<i>instancenormalization</i> (*args, **kwargs)	Carries out instance normalization as described in the paper https://arxiv.org/abs/1607.08022 .
<i>isinf</i> (*args, **kwargs)	Map infinity to true and other values to false.
<i>isnan</i> (*args, **kwargs)	Returns which elements of the input are NaN.
<i>leakyrelu</i> (*args, **kwargs)	LeakyRelu takes input data (Tensor<T>) and an argument α , and produces one output data (Tensor<T>) where the function $f(x) = \alpha * x$ for $x < 0$, $f(x) = x$ for $x \geq 0$, is applied to the data tensor elementwise.
<i>less</i> (*args, **kwargs)	Returns the tensor resulted from performing the <i>less</i> logical operation elementwise on the input tensors <i>A</i> and <i>B</i> (with Numpy-style broadcasting support).
<i>lessorequal</i> (*args, **kwargs)	Returns the tensor resulted from performing the <i>less_equal</i> logical operation elementwise on the input tensors <i>A</i> and <i>B</i> (with Numpy-style broadcasting support).
<i>log</i> (*args, **kwargs)	Calculates the natural log of the given input tensor, element-wise.
<i>logsoftmax</i> (*args, **kwargs)	The operator computes the log of softmax values for the given input:
<i>loop</i> (*args, **kwargs)	Generic Looping construct.
<i>lpnormalization</i> (*args, **kwargs)	Given a matrix, apply Lp-normalization along the provided axis.
<i>lppool</i> (*args, **kwargs)	LpPool consumes an input tensor <i>X</i> and applies Lp pooling across the tensor according to kernel sizes, stride sizes, and pad lengths.
<i>lrn</i> (*args, **kwargs)	Local Response Normalization proposed in the [AlexNet paper](https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf).
<i>lstm</i> (*args, **kwargs)	Computes an one-layer LSTM.
<i>matmul</i> (*args, **kwargs)	Matrix product that behaves like <code>numpy.matmul</code> : https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.matmul.html
<i>matmulinteger</i> (*args, **kwargs)	Matrix product that behaves like <code>numpy.matmul</code> : https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.matmul.html .

continues on next page

Table 53 – continued from previous page

<i>max</i> (*args, **kwargs)	Element-wise max of each of the input tensors (with Numpy-style broadcasting support).
<i>maxpool</i> (*args, **kwargs)	MaxPool consumes an input tensor X and applies max pooling across the tensor according to kernel sizes, stride sizes, and pad lengths.
<i>maxroipool</i> (*args, **kwargs)	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]).
<i>maxunpool</i> (*args, **kwargs)	MaxUnpool essentially computes the partial inverse of the MaxPool op.
<i>mean</i> (*args, **kwargs)	Element-wise mean of each of the input tensors (with Numpy-style broadcasting support).
<i>meanvariancenormalization</i> (*args, **kwargs)	A MeanVarianceNormalization Function: Perform mean variance normalization on the input tensor X using formula: $\frac{(X - EX)}{\sqrt{E(X - EX)^2}}$
<i>min</i> (*args, **kwargs)	Element-wise min of each of the input tensors (with Numpy-style broadcasting support).
<i>mod</i> (*args, **kwargs)	Performs element-wise binary modulus (with Numpy-style broadcasting support).
<i>mul</i> (*args, **kwargs)	Performs element-wise binary multiplication (with Numpy-style broadcasting support).
<i>multinomial</i> (*args, **kwargs)	Generate a tensor of samples from a multinomial distribution according to the probabilities of each of the possible outcomes.
<i>neg</i> (*args, **kwargs)	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.
<i>negativeloglikelihoodloss</i> (*args, **kwargs)	A NegativeLogLikelihoodLoss operator computes (weighted) negative log likelihood loss.
<i>nonmaxsuppression</i> (*args, **kwargs)	Filter out boxes that have high intersection-over-union (IOU) overlap with previously selected boxes.
<i>nonzero</i> (*args, **kwargs)	Returns the indices of the elements that are non-zero (in row-major order - by dimension).
<i>not</i> (*args, **kwargs)	Returns the negation of the input tensor element-wise.
<i>onehot</i> (*args, **kwargs)	Produces a one-hot tensor based on inputs.
<i>optional</i> (*args, **kwargs)	Constructs an optional-type value containing either an empty optional of a certain type specified by the attribute, or a non-empty value containing the input element.
<i>optionalgetelement</i> (*args, **kwargs)	Outputs the element in the optional-type input.
<i>optionalhaselement</i> (*args, **kwargs)	Returns true if the optional-type input contains an element.
<i>or</i> (*args, **kwargs)	Returns the tensor resulted from performing the <i>or</i> logical operation elementwise on the input tensors <i>A</i> and <i>B</i> (with Numpy-style broadcasting support).
<i>pad</i> (*args, **kwargs)	Given a tensor containing the data to be padded (<i>data</i>), a tensor containing the number of start and end pad values for axis (<i>pads</i>), (optionally) a <i>mode</i> , and (optionally) <i>constant_value</i> , a padded tensor (<i>output</i>) is generated.

continues on next page

Table 53 – continued from previous page

<code>pow(*args, **kwargs)</code>	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.
<code>predict_with_onnxruntime(model_def, *inputs)</code>	Simple helper to run an ONNX model with a set of inputs.
<code>prelu(*args, **kwargs)</code>	PRelu takes input data (Tensor<T>) and slope tensor as input, and produces one output data (Tensor<T>) where the function $f(x) = slope * x \text{ for } x < 0, f(x) = x \text{ for } x \geq 0.$, is applied to the data tensor elementwise.
<code>qlinearconv(*args, **kwargs)</code>	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.
<code>qlinearmatmul(*args, **kwargs)</code>	Matrix product that behaves like <code>numpy.matmul</code> : https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.matmul.html .
<code>quantizelinear(*args, **kwargs)</code>	The linear quantization operator.
<code>randomnormal(*args, **kwargs)</code>	Generate a tensor with random values drawn from a normal distribution.
<code>randomnormallike(*args, **kwargs)</code>	Generate a tensor with random values drawn from a normal distribution.
<code>randomuniform(*args, **kwargs)</code>	Generate a tensor with random values drawn from a uniform distribution.
<code>randomuniformlike(*args, **kwargs)</code>	Generate a tensor with random values drawn from a uniform distribution.
<code>range(*args, **kwargs)</code>	Generate a tensor containing a sequence of numbers that begin at <i>start</i> and extends by increments of <i>delta</i> up to <i>limit</i> (exclusive).
<code>reciprocal(*args, **kwargs)</code>	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.
<code>reducel1(*args, **kwargs)</code>	Computes the L1 norm of the input tensor's element along the provided axes.
<code>reducel2(*args, **kwargs)</code>	Computes the L2 norm of the input tensor's element along the provided axes.
<code>reducelogsum(*args, **kwargs)</code>	Computes the log sum of the input tensor's element along the provided axes.
<code>reducelogsumexp(*args, **kwargs)</code>	Computes the log sum exponent of the input tensor's element along the provided axes.
<code>reducemax(*args, **kwargs)</code>	Computes the max of the input tensor's element along the provided axes.
<code>reducemean(*args, **kwargs)</code>	Computes the mean of the input tensor's element along the provided axes.
<code>reducemin(*args, **kwargs)</code>	Computes the min of the input tensor's element along the provided axes.
<code>reduceprod(*args, **kwargs)</code>	Computes the product of the input tensor's element along the provided axes.
<code>reducesum(*args, **kwargs)</code>	Computes the sum of the input tensor's element along the provided axes.

continues on next page

Table 53 – continued from previous page

<code>reducesumsquare(*args, **kwargs)</code>	Computes the sum square of the input tensor's element along the provided axes.
<code>relu(*args, **kwargs)</code>	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 element-wise.
<code>reshape(*args, **kwargs)</code>	Reshape the input tensor similar to <code>numpy.reshape</code> .
<code>resize(*args, **kwargs)</code>	Resize the input tensor. In general, it calculates every value in the output tensor as a weighted average of neighborhood (a.k.a. sampling locations) in the input tensor. Each dimension value of the output tensor is: $\text{output_dimension} = \text{floor}(\text{input_dimension} * (\text{roi_end} - \text{roi_start}) * \text{scale})$ if input "sizes" is not specified.
<code>reversesequence(*args, **kwargs)</code>	Reverse batch of sequences having different lengths specified by <code>sequence_lens</code> .
<code>rnn(*args, **kwargs)</code>	Computes an one-layer simple RNN.
<code>roialign(*args, **kwargs)</code>	Region of Interest (RoI) align operation described in the [Mask R-CNN paper](https://arxiv.org/abs/1703.06870).
<code>round(*args, **kwargs)</code>	Round takes one input Tensor and rounds the values, element-wise, meaning it finds the nearest integer for each value.
<code>run_onnx_op(op_name, inputs, output_names[, ...])</code>	Simple helper function that invokes a single ONNX operator with inputs and attributes and returns the results.
<code>scan(*args, **kwargs)</code>	Scan can be used to iterate over one or more scan_input tensors, constructing zero or more scan_output tensors.
<code>scatter(*args, **kwargs)</code>	This operator is deprecated.
<code>scatterelements(*args, **kwargs)</code>	ScatterElements takes three inputs <i>data</i> , <i>updates</i> , and <i>indices</i> of the same rank $r \geq 1$ and an optional attribute axis that identifies an axis of <i>data</i> (by default, the outermost axis, that is axis 0).
<code>scatternd(*args, **kwargs)</code>	ScatterND takes three inputs <i>data</i> tensor of rank $r \geq 1$, <i>indices</i> tensor of rank $q \geq 1$, and <i>updates</i> tensor of rank $q + r - \text{indices.shape}[-1] - 1$.
<code>selu(*args, **kwargs)</code>	Selu takes one input data (Tensor<T>) and produces one output data (Tensor<T>) where the scaled exponential linear unit function, $y = \text{gamma} * (\text{alpha} * e^x - \text{alpha})$ for $x \leq 0$, $y = \text{gamma} * x$ for $x > 0$, is applied to the tensor elementwise.
<code>sequenceat(*args, **kwargs)</code>	Outputs a tensor copy from the tensor at 'position' in 'input_sequence'.
<code>sequenceconstruct(*args, **kwargs)</code>	Construct a tensor sequence containing 'inputs' tensors.
<code>sequenceempty(*args, **kwargs)</code>	Construct an empty tensor sequence, with given data type.
<code>sequenceerase(*args, **kwargs)</code>	Outputs a tensor sequence that removes the tensor at 'position' from 'input_sequence'.
<code>sequenceinsert(*args, **kwargs)</code>	Outputs a tensor sequence that inserts 'tensor' into 'input_sequence' at 'position'.
<code>sequencelength(*args, **kwargs)</code>	Produces a scalar(tensor of empty shape) containing the number of tensors in 'input_sequence'.

continues on next page

Table 53 – continued from previous page

<i>shape</i> (*args, **kwargs)	Takes a tensor as input and outputs an 1D int64 tensor containing the shape of the input tensor.
<i>shrink</i> (*args, **kwargs)	Shrink takes one input data (Tensor<numeric>) and produces one Tensor output, having same datatype and shape with input.
<i>sigmoid</i> (*args, **kwargs)	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.
<i>sign</i> (*args, **kwargs)	Calculate the sign of the given input tensor element-wise.
<i>sin</i> (*args, **kwargs)	Calculates the sine of the given input tensor, element-wise.
<i>sinh</i> (*args, **kwargs)	Calculates the hyperbolic sine of the given input tensor element-wise.
<i>size</i> (*args, **kwargs)	Takes a tensor as input and outputs a int64 scalar that equals to the total number of elements of the input tensor.
<i>slice</i> (*args, **kwargs)	Produces a slice of the input tensor along multiple axes.
<i>softmax</i> (*args, **kwargs)	The operator computes the normalized exponential values for the given input:
<i>softmaxcrossentropyloss</i> (*args, **kwargs)	Loss function that measures the softmax cross entropy between ‘scores’ and ‘labels’.
<i>softplus</i> (*args, **kwargs)	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 element-wise.
<i>softsign</i> (*args, **kwargs)	Calculates the softsign ($x/(1+ x)$) of the given input tensor element-wise.
<i>spacetodepth</i> (*args, **kwargs)	SpaceToDepth rearranges blocks of spatial data into depth.
<i>split</i> (*args, **kwargs)	Split a tensor into a list of tensors, along the specified ‘axis’.
<i>splittosequence</i> (*args, **kwargs)	Split a tensor into a sequence of tensors, along the specified ‘axis’.
<i>sqrt</i> (*args, **kwargs)	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.
<i>squeeze</i> (*args, **kwargs)	Remove single-dimensional entries from the shape of a tensor.
<i>stringnormalizer</i> (*args, **kwargs)	StringNormalization performs string operations for basic cleaning.
<i>sub</i> (*args, **kwargs)	Performs element-wise binary subtraction (with Numpy-style broadcasting support).
<i>sum</i> (*args, **kwargs)	Element-wise sum of each of the input tensors (with Numpy-style broadcasting support).
<i>tan</i> (*args, **kwargs)	Calculates the tangent of the given input tensor, element-wise.
<i>tanh</i> (*args, **kwargs)	Calculates the hyperbolic tangent of the given input tensor element-wise.

continues on next page

Table 53 – continued from previous page

<i>tfidfvectorizer</i> (*args, **kwargs)	This transform extracts n-grams from the input sequence and save them as a vector.
<i>thresholdedrelu</i> (*args, **kwargs)	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.
<i>tile</i> (*args, **kwargs)	Constructs a tensor by tiling a given tensor.
<i>topk</i> (*args, **kwargs)	Retrieve the top-K largest or smallest elements along a specified axis. Given an input tensor of shape $[a_1, a_2, \dots, a_n, r]$ and integer argument k , return two outputs: -Value tensor of shape $[a_1, a_2, \dots, a_{\{axis-1\}}, k, a_{\{axis+1\}}, \dots, a_n]$ which contains the values of the top k elements along the specified axis -Index tensor of shape $[a_1, a_2, \dots, a_{\{axis-1\}}, k, a_{\{axis+1\}}, \dots, a_n]$ which contains the indices of the top k elements (original indices from the input tensor).
<i>transpose</i> (*args, **kwargs)	Transpose the input tensor similar to <code>numpy.transpose</code> .
<i>trilu</i> (*args, **kwargs)	Given a 2-D matrix or batches of 2-D matrices, returns the upper or lower triangular part of the tensor(s). The attribute “upper” determines whether the upper or lower part is retained. If set to true, the upper triangular matrix is retained. Lower triangular matrix is retained otherwise. Default value for the “upper” attribute is true. Trilu takes one input tensor of shape $[*, N, M]$, where $*$ is zero or more batch dimensions. The upper triangular part consists of the elements on and above the given diagonal (k). The lower triangular part consists of elements on and below the diagonal. All other elements in the matrix are set to zero. If $k = 0$, the triangular part on and above/below the main diagonal is retained. If upper is set to true, a positive k retains the upper triangular matrix excluding the main diagonal and $(k-1)$ diagonals above it. A negative k value retains the main diagonal and $ k $ diagonals below it. If upper is set to false, a positive k retains the lower triangular matrix including the main diagonal and k diagonals above it. A negative k value excludes the main diagonal and $(k -1)$ diagonals below it.
<i>unique</i> (*args, **kwargs)	Find the unique elements of a tensor.
<i>unsqueeze</i> (*args, **kwargs)	Insert single-dimensional entries to the shape of an input tensor (<i>data</i>).
<i>upsample</i> (*args, **kwargs)	Upsample the input tensor. Each dimension value of the output tensor is: $\text{output_dimension} = \text{floor}(\text{input_dimension} * \text{scale})$.
<i>where</i> (*args, **kwargs)	Return elements, either from X or Y, depending on condition.
<i>xor</i> (*args, **kwargs)	Returns the tensor resulted from performing the <i>xor</i> logical operation elementwise on the input tensors <i>A</i> and <i>B</i> (with Numpy-style broadcasting support).

modeci_mdf.functions.onnx.abs

```
modeci_mdf.functions.onnx.abs(*args, **kwargs)
```

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

modeci_mdf.functions.onnx.acos

```
modeci_mdf.functions.onnx.acos(*args, **kwargs)
```

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

modeci_mdf.functions.onnx.acosh

```
modeci_mdf.functions.onnx.acosh(*args, **kwargs)
```

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

modeci_mdf.functions.onnx.add

```
modeci_mdf.functions.onnx.add(*args, **kwargs)
```

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).

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

modeci_mdf.functions.onnx.and

```
modeci_mdf.functions.onnx.and(*args, **kwargs)
```

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).

modeci_mdf.functions.onnx.argmax

```
modeci_mdf.functions.onnx.argmax(*args, **kwargs)
```

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

modeci_mdf.functions.onnx.argmaxin

`modeci_mdf.functions.onnx.argmaxin(*args, **kwargs)`

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

modeci_mdf.functions.onnx.asin

`modeci_mdf.functions.onnx.asin(*args, **kwargs)`

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

modeci_mdf.functions.onnx.asinh

`modeci_mdf.functions.onnx.asinh(*args, **kwargs)`

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

modeci_mdf.functions.onnx.atan

`modeci_mdf.functions.onnx.atan(*args, **kwargs)`

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

modeci_mdf.functions.onnx.atanh

`modeci_mdf.functions.onnx.atanh(*args, **kwargs)`

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

modeci_mdf.functions.onnx.averagepool

`modeci_mdf.functions.onnx.averagepool(*args, **kwargs)`

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).
```

modeci_mdf.functions.onnx.batchnormalization

`modeci_mdf.functions.onnx.batchnormalization(*args, **kwargs)`

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

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

When `training_mode=False`, extra outputs are invalid. The outputs are updated as follows when `training_mode=True`:
`running_mean = input_mean * momentum + current_mean * (1 - momentum)`
`running_var = input_var * momentum + current_var * (1 - momentum)`

$$Y = (X - \text{current_mean}) / \sqrt{\text{current_var} + \text{epsilon}} * \text{scale} + B$$

where:

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

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

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

When `training_mode=False`:

$$Y = (X - \text{input_mean}) / \sqrt{\text{input_var} + \text{epsilon}} * \text{scale} + B$$

For previous (deprecated) non-spatial cases, implementors are suggested to flatten the input shape to $(N \times C \times D1 \times D2 \times \dots \times Dn)$ before a `BatchNormalization` Op. This operator has **optional** inputs/outputs. See [the doc](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.

modeci_mdf.functions.onnx.bernoulli

`modeci_mdf.functions.onnx.bernoulli(*args, **kwargs)`

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

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

modeci_mdf.functions.onnx.bitshift

`modeci_mdf.functions.onnx.bitshift(*args, **kwargs)`

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).

modeci_mdf.functions.onnx.cast

`modeci_mdf.functions.onnx.cast(*args, **kwargs)`

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.

In more detail, the conversion among numerical types should follow these rules:

- Casting from floating point to: * floating point: +/- infinity if OOR (out of range). * fixed point: undefined if OOR. * bool: +/- 0.0 to False; all else to True.
- Casting from fixed point to: * floating point: +/- infinity if OOR. (+ infinity in the case of uint) * fixed point: when OOR, discard higher bits and reinterpret (with respect to two’s complement representation for

signed types). For example, 200 (int16) -> -56 (int8).

- bool: zero to False; nonzero to True.
- Casting from bool to: * floating point: {1.0, 0.0}. * fixed point: {1, 0}. * bool: no change.

modeci_mdf.functions.onnx.castlike

`modeci_mdf.functions.onnx.castlike(*args, **kwargs)`

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

modeci_mdf.functions.onnx.ceil

`modeci_mdf.functions.onnx.ceil(*args, **kwargs)`

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

modeci_mdf.functions.onnx.celu

`modeci_mdf.functions.onnx.celu(*args, **kwargs)`

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

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

modeci_mdf.functions.onnx.clip

`modeci_mdf.functions.onnx.clip(*args, **kwargs)`

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.

modeci_mdf.functions.onnx.compress

`modeci_mdf.functions.onnx.compress(*args, **kwargs)`

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>

modeci_mdf.functions.onnx.concat

`modeci_mdf.functions.onnx.concat(*args, **kwargs)`

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.

modeci_mdf.functions.onnx.concatfromsequence

`modeci_mdf.functions.onnx.concatfromsequence(*args, **kwargs)`

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

modeci_mdf.functions.onnx.constant

`modeci_mdf.functions.onnx.constant(*args, **kwargs)`

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

modeci_mdf.functions.onnx.constantofshape

`modeci_mdf.functions.onnx.constantofshape(*args, **kwargs)`
Generate a tensor with given value and shape.

modeci_mdf.functions.onnx.conv

`modeci_mdf.functions.onnx.conv(*args, **kwargs)`
The convolution operator consumes an input tensor and a filter, and computes the output.

modeci_mdf.functions.onnx.convert_type

`modeci_mdf.functions.onnx.convert_type(v)`
Helper function to convert types to ONNX compatible types.

modeci_mdf.functions.onnx.convinteger

`modeci_mdf.functions.onnx.convinteger(*args, **kwargs)`
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.

modeci_mdf.functions.onnx.convtranspose

`modeci_mdf.functions.onnx.convtranspose(*args, **kwargs)`
The convolution transpose operator consumes an input tensor and a filter, and computes the output.

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

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

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

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

modeci_mdf.functions.onnx.cos

`modeci_mdf.functions.onnx.cos(*args, **kwargs)`
Calculates the cosine of the given input tensor, element-wise.

modeci_mdf.functions.onnx.cosh

`modeci_mdf.functions.onnx.cosh(*args, **kwargs)`

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

modeci_mdf.functions.onnx.cumsum

`modeci_mdf.functions.onnx.cumsum(*args, **kwargs)`

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] ``

modeci_mdf.functions.onnx.depthtospace

`modeci_mdf.functions.onnx.depthtospace(*args, **kwargs)`

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])
```

modeci_mdf.functions.onnx.dequantizelinear

`modeci_mdf.functions.onnx.dequantizelinear(*args, **kwargs)`

The linear dequantization operator. It consumes a quantized tensor, a scale, and a zero point to compute the full precision tensor. The dequantization formula is $y = (x - x_zero_point) * x_scale$. 'x_scale' and 'x_zero_point' must have same shape, and can be either a scalar for per-tensor / per layer quantization, or a 1-D tensor for per-axis quantization. '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).

modeci_mdf.functions.onnx.det

`modeci_mdf.functions.onnx.det(*args, **kwargs)`

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: []).

modeci_mdf.functions.onnx.div

`modeci_mdf.functions.onnx.div(*args, **kwargs)`

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).

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

modeci_mdf.functions.onnx.dropout

`modeci_mdf.functions.onnx.dropout(*args, **kwargs)`

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.

modeci_mdf.functions.onnx.dynamicquantizelinear

`modeci_mdf.functions.onnx.dynamicquantizelinear(*args, **kwargs)`

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

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

modeci_mdf.functions.onnx.einsum

`modeci_mdf.functions.onnx.einsum(*args, **kwargs)`

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 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.

modeci_mdf.functions.onnx.elu

`modeci_mdf.functions.onnx.elu(*args, **kwargs)`

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

modeci_mdf.functions.onnx.equal

`modeci_mdf.functions.onnx.equal(*args, **kwargs)`

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).

modeci_mdf.functions.onnx.erf

`modeci_mdf.functions.onnx.erf(*args, **kwargs)`

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

modeci_mdf.functions.onnx.exp

`modeci_mdf.functions.onnx.exp(*args, **kwargs)`

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

modeci_mdf.functions.onnx.expand

`modeci_mdf.functions.onnx.expand(*args, **kwargs)`

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

modeci_mdf.functions.onnx.eyelike

`modeci_mdf.functions.onnx.eyelike(*args, **kwargs)`

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.

modeci_mdf.functions.onnx.flatten

`modeci_mdf.functions.onnx.flatten(*args, **kwargs)`

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

modeci_mdf.functions.onnx.floor

`modeci_mdf.functions.onnx.floor(*args, **kwargs)`

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

modeci_mdf.functions.onnx.gather

`modeci_mdf.functions.onnx.gather(*args, **kwargs)`

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

axis = 0 :

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

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

```

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


axis = 1 :

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


```

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

```


```

### modeci\_mdf.functions.onnx.gatherelements

`modeci_mdf.functions.onnx.gatherelements(*args, **kwargs)`

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

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

```

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

```

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

Example 1: ```

```

data = [ [ 1, 2], [3, 4],
] indices = [
    [0, 0], [1, 0],
] axis = 1 output = [
    [1, 1], [4, 3],
]
```

```

```

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

```

### modeci\_mdf.functions.onnx.gathernd

`modeci_mdf.functions.onnx.gathernd(*args, **kwargs)`

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

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

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

Some salient points about the inputs' rank and shape:

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

The output is computed as follows:

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

- 1) If *indices\_shape* $[-1] > r-b \Rightarrow$  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* $[0:b-1, \text{indices\_slice}]$  is filled into the corresponding location of the  $(q-b-1)$ -dimensional tensor to form the *output* tensor (Example 1 below)
- 3) If *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* $[0:b-1, \text{indices\_slice}, :]$  is filled into the corresponding location of the  $(q-b-1)$ -dimensional tensor to form the *output* tensor (Examples 2, 3, 4 and 5 below)

This operator is the inverse of *ScatterND*.

*Example 1*

```

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

```

*Example 2*

```

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

```

*Example 3*

```

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

```

*Example 4*

```

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

```

*Example 5*

```

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

```

**modeci\_mdf.functions.onnx.gemm**

`modeci_mdf.functions.onnx.gemm(*args, **kwargs)`

General Matrix multiplication: [https://en.wikipedia.org/wiki/Basic\\_Linear\\_Algebra\\_Subprograms#Level\\_3](https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms#Level_3)

$A' = \text{transpose}(A)$  if `transA` else `A`

$B' = \text{transpose}(B)$  if `transB` else `B`

Compute  $Y = \alpha * A' * B' + \beta * C$ , where input tensor `A` has shape  $(M, K)$  or  $(K, M)$ , input tensor `B` has shape  $(K, N)$  or  $(N, K)$ , input tensor `C` is broadcastable to shape  $(M, N)$ , and output tensor `Y` has shape  $(M, N)$ . `A` will be transposed before doing the computation if attribute `transA` is non-zero, same for `B` and `transB`. This operator supports **unidirectional broadcasting** (tensor `C` should be unidirectional broadcastable to tensor `A * B`); for more details please check [the doc](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.

### modeci\_mdf.functions.onnx.get\_all\_schemas\_version

`modeci_mdf.functions.onnx.get_all_schemas_version` (*max\_version*: *int*) → `List[onnx.onnx_cpp2py_export.defs.OpSchema]`

Enumerate all the OpSchemas available from ONNX.

**Parameters** `max_version` – Only include up to `max_version`.

**Returns** A list of all OpSchemas

### modeci\_mdf.functions.onnx.get\_onnx\_ops

`modeci_mdf.functions.onnx.get_onnx_ops` (*opset\_version*: *int* = 15) → `List[Dict]`  
Enumerate all available ONNX operations and generate MDF function specifications for each one.

**Parameters** `opset_version` – The opset version to use.

**Returns**

A list of MDF function specifications. Each entry is a Dict that is feed directly to `add_mdf_function`.

### modeci\_mdf.functions.onnx.get\_onnx\_schema

`modeci_mdf.functions.onnx.get_onnx_schema` (*func\_name*: *str*, *opset\_version*: *int* = 15) → `onnx.onnx_cpp2py_export.defs.OpSchema`

Return the ONNX schema corresponding to a generated ONNX python function with name `func_name`

**Parameters**

- `func_name` (*str*) – the name of the ONNX python function
- `opset_version` (*int*, *optional*) – The opset version to use. Defaults to `onnx_opset_version`.

**Raises** `ValueError` – `func_name` does not correspond to a generated ONNX python function

**Returns** The ONNX schema corresponding to function `func_name`

**Return type** `onnx.defs.OpSchema`

### modeci\_mdf.functions.onnx.globalaveragepool

`modeci_mdf.functions.onnx.globalaveragepool` (*\*args*, *\*\*kwargs*)

`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.

### modeci\_mdf.functions.onnx.globallppool

`modeci_mdf.functions.onnx.globallppool` (*\*args*, *\*\*kwargs*)

`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.

### modeci\_mdf.functions.onnx.globalmaxpool

`modeci_mdf.functions.onnx.globalmaxpool(*args, **kwargs)`

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.

### modeci\_mdf.functions.onnx.greater

`modeci_mdf.functions.onnx.greater(*args, **kwargs)`

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).

### modeci\_mdf.functions.onnx.greaterorequal

`modeci_mdf.functions.onnx.greaterorequal(*args, **kwargs)`

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).

### modeci\_mdf.functions.onnx.gru

`modeci_mdf.functions.onnx.gru(*args, **kwargs)`

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:

$\text{Relu}(x) - \max(0, x)$

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

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

(NOTE: Below are optional)

$\text{Affine}(x) - \alpha * x + \beta$

$\text{LeakyRelu}(x) - x \text{ if } x \geq 0 \text{ else } \alpha * x$

$\text{ThresholdedRelu}(x) - x \text{ if } x \geq \alpha \text{ else } 0$

$\text{ScaledTanh}(x) - \alpha * \text{Tanh}(\beta * x)$

$\text{HardSigmoid}(x) - \min(\max(\alpha * x + \beta, 0), 1)$

$\text{Elu}(x) - x \text{ if } x \geq 0 \text{ else } \alpha * (e^x - 1)$

$\text{Softsign}(x) - x/(1 + |x|)$

$\text{Softplus}(x) - \log(1 + e^x)$

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

- $z_t = f(X_t * (W_z^T) + H_{t-1} * (R_z^T) + W_{bz} + R_{bz})$
- $r_t = f(X_t * (W_r^T) + H_{t-1} * (R_r^T) + W_{br} + R_{br})$
- $h_t = g(X_t * (W_h^T) + (r_t \text{ (.) } H_{t-1}) * (R_h^T) + R_{bh} + W_{bh})$  # default, when  $\text{linear\_before\_reset} = 0$
- $h_t = g(X_t * (W_h^T) + (r_t \text{ (.) } (H_{t-1} * (R_h^T) + R_{bh})) + W_{bh})$  # when  $\text{linear\_before\_reset} \neq 0$
- $H_t = (1 - z_t) \text{ (.) } h_t + z_t \text{ (.) } H_{t-1}$

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.

## modeci\_mdf.functions.onnx.hardmax

`modeci_mdf.functions.onnx.hardmax(*args, **kwargs)`

The operator computes the hardmax values for the given input:

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

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



**modeci\_mdf.functions.onnx.hardsigmoid**

`modeci_mdf.functions.onnx.hardsigmoid(*args, **kwargs)`

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.

**modeci\_mdf.functions.onnx.hardswish**

`modeci_mdf.functions.onnx.hardswish(*args, **kwargs)`

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

**modeci\_mdf.functions.onnx.identity**

`modeci_mdf.functions.onnx.identity(*args, **kwargs)`

Identity operator

**modeci\_mdf.functions.onnx.if**

`modeci_mdf.functions.onnx.if(*args, **kwargs)`

If conditional

**modeci\_mdf.functions.onnx.import\_class**

`modeci_mdf.functions.onnx.import_class(name: str) → Any`

Import from a module specified by a string

**modeci\_mdf.functions.onnx.instancenormalization**

`modeci_mdf.functions.onnx.instancenormalization(*args, **kwargs)`

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

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

**modeci\_mdf.functions.onnx.isinf**

`modeci_mdf.functions.onnx.isinf(*args, **kwargs)`

Map infinity to true and other values to false.

**modeci\_mdf.functions.onnx.isnan**

`modeci_mdf.functions.onnx.isnan(*args, **kwargs)`

Returns which elements of the input are NaN.

**modeci\_mdf.functions.onnx.leakyrelu**

`modeci_mdf.functions.onnx.leakyrelu(*args, **kwargs)`

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  $x \geq 0$ , is applied to the data tensor elementwise.

**modeci\_mdf.functions.onnx.less**

`modeci_mdf.functions.onnx.less(*args, **kwargs)`

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).

**modeci\_mdf.functions.onnx.lessorequal**

`modeci_mdf.functions.onnx.lessorequal(*args, **kwargs)`

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).

**modeci\_mdf.functions.onnx.log**

`modeci_mdf.functions.onnx.log(*args, **kwargs)`

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

**modeci\_mdf.functions.onnx.logsoftmax**

`modeci_mdf.functions.onnx.logsoftmax(*args, **kwargs)`

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

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

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

## modeci\_mdf.functions.onnx.loop

modeci\_mdf.functions.onnx.**loop** (\*args, \*\*kwargs)

Generic Looping construct. This loop has multiple termination conditions:

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

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

Operator inputs defined as (max\_trip\_count, condition\_var).

**input (“”, “”):**

```
for (int i=0; ; ++i) { cond = ... // Note this value is ignored, but is required in the body
}
```

**input (“”, cond) // Note this is analogous to a while loop** bool cond = ...; for (int i=0; cond; ++i) {  
cond = ...;  
}

**input (“”, 1) // Note this is analogous to a do-while loop** bool cond = true for (int i=0; cond; ++i) {  
cond = ...;  
}

**input (trip\_count, “”) // Note this is analogous to a for loop** int trip\_count = ... for (int i=0; i < trip\_count; ++i) {  
cond = ...; // ignored  
}

**input (trip\_count, cond)** int trip\_count = ...; bool cond = ...; for (int i=0; i < trip\_count && cond; ++i) {  
cond = ...;  
}

*Sample usage - cond as well as trip count*

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

```
graph body-net (%i[INT32, scalar] // iteration number %keepgoing_in[BOOL, scalar] // incoming loop-termination-condition; not used %b_in[INT32, scalar] // incoming value of loop-carried-dependency b
```

```

) { %my_local = Add(%a, %b_in) %b_out = Sub(%a, %b_in) // outgoing value of loop-carried-
 dependency b %keepgoing_out = Greater(%my_local, %b_out) // outgoing loop-termination-
 condition %user_defined_val = Add(%b_in, %b_in) // scan-output value to be accumulated re-
 turn %keepgoing_out, %b_out, %user_defined_val
}

```

*Sample equivalent C code*

```

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

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

 for (int i=0; i < max_trip_count && keepgoing_out; ++i) {

 /* Implicitly-defined code: bind actual parameter values to formal parameter variables
 of loop-body */

 bool keepgoing_in = keepgoing_out; bool b_in = b_out;

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

 /* Implicitly defined-code */ user_defined_vals[i] = user_defined_val // accumulate scan-
 output values

 } // int t = my_local; // Can't do this. my_local is not accessible here.

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

}

```

There are several things of note in this code snippet:

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

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

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

### modeci\_mdf.functions.onnx.lpnormalization

`modeci_mdf.functions.onnx.lpnormalization(*args, **kwargs)`

Given a matrix, apply Lp-normalization along the provided axis.

### modeci\_mdf.functions.onnx.lppool

`modeci_mdf.functions.onnx.lppool(*args, **kwargs)`

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.

### modeci\_mdf.functions.onnx.lrn

`modeci_mdf.functions.onnx.lrn(*args, **kwargs)`

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  $X[n, c, d1, \dots, dk]$  in a tensor of shape  $(N \times C \times D1 \times D2, \dots, Dk)$ , its region is  $\{X[n, i, d1, \dots, dk] \mid \max(0, c - \text{floor}((\text{size} - 1) / 2)) \leq i \leq \min(C - 1, c + \text{ceil}((\text{size} - 1) / 2))\}$ .

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

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

### modeci\_mdf.functions.onnx.lstm

`modeci_mdf.functions.onnx.lstm(*args, **kwargs)`

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

Notations:

$X$  - input tensor

$i$  - input gate

$o$  - output gate

$f$  - forget gate

$c$  - cell gate

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

$W[iofc]$  -  $W$  parameter weight matrix for input, output, forget, and cell gates

$R[iofc]$  -  $R$  recurrence weight matrix for input, output, forget, and cell gates

$Wb[iofc]$  -  $W$  bias vectors for input, output, forget, and cell gates

$Rb[iofc]$  -  $R$  bias vectors for input, output, forget, and cell gates

$P[iof]$  -  $P$  peephole weight vector for input, output, and forget gates

$WB[iofc]$  - W parameter weight matrix for backward input, output, forget, and cell gates

$RB[iofc]$  - R recurrence weight matrix for backward input, output, forget, and cell gates

$WBb[iofc]$  - W bias vectors for backward input, output, forget, and cell gates

$RBb[iofc]$  - R bias vectors for backward input, output, forget, and cell gates

$PB[iof]$  - P peephole weight vector for backward input, output, and forget gates

$H$  - Hidden state

$num\_directions$  - 2 if direction == bidirectional else 1

Activation functions:

$Relu(x)$  -  $\max(0, x)$

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

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

(NOTE: Below are optional)

$Affine(x)$  -  $\alpha * x + \beta$

$LeakyRelu(x)$  -  $x$  if  $x \geq 0$  else  $\alpha * x$

$ThresholdedRelu(x)$  -  $x$  if  $x \geq \alpha$  else 0

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

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

$Elu(x)$  -  $x$  if  $x \geq 0$  else  $\alpha * (e^x - 1)$

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

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

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

- $it = f(Xt * (Wi^T) + H_{t-1} * (Ri^T) + Pi(.) Ct-1 + Wbi + Rbi)$
- $ft = f(Xt * (Wf^T) + H_{t-1} * (Rf^T) + Pf(.) Ct-1 + Wbf + Rbf)$
- $ct = g(Xt * (Wc^T) + H_{t-1} * (Rc^T) + Wbc + Rbc)$
- $Ct = ft(.) Ct-1 + it(.) ct$
- $ot = f(Xt * (Wo^T) + H_{t-1} * (Ro^T) + Po(.) Ct + Wbo + Rbo)$
- $Ht = ot(.) h(Ct)$

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.

### modeci\_mdf.functions.onnx.matmul

modeci\_mdf.functions.onnx.matmul(\*args, \*\*kwargs)

Matrix product that behaves like numpy.matmul: <https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.matmul.html>

### modeci\_mdf.functions.onnx.matmulinteger

modeci\_mdf.functions.onnx.matmulinteger(\*args, \*\*kwargs)

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.

### modeci\_mdf.functions.onnx.max

modeci\_mdf.functions.onnx.max(\*args, \*\*kwargs)

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).

### modeci\_mdf.functions.onnx.maxpool

modeci\_mdf.functions.onnx.maxpool(\*args, \*\*kwargs)

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:  $\text{output\_spatial\_shape}[i] = \text{floor}((\text{input\_spatial\_shape}[i] + \text{pad\_shape}[i] - ((\text{kernel\_spatial\_shape}[i] - 1) * \text{dilations}[i] + 1)) / \text{strides\_spatial\_shape}[i] + 1)$  or  $\text{output\_spatial\_shape}[i] = \text{ceil}((\text{input\_spatial\_shape}[i] + \text{pad\_shape}[i] - ((\text{kernel\_spatial\_shape}[i] - 1) * \text{dilations}[i] + 1)) / \text{strides\_spatial\_shape}[i] + 1)$  if `ceil_mode` is enabled

$\text{pad\_shape}[i]$  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`:  $\text{output\_spatial\_shape}[i] = \text{ceil}((\text{input\_spatial\_shape}[i] - ((\text{kernel\_spatial\_shape}[i] - 1) * \text{dilations}[i] + 1) + 1) / \text{strides\_spatial\_shape}[i])$  `SAME_UPPER` or `SAME_LOWER`:  $\text{output\_spatial\_shape}[i] = \text{ceil}(\text{input\_spatial\_shape}[i] / \text{strides\_spatial\_shape}[i])$  And pad shape will be following if `SAME_UPPER` or `SAME_LOWER`:  $\text{pad\_shape}[i] = (\text{output\_spatial\_shape}[i] - 1) * \text{strides\_spatial\_shape}[i] + ((\text{kernel\_spatial\_shape}[i] - 1) * \text{dilations}[i] + 1) - \text{input\_spatial\_shape}[i]$  The output of each pooling window is maximum number of elements exclude pad.

### modeci\_mdf.functions.onnx.maxroipool

modeci\_mdf.functions.onnx.maxroipool(\*args, \*\*kwargs)

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]).

### modeci\_mdf.functions.onnx.maxunpool

modeci\_mdf.functions.onnx.maxunpool(\*args, \*\*kwargs)

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

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

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

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

### modeci\_mdf.functions.onnx.mean

modeci\_mdf.functions.onnx.mean(\*args, \*\*kwargs)

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).

### modeci\_mdf.functions.onnx.meanvariancenormalization

modeci\_mdf.functions.onnx.meanvariancenormalization(\*args, \*\*kwargs)

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

### modeci\_mdf.functions.onnx.min

modeci\_mdf.functions.onnx.min(\*args, \*\*kwargs)

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).



## modeci\_mdf.functions.onnx.mod

`modeci_mdf.functions.onnx.mod(*args, **kwargs)`

**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).

## modeci\_mdf.functions.onnx.mul

`modeci_mdf.functions.onnx.mul(*args, **kwargs)`

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).

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

## modeci\_mdf.functions.onnx.multinomial

`modeci_mdf.functions.onnx.multinomial(*args, **kwargs)`

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

## modeci\_mdf.functions.onnx.neg

`modeci_mdf.functions.onnx.neg(*args, **kwargs)`

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.

## modeci\_mdf.functions.onnx.negativeloglikelihoodloss

`modeci_mdf.functions.onnx.negativeloglikelihoodloss(*args, **kwargs)`

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

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

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

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

loss is zero for the case when target-value equals ignore\_index.

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

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

$\text{mean}(\text{loss})$ , if “weight” is not provided,

or if weight is provided,

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

**If “reduction” attribute is set to “sum”, the output is a scalar:**  $\text{sum}(\text{loss})$ .

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
```

### modeci\_mdf.functions.onnx.nonmaxsuppression

`modeci_mdf.functions.onnx.nonmaxsuppression(*args, **kwargs)`

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.

### modeci\_mdf.functions.onnx.nonzero

`modeci_mdf.functions.onnx.nonzero(*args, **kwargs)`

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

### modeci\_mdf.functions.onnx.not

`modeci_mdf.functions.onnx.not(*args, **kwargs)`

Returns the negation of the input tensor element-wise.

### modeci\_mdf.functions.onnx.onehot

`modeci_mdf.functions.onnx.onehot(*args, **kwargs)`

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 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.

### modeci\_mdf.functions.onnx.optional

`modeci_mdf.functions.onnx.optional(*args, **kwargs)`

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

### modeci\_mdf.functions.onnx.optionalgetelement

`modeci_mdf.functions.onnx.optionalgetelement(*args, **kwargs)`

Outputs the element in the optional-type input. It is an error if the input value does not have an element and the behavior is undefined in this case.

### modeci\_mdf.functions.onnx.optionalhaselement

`modeci_mdf.functions.onnx.optionalhaselement(*args, **kwargs)`

Returns true if the optional-type input contains an element. If it is an empty optional-type, this op returns false.

### modeci\_mdf.functions.onnx.or

`modeci_mdf.functions.onnx.or(*args, **kwargs)`

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).

### modeci\_mdf.functions.onnx.pad

`modeci_mdf.functions.onnx.pad(*args, **kwargs)`

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, empty string, or False)*
- 2) *reflect - pads with the reflection of the vector mirrored on the first and last values of the vector along each axis*
- 3) *edge - pads with the edge values of array*

**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],
]

```

### modeci\_mdf.functions.onnx.pow

`modeci_mdf.functions.onnx.pow(*args, **kwargs)`

Pow takes input data (Tensor<T>) and exponent Tensor, and produces one output data (Tensor<T>) where the function  $f(x) = x^{\text{exponent}}$ , is applied to the data tensor elementwise. This operator supports **multidirectional (i.e., Numpy-style) broadcasting**; for more details please check [the doc](Broadcasting.md).

### modeci\_mdf.functions.onnx.predict\_with\_onnxruntime

`modeci_mdf.functions.onnx.predict_with_onnxruntime(model_def, *inputs) → Dict[str, numpy.array]`

Simple helper to run an ONNX model with a set of inputs.

#### Parameters

- **model\_def** – The ONNX model to run.
- **\*inputs** – Input values to pass to the model.

**Returns** A dict of output values, keys are output names for the model. Values are the output values of the model.

### modeci\_mdf.functions.onnx.prelu

`modeci_mdf.functions.onnx.prelu(*args, **kwargs)`

PReLU takes input data (Tensor<T>) and slope tensor as input, and produces one output data (Tensor<T>) where the function  $f(x) = \text{slope} * x \text{ for } x < 0, f(x) = x \text{ for } x \geq 0$ , is applied to the data tensor elementwise. This operator supports **unidirectional broadcasting** (tensor slope should be unidirectional broadcastable to input tensor X); for more details please check [the doc](Broadcasting.md).

### modeci\_mdf.functions.onnx.qlinearconv

`modeci_mdf.functions.onnx.qlinearconv(*args, **kwargs)`

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  $\text{scale} = \text{input scale} * \text{weight scale}$  and zero point as 0.

### modeci\_mdf.functions.onnx.qlinearmatmul

`modeci_mdf.functions.onnx.qlinearmatmul(*args, **kwargs)`

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  $y = \text{saturate}((x / y\_scale) + y\_zero\_point)$ . 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 N-D tensor (per row for 'a' and per column for 'b'). Scalar refers to per tensor quantization whereas N-D refers to per row or per column quantization. If the input is 2D of shape [M, K] then zero point and scale tensor may be an M element vector [ $v_1, v_2, \dots, v_M$ ] for per row quantization and K element vector of shape [ $v_1, v_2, \dots, v_K$ ] for per column quantization. If the input is N-D tensor with shape [D1, D2, M, K] then zero point and scale tensor may have shape [D1, D2, M, 1] for per row quantization and shape [D1, D2, 1, K] for per column quantization. Production must never overflow, and accumulation may overflow if and only if in 32 bits.

### modeci\_mdf.functions.onnx.quantizelinear

`modeci_mdf.functions.onnx.quantizelinear(*args, **kwargs)`

The linear quantization operator. It consumes a high precision tensor, a scale, and a zero point to compute the low precision / quantized tensor. The scale factor and zero point must have same shape, and can be either a scalar for per-tensor / per layer quantization, or a 1-D tensor for per-axis quantization. The quantization formula is  $y = \text{saturate}((x / y\_scale) + y\_zero\_point)$ . 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.

### modeci\_mdf.functions.onnx.randomnormal

`modeci_mdf.functions.onnx.randomnormal(*args, **kwargs)`

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.

### modeci\_mdf.functions.onnx.randomnormallike

`modeci_mdf.functions.onnx.randomnormallike(*args, **kwargs)`

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.

### modeci\_mdf.functions.onnx.randomuniform

`modeci_mdf.functions.onnx.randomuniform(*args, **kwargs)`

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.

### modeci\_mdf.functions.onnx.randomuniformlike

`modeci_mdf.functions.onnx.randomuniformlike(*args, **kwargs)`

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.

### modeci\_mdf.functions.onnx.range

`modeci_mdf.functions.onnx.range(*args, **kwargs)`

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-

$$\text{number\_of\_elements} = \max(\text{ceil}((\text{limit} - \text{start}) / \text{delta}), 0)$$

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]

### modeci\_mdf.functions.onnx.reciprocal

`modeci_mdf.functions.onnx.reciprocal(*args, **kwargs)`

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.

### modeci\_mdf.functions.onnx.reduce1

`modeci_mdf.functions.onnx.reduce11(*args, **kwargs)`

Computes the L1 norm of the input tensor's element along the provided axes. The resulting tensor has the same rank as the input if keepdims equals 1. If keepdims equals 0, then the resulting tensor has the reduced dimension pruned.

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

### modeci\_mdf.functions.onnx.reduce2

`modeci_mdf.functions.onnx.reduce12(*args, **kwargs)`

Computes the L2 norm of the input tensor's element along the provided axes. The resulting tensor has the same rank as the input if keepdims equals 1. If keepdims equals 0, then the resulting tensor has the reduced dimension pruned.

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

### modeci\_mdf.functions.onnx.reducelogsum

`modeci_mdf.functions.onnx.reducelogsum(*args, **kwargs)`

Computes the log sum of the input tensor's element along the provided axes. The resulting tensor has the same rank as the input if keepdims equals 1. If keepdims equals 0, then the resulting tensor has the reduced dimension pruned.

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

### modeci\_mdf.functions.onnx.reducelogsumexp

`modeci_mdf.functions.onnx.reducelogsumexp(*args, **kwargs)`

Computes the log sum exponent of the input tensor's element along the provided axes. The resulting tensor has the same rank as the input if keepdims equals 1. If keepdims equals 0, then the resulting tensor has the reduced dimension pruned.

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



### modeci\_mdf.functions.onnx.reduceamax

`modeci_mdf.functions.onnx.reduceamax(*args, **kwargs)`

Computes the max of the input tensor's element along the provided axes. The resulting tensor has the same rank as the input if keepdims equals 1. If keepdims equals 0, then the resulting tensor has the reduced dimension pruned.

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

### modeci\_mdf.functions.onnx.reduceamean

`modeci_mdf.functions.onnx.reduceamean(*args, **kwargs)`

Computes the mean of the input tensor's element along the provided axes. The resulting tensor has the same rank as the input if keepdims equals 1. If keepdims equals 0, then the resulting tensor has the reduced dimension pruned.

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

### modeci\_mdf.functions.onnx.reduceamin

`modeci_mdf.functions.onnx.reduceamin(*args, **kwargs)`

Computes the min of the input tensor's element along the provided axes. The resulting tensor has the same rank as the input if keepdims equals 1. If keepdims equals 0, then the resulting tensor has the reduced dimension pruned.

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

### modeci\_mdf.functions.onnx.reduceaprod

`modeci_mdf.functions.onnx.reduceaprod(*args, **kwargs)`

Computes the product of the input tensor's element along the provided axes. The resulting tensor has the same rank as the input if keepdims equals 1. If keepdims equals 0, then the resulting tensor has the reduced dimension pruned.

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

### modeci\_mdf.functions.onnx.reduceasum

`modeci_mdf.functions.onnx.reduceasum(*args, **kwargs)`

Computes the sum of the input tensor's element along the provided axes. The resulting tensor has the same rank as the input if keepdims equals 1. If keepdims equals 0, then the resulting tensor has the reduced dimension pruned.

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

### modeci\_mdf.functions.onnx.reducesumsquare

`modeci_mdf.functions.onnx.reducesumsquare(*args, **kwargs)`

Computes the sum square of the input tensor's element along the provided axes. The resulting tensor has the same rank as the input if keepdims equals 1. If keepdims equals 0, then the resulting tensor has the reduced dimension pruned.

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

### modeci\_mdf.functions.onnx.relu

`modeci_mdf.functions.onnx.relu(*args, **kwargs)`

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.

### modeci\_mdf.functions.onnx.reshape

`modeci_mdf.functions.onnx.reshape(*args, **kwargs)`

Reshape the input tensor similar to `numpy.reshape`. First input is the data tensor, second input is a shape tensor which specifies the output shape. It outputs the reshaped tensor. At most one dimension of the new shape can be -1. In this case, the value is inferred from the size of the tensor and the remaining dimensions. A dimension could also be 0, in which case the actual dimension value is unchanged (i.e. taken from the input tensor). If 'allowzero' is set, and the new shape includes 0, the dimension will be set explicitly to zero (i.e. not taken from input tensor). Shape (second input) could be an empty shape, which means converting to a scalar. The input tensor's shape and the output tensor's shape are required to have the same number of elements.

If the attribute 'allowzero' is set, it is invalid for the specified shape to contain both a zero value and -1, as the value of the dimension corresponding to -1 cannot be determined uniquely.

### modeci\_mdf.functions.onnx.resize

`modeci_mdf.functions.onnx.resize(*args, **kwargs)`

Resize the input tensor. In general, it calculates every value in the output tensor as a weighted average of neighborhood (a.k.a. sampling locations) in the input tensor. Each dimension value of the output tensor is:

$\text{output\_dimension} = \text{floor}(\text{input\_dimension} * (\text{roi\_end} - \text{roi\_start}) * \text{scale})$  if input "sizes" is not specified.

### modeci\_mdf.functions.onnx.reversesequence

`modeci_mdf.functions.onnx.reversesequence(*args, **kwargs)`

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]]

```

**modeci\_mdf.functions.onnx.rnn**

modeci\_mdf.functions.onnx.rnn(\*args, \*\*kwargs)

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

Notations:

$X$  - input tensor

$i$  - input gate

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

$W_i$  - W parameter weight matrix for input gate

$R_i$  - R recurrence weight matrix for input gate

$W_{bi}$  - W parameter bias vector for input gate

$R_{bi}$  - R parameter bias vector for input gate

$W_{Bi}$  - W parameter weight matrix for backward input gate

$R_{Bi}$  - R recurrence weight matrix for backward input gate

$W_{Bbi}$  - WR bias vectors for backward input gate

$R_{Bbi}$  - 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  $x \geq 0$  else  $\alpha * x$

ThresholdedRelu(x) -  $x$  if  $x \geq \alpha$  else 0

ScaledTanh(x) -  $\alpha * \tanh(\beta * x)$

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

Elu(x) -  $x$  if  $x \geq 0$  else  $\alpha * (e^x - 1)$

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

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

Equations (Default: f=Tanh):

- $H_t = f(X_t * (W_i^T) + H_{t-1} * (R_i^T) + W_{bi} + R_{bi})$

This operator has **optional** inputs/outputs. See [the doc](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.

### modeci\_mdf.functions.onnx.roialign

`modeci_mdf.functions.onnx.roialign(*args, **kwargs)`

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.

### modeci\_mdf.functions.onnx.round

`modeci_mdf.functions.onnx.round(*args, **kwargs)`

Round takes one input Tensor and rounds the values, element-wise, meaning it finds the nearest integer for each value. In case of halves, the rule is to round them to the nearest even integer. 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] `

### modeci\_mdf.functions.onnx.run\_onnx\_op

`modeci_mdf.functions.onnx.run_onnx_op(op_name: str, inputs: Dict[str, numpy.array], output_names: List[str], opset_version: int = 15, **attributes)`

Simple helper function that invokes a single ONNX operator with inputs and attributes and returns the results. This isn't typically done in ONNX because graphs usually consist of more than one operation. This wrapper probably creates a significant amount of overhead for but if we want to execute an ONNX graph op by op it is the easiest thing to do.

#### Parameters

- **op\_name** – The name of the operation to run, (Conv, Pad, etc.)
- **inputs** – A dict keyed by input name where the values are the input values to pass to the operation.
- **output\_names** – The names to use for the output values.
- **\*\*attributes** – Any additional attributes for the ONNX operation.

**Returns** A dict of output values, keys are output\_names. Values are the output values of the operation.

## modeci\_mdf.functions.onnx.scan

`modeci_mdf.functions.onnx.scan(*args, **kwargs)`

Scan can be used to iterate over one or more `scan_input` tensors, constructing zero or more `scan_output` tensors. It combines ideas from general recurrences, functional programming constructs such as `scan`, `fold`, `map`, and `zip`, and is intended to enable generalizations of RNN-like constructs for sequence-to-sequence processing. Other tensors (referred to as `state_variables` here) can be used to carry a state when iterating from one element to another (similar to hidden-state in RNNs, also referred to as loop-carried dependences in the context of loops). Many common usages involve a single `scan_input` tensor (where functionality similar to `scan`, `fold` and `map` can be obtained). When more than one `scan_input` is used, a behavior similar to `zip` is obtained.

The attribute body must be a graph, specifying the computation to be performed in every iteration. It takes as input the current values of the `state_variables` and the current iterated element of the `scan_inputs`. It must return the (updated) values of the `state_variables` and zero or more `scan_output_element` tensors. The values of the `scan_output_element` tensors are concatenated over all the iterations to produce the `scan_output` values of the `scan` construct (similar to the concatenated intermediate hidden-state values of RNN-like constructs). All the output tensors (`state_variables` as well as `scan_output_element` tensors) are required to have the same shape in each iteration of the loop (a restriction imposed to enable efficient memory allocation).

Note that the iterated element passed to the body subgraph does not have a sequence axis. It will have a rank one less than the rank of the corresponding `scan_input`.

The `scan` operation returns the final values of the `state_variables` as well as the `scan_outputs`.

The optional attribute `scan_input_directions` specifies the direction (forward or backward) for each `scan_input`. If this attribute is omitted, all sequences are scanned in the forward direction. A bidirectional scan may be performed by specifying the same tensor input twice in the `scan_inputs`, once with a forward direction, and once with a backward direction.

The `scan_output` of the operation is produced by concatenating the `scan_output_element` values produced by the body in each iteration. The optional attribute `scan_output_directions` specifies the direction in which `scan_output` is constructed (by appending or prepending the `scan_output_element` to `scan_output` in each iteration) for each `scan_output`. If this attribute is omitted, the `scan_output_element` is appended to the `scan_output` in each iteration.

The optional attribute `scan_input_axes` specifies the axis to be scanned for each `scan_input`. If omitted, every `scan_input` will be scanned in axis 0. For example, if axis 0 is the batch axis and axis 1 is the time axis (to be scanned), specify an axis value of 1. Note that scanning a non-zero axis may be less efficient than scanning axis zero.

The optional attribute `scan_output_axes` specifies the axis along which the `scan_outputs` are accumulated for each `scan_output`. For example, if axis 1 is the time axis (to be scanned) for both inputs and outputs, specify a `scan_input` axis and `scan_output` axis value of 1.

Note that because of the ONNX restriction that only the last parameter of an operator can be variadic, the initial-states and scan-inputs are listed together as one input parameter. Similarly, the final-states and scan-outputs are listed together as one output parameter. The attribute `num_scan_inputs` indicates the number `M` of scan-inputs.

The behavior of

```
Scan < num_scan_inputs = m, body = loop-body, scan_input_axes = [axis_1, ..., axis_m]
> (init_1, ..., init_n, scan_1, ..., scan_m)
```

is equivalent to the following pseudo-code:

```
// scan_i.shape[axis_i] denotes the (max) sequence-length of scan_i // scan_i.shape[axis_i] is required
to be equal to scan_j.shape[axis_j] for all i,j. sequence_length = scan_1.shape[axis_1];

// initialize state-variables st_1 = init_1; ... st_n = init_n; // initialize scan-output variables: []
denotes an empty tensor scan_out_1 = []; ...; scan_out_k = []; // identify number of iterations:
```

```
// execute loop for (int t = 0; t < sequence_length; ++t) {
 // generate the scan-input elements: the notation T<axis=k>[t] indicates the sub-tensor
 // of rank one less than T obtained by indexing T at position t along axis k. si_1 =
 scan_1<axis=axis_1>[t]; ... ; si_m = scan_m<axis=axis_m>[t]; // execute loop-body st_1,
 ..., st_n, so_1, ..., so_k = loop-body(st_1, ..., st_n, si_1, ..., si_m) // accumulate the
 scan-output elements scan_out_1 = Concat<axis=0>(scan_out_1, so_1); ... ; scan_out_k
 = Concat<axis=0>(scan_out_k, so_k);
}
return st_1, ..., st_n, scan_out_1, ..., scan_out_k;
```

*Sample usage: Encoding RNN using a Scan*

The following example shows how a simple RNN over an input tensor %X, with weight tensor %Wi, recurrence weight tensor %Ri, bias tensors %Wbi and %Rbi, and initial hidden-state %H\_0 can be encoded as a ScanLoop. Note that the loop-body is a nested graph, and it directly computes %Wi, %Ri, %Wbi, and %Rbi (typically constants or initializers in the body graph). If these values are computed in the outer graph, they need to be passed in as extra state\_variables.

```
graph rnn-encoding { %H_0 = ... %X = ... %Y_h, %Y = Scan[body = <graph rnn-cell-1>,
 num_scan_inputs=1](%H_0, %X) return %Y, %Y_h
}

graph rnn-cell-1 (%H_tminus1[FLOAT, tensor] %X_t[FLOAT, tensor]
) { %Wi = ... %Ri = ... %Wbi = ... %Rbi = ... %t1 = X_t * (Wi^T) %t2 = H_tminus1*(Ri^T)
 %t3 = Add(%t1, %t2) %t4 = Add(%t3, %Wbi) %t5 = Add(%t4, %Rbi) %Ht = Tanh(%t5) %Ac-
 cumulate = Identity(%Ht) return %Ht, %Accumulate
}
```

## modeci\_mdf.functions.onnx.scatter

`modeci_mdf.functions.onnx.scatter (*args, **kwargs)`

This operator is deprecated. Please use `ScatterElements`, which provides the same functionality.

`Scatter` takes three inputs *data*, *updates*, and *indices* of the same rank  $r \geq 1$  and an optional attribute *axis* that identifies an axis of *data* (by default, the outer-most axis, that is axis 0). The output of the operation is produced by creating a copy of the input *data*, and then updating its value to values specified by *updates* at specific index positions specified by *indices*. Its output shape is the same as the shape of *data*.

For each entry in *updates*, the target index in *data* is obtained by combining the corresponding entry in *indices* with the index of the entry itself: the index-value for dimension = *axis* is obtained from the value of the corresponding entry in *indices* and the index-value for dimension != *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]]
`

```

### modeci\_mdf.functions.onnx.scatterelements

`modeci_mdf.functions.onnx.scatterelements (*args, **kwargs)`

ScatterElements takes three inputs *data*, *updates*, and *indices* of the same rank  $r \geq 1$  and an optional attribute *axis* that identifies an axis of *data* (by default, the outer-most axis, that is axis 0). The output of the operation is produced by creating a copy of the input *data*, and then updating its value to values specified by *updates* at specific index positions specified by *indices*. Its output shape is the same as the shape of *data*.

For each entry in *updates*, the target index in *data* is obtained by combining the corresponding entry in *indices* with the index of the entry itself: the index-value for dimension = *axis* is obtained from the value of the corresponding entry in *indices* and the index-value for dimension  $\neq$  *axis* is obtained from the index of the entry itself.

For instance, in a 2-D tensor case, the update corresponding to the  $[i][j]$  entry is performed as below: `'''`

```

 output[indices[i][j]][j] = updates[i][j] if axis = 0, output[i][indices[i][j]] = updates[i][j] if axis = 1,
'''

```

This operator is the inverse of GatherElements. It is similar to Torch's Scatter operation.

Example 1: `'''`

```

data = [[0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0],
] indices = [
 [1, 0, 2], [0, 2, 1],
] updates = [
 [1.0, 1.1, 1.2], [2.0, 2.1, 2.2],
] output = [
 [2.0, 1.1, 0.0] [1.0, 0.0, 2.2] [0.0, 2.1, 1.2]
]
` Example 2: `
data = [[1.0, 2.0, 3.0, 4.0, 5.0]] indices = [[1, 3]] updates = [[1.1, 2.1]] axis = 1 output = [[1.0, 1.1,
3.0, 2.1, 5.0]]
'''

```

## modeci\_mdf.functions.onnx.scatternd

modeci\_mdf.functions.onnx.scatternd(\*args, \*\*kwargs)

ScatterND takes three inputs *data* tensor of rank  $r \geq 1$ , *indices* tensor of rank  $q \geq 1$ , and *updates* tensor of rank  $q + r - \text{indices.shape}[-1] - 1$ . The output of the operation is produced by creating a copy of the input *data*, and then updating its value to values specified by *updates* at specific index positions specified by *indices*. Its output shape is the same as the shape of *data*. 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  $\text{indices.shape}[-1]$ , the last dimension in the shape of *indices*.**

*indices* is treated as a  $(q-1)$ -dimensional tensor of  $k$ -tuples, where each  $k$ -tuple is a partial-index into *data*.

Hence,  $k$  can be a value at most the rank of *data*. When  $k$  equals  $\text{rank}(\text{data})$ , each update entry specifies an update to a single element of the tensor. When  $k$  is less than  $\text{rank}(\text{data})$  each update entry specifies an update to a slice of the tensor. Index values are allowed to be negative, as per the usual convention for counting backwards from the end, but are expected in the valid range.

*updates* is treated as a  $(q-1)$ -dimensional tensor of replacement-slice-values. Thus, the first  $(q-1)$  dimensions of  $\text{updates.shape}$  must match the first  $(q-1)$  dimensions of  $\text{indices.shape}$ . 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  $\text{indices.shape}[0:q-1] ++ \text{data.shape}[k:r-1]$ , 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  $\text{idx1} \neq \text{idx2}$ , then  $\text{indices}[\text{idx1}] \neq \text{indices}[\text{idx2}]$ . 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]]]
```

`'''`



**modeci\_mdf.functions.onnx.selu**

`modeci_mdf.functions.onnx.selu(*args, **kwargs)`

Selu takes one input data (Tensor<T>) and produces one output data (Tensor<T>) where the scaled exponential linear unit function,  $y = \text{gamma} * (\alpha * e^x - \alpha)$  for  $x \leq 0$ ,  $y = \text{gamma} * x$  for  $x > 0$ , is applied to the tensor elementwise.

**modeci\_mdf.functions.onnx.sequenceat**

`modeci_mdf.functions.onnx.sequenceat(*args, **kwargs)`

Outputs a tensor copy from the tensor at 'position' in 'input\_sequence'. Accepted range for 'position' is in  $[-n, n - 1]$ , where  $n$  is the number of tensors in 'input\_sequence'. Negative value means counting positions from the back.

**modeci\_mdf.functions.onnx.sequenceconstruct**

`modeci_mdf.functions.onnx.sequenceconstruct(*args, **kwargs)`

Construct a tensor sequence containing 'inputs' tensors. All tensors in 'inputs' must have the same data type.

**modeci\_mdf.functions.onnx.sequenceempty**

`modeci_mdf.functions.onnx.sequenceempty(*args, **kwargs)`

Construct an empty tensor sequence, with given data type.

**modeci\_mdf.functions.onnx.sequenceerase**

`modeci_mdf.functions.onnx.sequenceerase(*args, **kwargs)`

Outputs a tensor sequence that removes the tensor at 'position' from 'input\_sequence'. Accepted range for 'position' is in  $[-n, n - 1]$ , where  $n$  is the number of tensors in 'input\_sequence'. Negative value means counting positions from the back. 'position' is optional, by default it erases the last tensor from 'input\_sequence'.

**modeci\_mdf.functions.onnx.sequenceinsert**

`modeci_mdf.functions.onnx.sequenceinsert(*args, **kwargs)`

Outputs a tensor sequence that inserts 'tensor' into 'input\_sequence' at 'position'. 'tensor' must have the same data type as 'input\_sequence'. Accepted range for 'position' is in  $[-n, n]$ , where  $n$  is the number of tensors in 'input\_sequence'. Negative value means counting positions from the back. 'position' is optional, by default it inserts 'tensor' to the back of 'input\_sequence'.

**modeci\_mdf.functions.onnx.sequencelength**

`modeci_mdf.functions.onnx.sequencelength(*args, **kwargs)`

Produces a scalar(tensor of empty shape) containing the number of tensors in 'input\_sequence'.

### modeci\_mdf.functions.onnx.shape

`modeci_mdf.functions.onnx.shape(*args, **kwargs)`

Takes a tensor as input and outputs an 1D int64 tensor containing the shape of the input tensor. Optional attributes start and end can be used to compute a slice of the input tensor's shape. If start axis is omitted, the slice starts from axis 0. The end axis, if specified, is exclusive (and the returned value will not include the size of that axis). If the end axis is omitted, the axes upto the last one will be included. Negative axes indicate counting back from the last axis. Note that axes will be clamped to the range [0, r-1], where r is the rank of the input tensor if they are out-of-range (after adding r in the case of negative axis). Thus, specifying any end value > r is equivalent to specifying an end value of r, and specifying any start value < -r is equivalent to specifying a start value of 0.

For example: Input tensor with shape: [2, 3, 4] No attributes specified. Output: [2, 3, 4]

Input tensor with shape: [2, 3, 4] start: -1 Output: [4]

Input tensor with shape: [2, 3, 4] end: -1 Output: [2, 3]

Input tensor with shape: [2, 3, 4] start: 1 end: 2 Output: [3]

### modeci\_mdf.functions.onnx.shrink

`modeci_mdf.functions.onnx.shrink(*args, **kwargs)`

Shrink takes one input data (Tensor<numeric>) and produces one Tensor output, having same datatype and shape with input. It has two attributes, lambda and bias. The formula of this operator is: If  $x < -\lambda$ ,  $y = x + \text{bias}$ ; If  $x > \lambda$ ,  $y = x - \text{bias}$ ; Otherwise,  $y = 0$ .

### modeci\_mdf.functions.onnx.sigmoid

`modeci_mdf.functions.onnx.sigmoid(*args, **kwargs)`

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.

### modeci\_mdf.functions.onnx.sign

`modeci_mdf.functions.onnx.sign(*args, **kwargs)`

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.

### modeci\_mdf.functions.onnx.sin

`modeci_mdf.functions.onnx.sin(*args, **kwargs)`

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

**modeci\_mdf.functions.onnx.sinh**

```
modeci_mdf.functions.onnx.sinh(*args, **kwargs)
```

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

**modeci\_mdf.functions.onnx.size**

```
modeci_mdf.functions.onnx.size(*args, **kwargs)
```

Takes a tensor as input and outputs a int64 scalar that equals to the total number of elements of the input tensor.

**modeci\_mdf.functions.onnx.slice**

```
modeci_mdf.functions.onnx.slice(*args, **kwargs)
```

Produces a slice of the input tensor along multiple axes. Similar to numpy: <https://numpy.org/doc/stable/user/basics.indexing.html?highlight=slice#slicing-and-striding>

Slice uses the *starts*, *ends*, *axes* and *steps* inputs to select a sub-tensor of its input *data* tensor.

An effective *start[i]*, *end[i]*, and *step[i]* must be computed for each *i* in  $[0, \dots, r-1]$  where  $r = \text{rank}(\text{input})$  as follows:

If *axes* are omitted, they are set to  $[0, \dots, r-1]$ . If *steps* are omitted, they are set to  $[1, \dots, 1]$  of length  $\text{len}(\text{starts})$

The effective values are initialized as  $\text{start}[i] = 0$ ,  $\text{end}[i] = \text{dims}[i]$  where *dims* are the dimensions of *input* and  $\text{step}[i] = 1$ .

All negative elements of *axes* are made non-negative by adding  $r$  to them, where  $r = \text{rank}(\text{input})$ .

All negative values in *starts[i]* and *ends[i]* have  $\text{dims}[\text{axes}[i]]$  added to them, where *dims* are the dimensions of *input*. Then  $\text{start}[\text{axes}[i]]$  is the adjusted *starts[i]* is clamped into the range  $[0, \text{dims}[\text{axes}[i]]]$  for positive stepping and  $[0, \text{dims}[\text{axes}[i]]-1]$  for negative stepping.

The clamping for the adjusted *ends[i]* depends on the sign of *steps[i]* and must accommodate copying 0 through  $\text{dims}[\text{axes}[i]]$  elements, so for positive stepping  $\text{end}[\text{axes}[i]]$  is clamped to  $[0, \text{dims}[\text{axes}[i]]]$ , while for negative stepping it is clamped to  $[-1, \text{dims}[\text{axes}[i]]-1]$ .

Finally,  $\text{step}[\text{axes}[i]] = \text{steps}[i]$ .

For slicing to the end of a dimension with unknown size, it is recommended to pass in *INT\_MAX* when slicing forward and 'INT\_MIN' when slicing backward.

**Example 1:**

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

**Example 2:**

```
data = [[1, 2, 3, 4], [5, 6, 7, 8],
] starts = [0, 1] ends = [-1, 1000] result = [
 [2, 3, 4],
]
```

### modeci\_mdf.functions.onnx.softmax

`modeci_mdf.functions.onnx.softmax(*args, **kwargs)`

The operator computes the normalized exponential values for the given input:

$$\text{Softmax}(\text{input}, \text{axis}) = \text{Exp}(\text{input}) / \text{ReduceSum}(\text{Exp}(\text{input}), \text{axis}=\text{axis}, \text{keepdims}=1)$$

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

### modeci\_mdf.functions.onnx.softmaxcrossentropyloss

`modeci_mdf.functions.onnx.softmaxcrossentropyloss(*args, **kwargs)`

Loss function that measures the softmax cross entropy between ‘scores’ and ‘labels’. This operator first computes a loss tensor whose shape is identical to the labels input. If the input is 2-D with shape (N, C), the loss tensor may be a N-element vector  $L = (l_1, l_2, \dots, l_N)$ . If the input is N-D tensor with shape (N, C, D1, D2, ..., Dk), the loss tensor L may have (N, D1, D2, ..., Dk) as its shape and  $L[i,][j_1][j_2] \dots [j_k]$  denotes a scalar element in L. After L is available, this operator can optionally do a reduction operator.

**shape(scores):** (N, C) where C is the number of classes, or (N, C, D1, D2,..., Dk), with  $K \geq 1$  in case of K-dimensional loss.

**shape(labels):** (N) where each value is  $0 \leq \text{labels}[i] \leq C-1$ , or (N, D1, D2,..., Dk), with  $K \geq 1$  in case of K-dimensional loss.

**The loss for one sample, l<sub>i</sub>, can calculated as follows:**  $l[i][d1][d2] \dots [dk] = -y[i][c][d1][d2] \dots [dk]$ , where i is the index of classes.

**or**  $l[i][d1][d2] \dots [dk] = -y[i][c][d1][d2] \dots [dk] * \text{weights}[c]$ , if ‘weights’ is provided.

**loss is zero for the case when label-value equals ignore\_index.**  $l[i][d1][d2] \dots [dk] = 0$ , when  $\text{labels}[n][d1][d2] \dots [dk] = \text{ignore\_index}$

**where:**  $p = \text{Softmax}(\text{scores})$   $y = \text{Log}(p)$   $c = \text{labels}[i][d1][d2] \dots [dk]$

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:  $\text{ReduceSum}(L) / \text{ReduceSum}(W)$ , where tensor W is of shape (N, D1, D2, ..., Dk) and  $W[n][d1][d2] \dots [dk] = \text{weights}[\text{labels}[i][d1][d2] \dots [dk]]$ .

### modeci\_mdf.functions.onnx.softplus

`modeci_mdf.functions.onnx.softplus(*args, **kwargs)`

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.

### modeci\_mdf.functions.onnx.softsign

`modeci_mdf.functions.onnx.softsign(*args, **kwargs)`

Calculates the softsign  $(x/(1+|x|))$  of the given input tensor element-wise.

**modeci\_mdf.functions.onnx.spacetodepth**

`modeci_mdf.functions.onnx.spacetodepth(*args, **kwargs)`

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.

**modeci\_mdf.functions.onnx.split**

`modeci_mdf.functions.onnx.split(*args, **kwargs)`

Split a tensor into a list of tensors, along the specified 'axis'. Lengths of the parts can be specified using input 'split'. Otherwise, the tensor is split to equal sized parts.

**modeci\_mdf.functions.onnx.splittosequence**

`modeci_mdf.functions.onnx.splittosequence(*args, **kwargs)`

Split a tensor into a sequence of tensors, along the specified 'axis'. Lengths of the parts can be specified using argument 'split'. 'split' must contain only positive numbers. 'split' is either a scalar (tensor of empty shape), or a 1-D tensor. If 'split' is a scalar, then 'input' will be split into equally sized chunks(if possible). Last chunk will be smaller if the 'input' size along the given axis 'axis' is not divisible by 'split'. Otherwise, the tensor is split into 'size(split)' chunks, with lengths of the parts on 'axis' specified in 'split'. In this scenario, the sum of entries in 'split' must be equal to the dimension size of input tensor on 'axis'.

**modeci\_mdf.functions.onnx.sqrt**

`modeci_mdf.functions.onnx.sqrt(*args, **kwargs)`

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.

**modeci\_mdf.functions.onnx.squeeze**

`modeci_mdf.functions.onnx.squeeze(*args, **kwargs)`

Remove single-dimensional entries from the shape of a tensor. Takes an input *axes* with a list of axes to squeeze. If *axes* is not provided, all the single dimensions will be removed from the shape. If an axis is selected with shape entry not equal to one, an error is raised.

**modeci\_mdf.functions.onnx.stringnormalizer**

`modeci_mdf.functions.onnx.stringnormalizer(*args, **kwargs)`

StringNormalization performs string operations for basic cleaning. This operator has only one input (denoted by X) and only one output (denoted by Y). This operator first examines the elements in the X, and removes elements specified in "stopwords" attribute. After removing stop words, the intermediate result can be further lowercased, uppercased, or just returned depending the "case\_change\_action" attribute. This operator only accepts [C]- and [1, C]-tensor. If all elements in X are dropped, the output will be the empty value of string tensor with shape [1] if input shape is [C] and shape [1, 1] if input shape is [1, C].

### modeci\_mdf.functions.onnx.sub

`modeci_mdf.functions.onnx.sub(*args, **kwargs)`

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).

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

### modeci\_mdf.functions.onnx.sum

`modeci_mdf.functions.onnx.sum(*args, **kwargs)`

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).

### modeci\_mdf.functions.onnx.tan

`modeci_mdf.functions.onnx.tan(*args, **kwargs)`

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

### modeci\_mdf.functions.onnx.tanh

`modeci_mdf.functions.onnx.tanh(*args, **kwargs)`

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

### modeci\_mdf.functions.onnx.tfidfvectorizer

`modeci_mdf.functions.onnx.tfidfvectorizer(*args, **kwargs)`

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.

### modeci\_mdf.functions.onnx.thresholdedrelu

`modeci_mdf.functions.onnx.thresholdedrelu(*args, **kwargs)`

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.

### modeci\_mdf.functions.onnx.tile

`modeci_mdf.functions.onnx.tile(*args, **kwargs)`

Constructs a tensor by tiling a given tensor. This is the same as function *tile* in Numpy, but no broadcast. For example  $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ ,  $B = [1, 2]$ ,  $\text{tile}(A, B) = \begin{bmatrix} 1 & 2 & 1 & 2 \\ 3 & 4 & 3 & 4 \end{bmatrix}$

### modeci\_mdf.functions.onnx.topk

`modeci_mdf.functions.onnx.topk(*args, **kwargs)`

Retrieve the top-K largest or smallest elements along a specified axis. Given an input tensor of shape  $[a_1, a_2, \dots, a_n, r]$  and integer argument  $k$ , return two outputs:

- Value tensor of shape  $[a_1, a_2, \dots, a_{\text{axis}-1}, k, a_{\text{axis}+1}, \dots, a_n]$**  which contains the values of the top  $k$  elements along the specified axis
- Index tensor of shape  $[a_1, a_2, \dots, a_{\text{axis}-1}, k, a_{\text{axis}+1}, \dots, a_n]$  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.

### modeci\_mdf.functions.onnx.transpose

`modeci_mdf.functions.onnx.transpose(*args, **kwargs)`

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  $(2, 1, 3)$ .

### modeci\_mdf.functions.onnx.trilu

`modeci_mdf.functions.onnx.trilu(*args, **kwargs)`

Given a 2-D matrix or batches of 2-D matrices, returns the upper or lower triangular part of the tensor(s). The attribute “upper” determines whether the upper or lower part is retained. If set to true, the upper triangular matrix is retained. Lower triangular matrix is retained otherwise. Default value for the “upper” attribute is true. Trilu takes one input tensor of shape  $[*, N, M]$ , where  $*$  is zero or more batch dimensions. The upper triangular part consists of the elements on and above the given diagonal ( $k$ ). The lower triangular part consists of elements on and below the diagonal. All other elements in the matrix are set to zero. If  $k = 0$ , the triangular part on and above/below the main diagonal is retained. If upper is set to true, a positive  $k$  retains the upper triangular matrix excluding the main diagonal and  $(k-1)$  diagonals above it. A negative  $k$  value retains the main diagonal and  **$|k|$**

diagonals below it. If upper is set to false, a positive k retains the lower triangular matrix including the main diagonal and k diagonals above it. A negative k value excludes the main diagonal and  $(|k|-1)$  diagonals below it.

## modeci\_mdf.functions.onnx.unique

`modeci_mdf.functions.onnx.unique(*args, **kwargs)`

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 occurrence in 'X'. The third optional output tensor 'inverse\_indices' contains, for elements of 'X', its corresponding indices in 'Y'. The fourth optional output tensor 'counts' contains the count of each element of 'Y' in the input.

Outputs are either sorted in ascending order or optionally in the order of the first occurrence of the values in the input.

<https://docs.scipy.org/doc/numpy/reference/generated/numpy.unique.html>

**Example 1:** `input_X = [2, 1, 1, 3, 4, 3]` `attribute_sorted = 0` `attribute_axis = None` `output_Y = [2, 1, 3, 4]` `output_indices = [0, 1, 3, 4]` `output_inverse_indices = [0, 1, 1, 2, 3, 2]` `output_counts = [1, 2, 2, 1]`

**Example 2:** `input_X = [[1, 3], [2, 3]]` `attribute_sorted = 1` `attribute_axis = None` `output_Y = [1, 2, 3]` `output_indices = [0, 2, 1]` `output_inverse_indices = [0, 2, 1, 2]` `output_counts = [1, 1, 2]`

**Example 3:** `input_X = [[1, 0, 0], [1, 0, 0], [2, 3, 4]]` `attribute_sorted = 1` `attribute_axis = 0` `output_Y = [[1, 0, 0], [2, 3, 4]]` `output_indices = [0, 2]` `output_inverse_indices = [0, 0, 1]` `output_counts = [2, 1]`

**Example 4:**

`input_x = [[[1., 1.], [0., 1.], [2., 1.], [0., 1.]], [[1., 1.], [0., 1.], [2., 1.], [0., 1.]]]`

`attribute_sorted = 1` `attribute_axis = 1`

intermediate data are presented below for better understanding:

there are 4 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]`



### modeci\_mdf.functions.onnx.unsqueeze

`modeci_mdf.functions.onnx.unsqueeze(*args, **kwargs)`

Insert single-dimensional entries to the shape of an input tensor (*data*). Takes one required input *axes* - which contains a list of dimension indices and this operator will insert a dimension of value 1 into the corresponding index of the output tensor (*expanded*).

**For example:** Given an input tensor (*data*) of shape [3, 4, 5], then `Unsqueeze(data, axes=[0, 4])` outputs a tensor (*expanded*) containing same data as *data* but with shape [1, 3, 4, 5, 1].

The input *axes* should not contain any duplicate entries. It is an error if it contains duplicates. The rank of the output tensor (*output\_rank*) is the rank of the input tensor (*data*) plus the number of values in *axes*. Each value in *axes* should be within the (inclusive) range [-*output\_rank*, *output\_rank* - 1]. The order of values in *axes* does not matter and can come in any order.

### modeci\_mdf.functions.onnx.upsample

`modeci_mdf.functions.onnx.upsample(*args, **kwargs)`

Upsample the input tensor. Each dimension value of the output tensor is:

$\text{output\_dimension} = \text{floor}(\text{input\_dimension} * \text{scale})$ .

### modeci\_mdf.functions.onnx.where

`modeci_mdf.functions.onnx.where(*args, **kwargs)`

Return elements, either from X or Y, depending on condition. Where behaves like `[numpy.where](https://docs.scipy.org/doc/numpy/reference/generated/numpy.where.html)` with three parameters.

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

### modeci\_mdf.functions.onnx.xor

`modeci_mdf.functions.onnx.xor(*args, **kwargs)`

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).

## 25.3.3 modeci\_mdf.functions.standard

Implementation of core MDF function ontology.

This module implements and registers all builtin MDF functions.

## Functions

|                                                            |                                                                                         |
|------------------------------------------------------------|-----------------------------------------------------------------------------------------|
| <code>add_mdf_function([name, description, ...])</code>    | Register a function with MDF function ontology.                                         |
| <code>create_python_expression([expression_string])</code> | Converts the mathematical representation of function into function expression in python |
| <code>create_python_function([name, ...])</code>           | Create a Python function e.g.                                                           |
| <code>substitute_args([expression_string, args])</code>    | Substitute arg with the value in args dict                                              |

### modeci\_mdf.functions.standard.add\_mdf\_function

`modeci_mdf.functions.standard.add_mdf_function` (*name: Optional[str] = None, description: Optional[str] = None, arguments: Optional[List[str]] = None, expression\_string: Optional[str] = None*)

Register a function with MDF function ontology.

Adds a function to the registered list of available MDF functions.

#### Parameters

- **name** – name of the function e.g. 'sin', 'cos', 'linear'
- **description** – Information about the function
- **arguments** – Inputs provided to obtain the result of function
- **expression\_string** – Function expression in string format

**Returns** Updates md\_f\_functions

### modeci\_mdf.functions.standard.create\_python\_expression

`modeci_mdf.functions.standard.create_python_expression` (*expression\_string: Optional[str] = None*) → `str`

Converts the mathematical representation of function into function expression in python

**Parameters** **expression\_string** – Mathematical expression of function in string format

**Returns** function expression in python

### modeci\_mdf.functions.standard.create\_python\_function

`modeci_mdf.functions.standard.create_python_function` (*name: str = None, expression\_string: str = None, arguments: List[str] = None*) → `types.FunctionType`

Create a Python function e.g. linear, exponential, sin, cos, ReLu

#### Parameters

- **name** – name of the function e.g. 'sin', 'cos', 'linear'
- **expression\_string** – Function expression in string format
- **arguments** – list of inputs provided to obtain result from the function

**Returns** A function object

## modeci\_mdf.functions.standard.substitute\_args

`modeci_mdf.functions.standard.substitute_args` (*expression\_string*: *Optional[str]* = None, *args*: *Optional[Dict[str, str]]* = None) → *str*

Substitute arg with the value in args dict

### Parameters

- **expression\_string** – function expression
- **args** – Dictionary of arguments

**Returns** modified expression string after substitution

## 25.4 modeci\_mdf.interfaces

Implementations of importers and exporters for supported environments; fulfilling the [hub and spoke model](#) of MDF by allowing exchange between different modeling environments via MDF.

Most of these exporters and importers are currently a work and progress.

|                                             |                                                            |
|---------------------------------------------|------------------------------------------------------------|
| <code>modeci_mdf.interfaces.actr</code>     | Import and export code for <a href="#">ACT-R</a> models    |
| <code>modeci_mdf.interfaces.graphviz</code> | Import and export code for <a href="#">GraphViz</a> models |
| <code>modeci_mdf.interfaces.onnx</code>     | Import and export code for <a href="#">ONNX</a> models     |
| <code>modeci_mdf.interfaces.pytorch</code>  | Import and export code for <a href="#">PyTorch</a> models  |

### 25.4.1 modeci\_mdf.interfaces.actr

Import and export code for [ACT-R](#) models

|                                                  |                                           |
|--------------------------------------------------|-------------------------------------------|
| <code>modeci_mdf.interfaces.actr.importer</code> | Code for importing ACT-R models into MDF. |
|--------------------------------------------------|-------------------------------------------|

### modeci\_mdf.interfaces.actr.importer

Code for importing ACT-R models into MDF.

### Functions

|                                     |                                                                                                           |
|-------------------------------------|-----------------------------------------------------------------------------------------------------------|
| <code>actr_to_mdf(file_name)</code> | Parses an ACT-R .lisp model file and outputs MDF .json and .yaml files.                                   |
| <code>build_model()</code>          | Builds the base model of the ACT-R production system without any chunks, goals, or productions specified. |

## modeci\_mdf.interfaces.actr.importer.actr\_to\_mdf

`modeci_mdf.interfaces.actr.importer.actr_to_mdf (file_name: str)`

Parses an ACT-R .lisp model file and outputs MDF .json and .yaml files.

**Parameters** `file_name` – The name of the ACT-R model file ending in .lisp.

## modeci\_mdf.interfaces.actr.importer.build\_model

`modeci_mdf.interfaces.actr.importer.build_model ()` → *modeci\_mdf.mdf.Model*

Builds the base model of the ACT-R production system without any chunks, goals, or productions specified.

**Returns** An MDF model object representing the core ACT-R production system.

## 25.4.2 modeci\_mdf.interfaces.graphviz

Import and export code for *GraphViz* models

|                                                      |                                                           |
|------------------------------------------------------|-----------------------------------------------------------|
| <code>modeci_mdf.interfaces.graphviz.exporter</code> | Simple export of MDF to GraphViz for generating graphics. |
|------------------------------------------------------|-----------------------------------------------------------|

## modeci\_mdf.interfaces.graphviz.exporter

Simple export of MDF to GraphViz for generating graphics.

Work in progress...

### Functions

|                                           |
|-------------------------------------------|
| <code>format_bold(s[, use_bold])</code>   |
| <code>format_condition(s)</code>          |
| <code>format_function(s)</code>           |
| <code>format_input(s)</code>              |
| <code>format_label(s)</code>              |
| <code>format_num(s)</code>                |
| <code>format_output(s)</code>             |
| <code>format_param(s)</code>              |
| <code>format_standard_func(s)</code>      |
| <code>format_standard_func_long(s)</code> |

continues on next page

Table 59 – continued from previous page

---

|                                                   |
|---------------------------------------------------|
| <i>format_term_condition</i> (s)                  |
| <i>match_in_expr</i> (expr, node)                 |
| <i>mdf_to_graphviz</i> (mdf_graph[, engine, ...]) |
| <i>safe_comparator</i> (comp)                     |

---

**modeci\_mdf.interfaces.graphviz.exporter.format\_bold**

modeci\_mdf.interfaces.graphviz.exporter.**format\_bold**(s, use\_bold=True)

**modeci\_mdf.interfaces.graphviz.exporter.format\_condition**

modeci\_mdf.interfaces.graphviz.exporter.**format\_condition**(s)

**modeci\_mdf.interfaces.graphviz.exporter.format\_function**

modeci\_mdf.interfaces.graphviz.exporter.**format\_function**(s)

**modeci\_mdf.interfaces.graphviz.exporter.format\_input**

modeci\_mdf.interfaces.graphviz.exporter.**format\_input**(s)

**modeci\_mdf.interfaces.graphviz.exporter.format\_label**

modeci\_mdf.interfaces.graphviz.exporter.**format\_label**(s)

**modeci\_mdf.interfaces.graphviz.exporter.format\_num**

modeci\_mdf.interfaces.graphviz.exporter.**format\_num**(s)

**modeci\_mdf.interfaces.graphviz.exporter.format\_output**

modeci\_mdf.interfaces.graphviz.exporter.**format\_output**(s)

**modeci\_mdf.interfaces.graphviz.exporter.format\_param**

`modeci_mdf.interfaces.graphviz.exporter.format_param(s)`

**modeci\_mdf.interfaces.graphviz.exporter.format\_standard\_func**

`modeci_mdf.interfaces.graphviz.exporter.format_standard_func(s)`

**modeci\_mdf.interfaces.graphviz.exporter.format\_standard\_func\_long**

`modeci_mdf.interfaces.graphviz.exporter.format_standard_func_long(s)`

**modeci\_mdf.interfaces.graphviz.exporter.format\_term\_condition**

`modeci_mdf.interfaces.graphviz.exporter.format_term_condition(s)`

**modeci\_mdf.interfaces.graphviz.exporter.match\_in\_expr**

`modeci_mdf.interfaces.graphviz.exporter.match_in_expr(expr, node)`

**modeci\_mdf.interfaces.graphviz.exporter.mdf\_to\_graphviz**

`modeci_mdf.interfaces.graphviz.exporter.mdf_to_graphviz(mdf_graph, engine='dot',  
output_format='png',  
view_on_render=False,  
level=2, file-  
name_root=None,  
is_horizontal=False,  
solid_color=False)`

**modeci\_mdf.interfaces.graphviz.exporter.safe\_comparator**

`modeci_mdf.interfaces.graphviz.exporter.safe_comparator(comp)`

### 25.4.3 modeci\_mdf.interfaces.onnx

Import and export code for **ONNX** models

---

|                                                  |                                          |
|--------------------------------------------------|------------------------------------------|
| <code>modeci_mdf.interfaces.onnx.exporter</code> | Code for exporting MDF models to ONNX.   |
| <code>modeci_mdf.interfaces.onnx.importer</code> | Code for importing ONNX models into MDF. |

---

**modeci\_mdf.interfaces.onnx.exporter**

Code for exporting MDF models to ONNX.

**Functions**

|                                                   |                                                                                          |
|---------------------------------------------------|------------------------------------------------------------------------------------------|
| <code>convert_mdf_file_to_onnx(input_file)</code> | Converter from MDF to ONNX.                                                              |
| <code>generate_onnx_graph(graph, ...)</code>      |                                                                                          |
| <code>generate_onnx_node(node, graph)</code>      | Convert an MDF node into an ONNX node.                                                   |
| <code>main()</code>                               |                                                                                          |
| <code>mdf_to_onnx(mdf_model)</code>               | Takes an MDF model object and returns a list of ONNX models for each graph in the model. |

**modeci\_mdf.interfaces.onnx.exporter.convert\_mdf\_file\_to\_onnx**

`modeci_mdf.interfaces.onnx.exporter.convert_mdf_file_to_onnx(input_file: str)`

Converter from MDF to ONNX. Takes in a JSON/ONNX file and generates ONNX files.

**Parameters** `input_file` – The input file path to the MDF file. Output files are generated in same directory with -m2o.onnx extensions.

**Returns** `NoneType`

**modeci\_mdf.interfaces.onnx.exporter.generate\_onnx\_graph**

`modeci_mdf.interfaces.onnx.exporter.generate_onnx_graph(graph, node_names_in_execution_order)`

**modeci\_mdf.interfaces.onnx.exporter.generate\_onnx\_node**

`modeci_mdf.interfaces.onnx.exporter.generate_onnx_node(node, graph)`

Convert an MDF node into an ONNX node. Takes an MDF node, MDF graph and returns the ONNX node, any inputs to the node coming from outside the graph, any outputs from the node going outside the graph, and an initializer for constants

**modeci\_mdf.interfaces.onnx.exporter.main**

`modeci_mdf.interfaces.onnx.exporter.main()`

## modeci\_mdf.interfaces.onnx.exporter.mdf\_to\_onnx

`modeci_mdf.interfaces.onnx.exporter.mdf_to_onnx(mdf_model)`

Takes an MDF model object and returns a list of ONNX models for each graph in the model.

## modeci\_mdf.interfaces.onnx.importer

Code for importing ONNX models into MDF.

## Functions

|                                                          |                                                                                     |
|----------------------------------------------------------|-------------------------------------------------------------------------------------|
| <code>convert_file(input_file)</code>                    | Simple converter from ONNX to MDF.                                                  |
| <code>find_subgraphs(graph[, graph_dict])</code>         | Recurse through an ONNX graph and find all sub-graphs.                              |
| <code>get_category_of_onnx_node(entry)</code>            |                                                                                     |
| <code>get_color_for_onnx_category(shape)</code>          |                                                                                     |
| <code>get_onnx_attribute(a)</code>                       |                                                                                     |
| <code>get_shape_params(shape)</code>                     | Small helper function to extract a tuple from the TensorShapeProto.                 |
| <code>id_to_port(id)</code>                              | Turn unique ONNX output and input value names into valid MDF input and output names |
| <code>main()</code>                                      |                                                                                     |
| <code>onnx_node_to_mdf(node, onnx_initializer)</code>    | Construct an MDF node (and function) from an ONNX NodeProto or ValueInfoProto       |
| <code>onnx_to_mdf(onnx_model[, onnx_initializer])</code> | Convert a loaded ONNX model into a MDF model.                                       |

## modeci\_mdf.interfaces.onnx.importer.convert\_file

`modeci_mdf.interfaces.onnx.importer.convert_file(input_file: str)`

Simple converter from ONNX to MDF. Takes in ONNX files and generates MDF JSON/YAML files.

**Parameters** `input_file` – The input file path to the ONNX file. Output files are generated in same directory with -mdf.json and -mdf.yml extensions.

**Returns** MoneType

## modeci\_mdf.interfaces.onnx.importer.find\_subgraphs

`modeci_mdf.interfaces.onnx.importer.find_subgraphs(graph:`

`onnx.onnx_ml_pb2.GraphProto,`  
`graph_dict: Optional[Dict[str,`  
`onnx.onnx_ml_pb2.GraphProto]]`  
`= None) → Dict[str,`  
`onnx.onnx_ml_pb2.GraphProto]`

Recurse through an ONNX graph and find all subgraphs.

**Parameters**



- **graph** – The graph to search.
- **graph\_list** – Insert graphs we find into this dict. Use the parent node name as a key. If None, initialize to empty dict.

**Returns** All the subgraphs in the for the graph.

#### **modeci\_mdf.interfaces.onnx.importer.get\_category\_of\_onnx\_node**

`modeci_mdf.interfaces.onnx.importer.get_category_of_onnx_node(entry)`

#### **modeci\_mdf.interfaces.onnx.importer.get\_color\_for\_onnx\_category**

`modeci_mdf.interfaces.onnx.importer.get_color_for_onnx_category(shape)`

#### **modeci\_mdf.interfaces.onnx.importer.get\_onnx\_attribute**

`modeci_mdf.interfaces.onnx.importer.get_onnx_attribute(a)`

#### **modeci\_mdf.interfaces.onnx.importer.get\_shape\_params**

`modeci_mdf.interfaces.onnx.importer.get_shape_params(shape: onnx.onnx_ml_pb2.TensorShapeProto) → Tuple`

Small helper function to extract a tuple from the TensorShapeProto. These objects can contain both integer dimensions and parameter dimensions that are variable, like ‘batch\_size’.

**Parameters** **shape** – The ONNX shape proto to process.

**Returns** A tuple that can contain both integers and strings for parameter dimensions.

#### **modeci\_mdf.interfaces.onnx.importer.id\_to\_port**

`modeci_mdf.interfaces.onnx.importer.id_to_port(id: str)`

Turn unique ONNX output and input value names into valid MDF input and output names

#### **modeci\_mdf.interfaces.onnx.importer.main**

`modeci_mdf.interfaces.onnx.importer.main()`

**modeci\_mdf.interfaces.onnx.importer.onnx\_node\_to\_mdf**

```
modeci_mdf.interfaces.onnx.importer.onnx_node_to_mdf (node:
 Union[onnx.onnx_ml_pb2.NodeProto,
onnx.onnx_ml_pb2.ValueInfoProto],
onnx_initializer: Dict[str,
Dict[str, Any]]) → modeci_mdf.mdf.Node
```

Construct an MDF node (and function) from an ONNX NodeProto or ValueInfoProto

**Parameters**

- **node** – The ONNX node to use to form the MDF node. Can be a node from the model or a ValueInfoProto specifying an input or output.
- **onnx\_initializer** – A specification of values in the graph that ONNX has marked as initializer's. This dict is keyed on the name of the parameter, the value is another dict with three entries; shape, type, and value.

**Returns** The equivalent MDF node for the ONNX node passed in as argument.

**modeci\_mdf.interfaces.onnx.importer.onnx\_to\_mdf**

```
modeci_mdf.interfaces.onnx.importer.onnx_to_mdf (onnx_model:
 Union[onnx.onnx_ml_pb2.ModelProto,
onnx.onnx_ml_pb2.GraphProto],
onnx_initializer: Optional[Dict[str,
Dict[str, Any]]] = None)
```

Convert a loaded ONNX model into a MDF model.

**Parameters**

- **onnx\_model** – The ONNX model to convert. Typically, this is the result of a call to `onnx.load()`
- **onnx\_initializer** – A specification of values in the graph that ONNX has marked as initializer's. This dict is keyed on the name of the parameter, the value is another dict with three entries; shape, type, and value.

**Returns** An MDF description of the ONNX model.

**25.4.4 modeci\_mdf.interfaces.pytorch**

Import and export code for [PyTorch](#) models

|                                                     |                                                                  |
|-----------------------------------------------------|------------------------------------------------------------------|
| <code>modeci_mdf.interfaces.pytorch.exporter</code> | Functions for converting from MDF models to PyTorch              |
| <code>modeci_mdf.interfaces.pytorch.importer</code> | Functions for converting from PyTorch TorchScript to MDF models. |

## modeci\_mdf.interfaces.pytorch.exporter

Functions for converting from MDF models to PyTorch

### Functions

|                                                                |                                                                                                                |
|----------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------|
| <code>build_script(nodes, execution_order, ...)</code>         | Helper function to create and assemble text components necessary to specify module.py importable model script. |
| <code>func_args(exp, arg_dict)</code>                          |                                                                                                                |
| <code>generate_main_forward(nodes, ...)</code>                 | Helper function to generate the main forward method that will specify the execution of the pytorch model.      |
| <code>get_module_declaration_text(name, node_dict, ...)</code> | Helper function to generate string in module.py.                                                               |
| <code>mdf_to_pytorch(mdf_model, model_input, ...)</code>       | Function loads and returns a pytorch model for all models specified in an mdf file.                            |
| <code>sym(value)</code>                                        |                                                                                                                |

## modeci\_mdf.interfaces.pytorch.exporter.build\_script

`modeci_mdf.interfaces.pytorch.exporter.build_script` (*nodes*: *List[node]*, *execution\_order*: *List[str]*, *model\_id1*: *str*, *d\_e*: *Dict[str, Any]*, *conditions*, *version*: *str*)

Helper function to create and assemble text components necessary to specify module.py importable model script. These include:

- **Module declarations**
  - Initialization of functions
  - Definition of forward function
- **Model main call declaration:**
  - Initialization of subcomponents
  - Forward function logic

#### Parameters

- **nodes** – list of nodes in the graph
- **execution\_order** – List of nodes in the order of execution
- **d\_e** – Weights on edges stored in dict format
- **model\_id1** – id of the model to be converted
- **conditions** –

Returns: complete module.py script as a formatted string

**modeci\_mdf.interfaces.pytorch.exporter.func\_args**

`modeci_mdf.interfaces.pytorch.exporter.func_args` (*exp*, *arg\_dict*)

**modeci\_mdf.interfaces.pytorch.exporter.generate\_main\_forward**

`modeci_mdf.interfaces.pytorch.exporter.generate_main_forward` (*nodes*: *List*[*node*],  
*execution\_order*:  
*List*[*str*], *d\_e*:  
*Dict*[*str*, *Any*])

Helper function to generate the main forward method that will specify the execution of the pytorch model. This requires proper ordering of module calls as well as preservation of variables.

**Parameters**

- **nodes** – list of nodes in the graph
- **execution\_order** – List of nodes in the order of execution
- **d\_e** – Weights on edges stored in dict format

Returns: Function returns the main forward call that will be used to define pytorch model

**modeci\_mdf.interfaces.pytorch.exporter.get\_module\_declaration\_text**

`modeci_mdf.interfaces.pytorch.exporter.get_module_declaration_text` (*name*: *str*,  
*node\_dict*:  
*Dict*[*Any*,  
*Any*],  
*execution\_order*:  
*List*[*str*],  
*version*:  
*str*)

Helper function to generate string in module.py. String will create an instance of the torch object corresponding to the node and function at node as method. Generated text specifies the definition of class that will form the pytorch model, including `__init__` method and forward method. Returns string of the class definition with parameters and arguments assigned.

**Parameters**

- **name** – Name of the node
- **node\_dict** – dictionary with attributes of the node such as Input Ports, functions, parameters and
- **execution\_order** – List of nodes in the order of execution

Returns: Script in PyTorch schema

## modeci\_mdf.interfaces.pytorch.exporter.mdf\_to\_pytorch

```
modeci_mdf.interfaces.pytorch.exporter.mdf_to_pytorch(mdf_model: Model,
 model_input: str,
 eval_models: bool,
 version: str)
```

Function loads and returns a pytorch model for all models specified in an mdf file.

### Parameters

- **mdf\_model** – model in MDF format
- **eval\_models** – Set Evaluation of model to True or False
- **version** – MDF version
- **model\_input** – input file name

Returns: Returns a dictionary where key = model name, value = pytorch model object

## modeci\_mdf.interfaces.pytorch.exporter.sym

```
modeci_mdf.interfaces.pytorch.exporter.sym(value)
```

## modeci\_mdf.interfaces.pytorch.importer

Functions for converting from PyTorch TorchScript to MDF models.

This code was originally inspired by the following blog post:

Mike He, “From Models to Computation Graphs (Part I)”, <https://ad1024.space/articles/22>

## Functions

|                                                             |         |                                                                                                    |
|-------------------------------------------------------------|---------|----------------------------------------------------------------------------------------------------|
| <code>convert_to_serializable(value)</code>                 |         | Helper function that converts some common unserializable types to JSON serializable types          |
| <code>get_graph_constants(graph)</code>                     |         | Find all constant nodes in the graph and extract their values as a proper JSON serializable value. |
| <code>make_func_id(node)</code>                             |         | Helper function to get a unique name (used in MDF as id) for a TorchScript node’s op/function.     |
| <code>make_model_graph_name(model)</code>                   |         | Helper function that generates a clean graph and model name from a TorchScript model               |
| <code>make_node_id(node)</code>                             |         | Helper function to get a unique name (used in MDF as id) from a TorchScript Node object            |
| <code>process_onnx_schema(node, port_mapper)</code>         | consts, | Retrieve the argument names and attributes (parameters in MDF) for this Operation.                 |
| <code>process_torch_schema(node, port_mapper)</code>        | consts, | Parse a TorchScript node schema into argument names and constant attributes (parameters in MDF)    |
| <code>pytorch_to_mdf(model[, args, trace, ...])</code>      |         | Convert a PyTorch model to an MDF model.                                                           |
| <code>torchnode_to_mdftype(node, graph, consts, ...)</code> |         | Convert a TorchScript node to an MDF node.                                                         |
| <code>translate_graph(graph, mdf_graph, consts, ...)</code> |         | Go through a Graph or Block and translate the nodes and edges to MDF nodes and edges.              |

**modeci\_mdf.interfaces.pytorch.importer.convert\_to\_serializable**

`modeci_mdf.interfaces.pytorch.importer.convert_to_serializable` (*value*)

Helper function that converts some common unserializable types to JSON serializable types

**modeci\_mdf.interfaces.pytorch.importer.get\_graph\_constants**

`modeci_mdf.interfaces.pytorch.importer.get_graph_constants` (*graph*:  
*torch.\_C.Graph*)  
→ Dict[str, Any]

Find all constant nodes in the graph and extract their values as a proper JSON serializable value.

**Parameters** **graph** – The graph to extract constants from.

**Returns** A Dict that maps the constant nodes unique TorchScript node ID string to its value.

**modeci\_mdf.interfaces.pytorch.importer.make\_func\_id**

`modeci_mdf.interfaces.pytorch.importer.make_func_id` (*node*: *torch.\_C.Node*) → str

Helper function to get a unique name (used in MDF as id) for a TorchScript node's op/function.

**modeci\_mdf.interfaces.pytorch.importer.make\_model\_graph\_name**

`modeci_mdf.interfaces.pytorch.importer.make_model_graph_name` (*model*:  
*Union[torch.\_C.ScriptModule,*  
*torch.jit.ScriptFunction]*)  
→ Tuple[str, str]

Helper function that generates a clean graph and model name from a TorchScript model

**modeci\_mdf.interfaces.pytorch.importer.make\_node\_id**

`modeci_mdf.interfaces.pytorch.importer.make_node_id` (*node*: *torch.\_C.Node*) → str

Helper function to get a unique name (used in MDF as id) from a TorchScript Node object

**modeci\_mdf.interfaces.pytorch.importer.process\_onnx\_schema**

`modeci_mdf.interfaces.pytorch.importer.process_onnx_schema` (*node*: *torch.\_C.Node*,  
*consts*: Dict,  
*port\_mapper*: *modeci\_mdf.interfaces.pytorch.importer.PortMapper*)  
→ Tuple[Dict[str, str],  
Dict[str, Any]]

Retrieve the argument names and attributes (parameters in MDF) for this Operation.

**Parameters**

- **op** – The TorchScript node containing the ONNX operation.
- **port\_mapper** – The utility class for assigning TorchScript input output ids to Input Output Port ids.

**Returns**

- A dict representing argument names mapping to input port ids
- A dict mapping parameters (ONNX attributes) names mapping to values

**Return type** A two element tuple

### modeci\_mdf.interfaces.pytorch.importer.process\_torch\_schema

```
modeci_mdf.interfaces.pytorch.importer.process_torch_schema (node:
 torch._C.Node,
 consts: Dict,
 port_mapper: modeci_mdf.interfaces.pytorch.importer.PortMapper)
 → Tuple[List[str],
 Dict[str, Any]]
```

Parse a TorchScript node schema into argument names and constant attributes (parameters in MDF)

#### Parameters

- **node** – The TorchScript node to retrieve the schema from.
- **consts** – The constant nodes Dict for the graph we are working with.

**Returns** A tuple containing a list of argument names and Dict of parameter names and values.

### modeci\_mdf.interfaces.pytorch.importer.pytorch\_to\_mdf

```
modeci_mdf.interfaces.pytorch.importer.pytorch_to_mdf (model: Union[Callable,
 torch.nn.modules.module.Module,
 torch.jit.ScriptFunction,
 torch._C.ScriptModule],
 args: Union[None,
 torch.Tensor,
 Tuple[torch.Tensor]] =
 None,
 trace: bool =
 False,
 use_onnx_ops:
 bool =
 True) →
 Union[modeci_mdf.mdf.Model,
 modeci_mdf.mdf.Graph]
```

Convert a PyTorch model to an MDF model. By default, this function will invoke `torch.jit.script` on the model to compile it down to TorchScript IR and simplify the graph before exporting the MDF. The default is to use ONNX operations when possible and fallback to ATENTorch ops when ONNX support is not available (`torch._C._onnx.OperatorExportTypes.ONNX_ATEN_FALLBACK` mode). To use all ATENTorch ops, set `use_onnx_ops` to False.

#### Parameters

- **model** – The model to translate into MDF.
- **args** – The input arguments for this model. If a `nn.Module` is passed then the model will be traced with these inputs. If a `ScriptModule` is passed, they are still needed to determine input shapes.
- **trace** – Force the use of tracing to compile the model. The default is to use `torch.jit.script`
- **use\_onnx\_ops** – Use ONNX ops when possible, fallback to ATEN ops when not available. Default is True. If False, use only ATEN ops.

**Returns** The translated MDF model

### modeci\_mdf.interfaces.pytorch.importer.torchnode\_to\_mdfnode

```
modeci_mdf.interfaces.pytorch.importer.torchnode_to_mdfnode (node:
 torch._C.Node,
 graph:
 torch._C.Graph,
 consts:
 Dict[str, Any],
 port_mapper: modeci_mdf.interfaces.pytorch.importer.PortMapper)
 →
 Optional[modeci_mdf.mdf.Node]
```

Convert a TorchScript node to an MDF node.

#### Parameters

- **node** – The node to convert.
- **graph** – The graph that this node is a member.
- **consts** – A dict containing any constants in the graph.

**Returns** The MDF node for this TorchScript node. `prim::Constant` nodes are excluded from the MDF graph and are instead placed as parameters. In this case, return `None`.

### modeci\_mdf.interfaces.pytorch.importer.translate\_graph

```
modeci_mdf.interfaces.pytorch.importer.translate_graph (graph:
 Union[torch._C.Graph,
 torch._C.Block],
 mdf_graph: modeci_mdf.mdf.Graph,
 consts: Dict[str, Any],
 port_mapper: modeci_mdf.interfaces.pytorch.importer.PortMapper)
```

Go through a `Graph` or `Block` and translate the nodes and edges to MDF nodes and edges.

#### Parameters

- **graph** – The graph to translate.
- **mdf\_graph** – The MDF graph to store the translation into.
- **consts** – Constant to use for parameters of nodes.
- **port\_mapper** – A port mapper instance to handle translating names.

Returns:



## Classes

---

|                                 |                                                                                                 |
|---------------------------------|-------------------------------------------------------------------------------------------------|
| <i>PortMapper</i> (graph, args) | A simple class that handles mapping TorchScript inputoutput ids to MDF InputPortOutputPort ids. |
|---------------------------------|-------------------------------------------------------------------------------------------------|

---

### modeci\_mdf.interfaces.pytorch.importer.PortMapper

**class** modeci\_mdf.interfaces.pytorch.importer.**PortMapper** (*graph:* *torch.\_C.Graph*,  
*args:* *Tuple*)

Bases: `object`

A simple class that handles mapping TorchScript inputoutput ids to MDF InputPortOutputPort ids. It keeps track of annoying details like graph level inputs and stuff.

#### Methods

---

|                          |                                                                                            |
|--------------------------|--------------------------------------------------------------------------------------------|
| <i>id_to_port</i> (id)   | Turn unique TorchScript output and input value names into valid MDF input and output names |
| <i>port_to_id</i> (name) | Transform a port name back to is TorchScript ID                                            |

---

**id\_to\_port** (*id:* *str*)

Turn unique TorchScript output and input value names into valid MDF input and output names

**port\_to\_id** (*name:* *str*)

Transform a port name back to is TorchScript ID

## 25.5 modeci\_mdf.mdf

The main object-oriented implementation of the MDF schema, with each core component of the [MDF specification](#) implemented as a `class`. Instances of these objects can be composed to create a representation of an MDF model as Python objects. These models can then be serialized and deserialized to and from JSON or YAML, executed via the *execution\_engine* module, or imported and exported to supported external environments using the *interfaces* module.

### Functions

---

*parsed\_structure\_factory*(cl)

---

*parsed\_unstructure\_factory*(cl)

---

*v*(cl)

---

### 25.5.1 modeci\_mdf.mdf.parsed\_structure\_factory

`modeci_mdf.mdf.parsed_structure_factory(cl)`

### 25.5.2 modeci\_mdf.mdf.parsed\_unstructure\_factory

`modeci_mdf.mdf.parsed_unstructure_factory(cl)`

### 25.5.3 modeci\_mdf.mdf.v

`modeci_mdf.mdf.v(cl)`

## Classes

|                                                                |                                                                                                                                                                                                                                                       |
|----------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>Condition(type, **kwargs, Any[])</code>                  | A set of descriptors which specifies conditional execution of Nodes to meet complex execution requirements.                                                                                                                                           |
| <code>ConditionSet(node_specific, ...)</code>                  | Specifies the non-default pattern of execution of Nodes                                                                                                                                                                                               |
| <code>Edge(id, sender, receiver, sender_port, ...)</code>      | An <i>Edge</i> is an attribute of a <i>Graph</i> that transmits computational results from a sender's <i>OutputPort</i> to a receiver's <i>InputPort</i> .                                                                                            |
| <code>Function(id, function, args, Any[] = None, ...)</code>   | A single value which is evaluated as a function of values on :class:`InputPort` (s) and other Functions                                                                                                                                               |
| <code>Graph(id, nodes, edges, parameters, ...)</code>          | A directed graph consisting of :class:`~Node`'s (with :class:`~Parameter`'s and :class:`~Function`'s evaluated internally) connected via :class:`~Edge`'s.                                                                                            |
| <code>InputPort(id[, shape])</code>                            | The <i>InputPort</i> is an attribute of a Node which allows external information to be input to the Node                                                                                                                                              |
| <code>Model(id, graphs, format, ...)</code>                    | The top level construct in MDF is Model, which may contain multiple <i>Graph</i> objects and model attribute(s)                                                                                                                                       |
| <code>Node(id, input_ports, functions, parameters, ...)</code> | A self contained unit of evaluation receiving input from other nodes on :class:`InputPort` (s). The values from these are processed via a number of :class:`Function` (s) and one or more final values are calculated on the :class:`OutputPort` (s). |
| <code>OutputPort(id, value[, shape])</code>                    | The <i>OutputPort</i> is an attribute of a <i>Node</i> which exports information to another <i>Node</i> connected by an <i>Edge</i>                                                                                                                   |
| <code>Parameter(id, value, List, Dict, ...)</code>             | A parameter of the <i>Node</i> , which can be: 1) a specific fixed value (a constant (int/float) or an array) 2) a string expression for the value referencing other named :class:`Parameter` (s).                                                    |
| <code>ParameterCondition(id, test, List, Dict, ...)</code>     | A condition to test on a Node's parameters, which if true, sets the value of this Parameter                                                                                                                                                           |

## 25.5.4 modeci\_mdf.mdf.Condition

```
class modeci_mdf.mdf.Condition (type: Optional[str] = None, **kwargs: Optional[Dict[str, Any]])
```

Bases: modeci\_mdf.mdf.MdfBase

A set of descriptors which specifies conditional execution of Nodes to meet complex execution requirements.

### Variables

- **type** (*str*) – The type of *Condition* from the library
- **kwargs** (*Optional[Dict[str, Any]]*) – The dictionary of keyword arguments needed to evaluate the *Condition*

Method generated by attrs for class MdfBase.

### Methods

---

## 25.5.5 modeci\_mdf.mdf.ConditionSet

```
class modeci_mdf.mdf.ConditionSet (node_specific: Optional[Dict[str, modeci_mdf.mdf.Condition]] = None, termination: Optional[Dict[str, modeci_mdf.mdf.Condition]] = None, *, metadata: Optional[Dict[str, Any]] = None)
```

Bases: modeci\_mdf.mdf.MdfBase

Specifies the non-default pattern of execution of Nodes

### Variables

- **node\_specific** (*Optional[Dict[str, modeci\_mdf.mdf.Condition]]*) – A dictionary mapping nodes to any non-default run conditions
- **termination** (*Optional[Dict[str, modeci\_mdf.mdf.Condition]]*) – A dictionary mapping time scales of model execution to conditions indicating when they end

Method generated by attrs for class ConditionSet.

### Methods

---

## 25.5.6 modeci\_mdf.mdf.Edge

```
class modeci_mdf.mdf.Edge (id: str, sender: str, receiver: str, sender_port: str, receiver_port: str, parameters: Optional[Dict[str, Any]] = None, *, metadata: Optional[Dict[str, Any]] = None)
```

Bases: modeci\_mdf.mdf.MdfBase

An *Edge* is an attribute of a *Graph* that transmits computational results from a sender's *OutputPort* to a receiver's *InputPort*.

### Variables

- **id** (*str*) – A unique string identifier for this edge.

- **sender** (*str*) – The id of the *Node* which is the source of the edge.
- **receiver** (*str*) – The id of the *Node* which is the target of the edge.
- **sender\_port** (*str*) – The id of the *OutputPort* on the sender *Node*, whose value should be sent to the *receiver\_port*
- **receiver\_port** (*str*) – The id of the *InputPort* on the receiver *Node*
- **parameters** (*Optional[Dict[str, Any]]*) – Dictionary of parameters for the edge.

Method generated by attrs for class Edge.

## Methods

### 25.5.7 modeci\_mdf.mdf.Function

```
class modeci_mdf.mdf.Function (id: str, function: Optional[str] = None, args:
 Optional[Dict[str, Any]] = None, value: Optional[Union[modelspec.base_types.EvaluableExpression, List,
 Dict, numpy.ndarray, int, float, str]] = None, *, metadata:
 Optional[Dict[str, Any]] = None)
```

Bases: modeci\_mdf.mdf.MdfBase

A single value which is evaluated as a function of values on :class:`InputPort` (s) and other Functions

#### Variables

- **id** (*str*) – The unique (for this Node) id of the function, which will be used in other :class:`~Function`'s and the :class:`~OutputPort`'s for its value
- **function** (*Optional[str]*) – Which of the in-build MDF functions (linear, etc.). See supported functions: [https://mdf.readthedocs.io/en/latest/api/MDF\\_function\\_specifications.html](https://mdf.readthedocs.io/en/latest/api/MDF_function_specifications.html)
- **args** (*Optional[Dict[str, Any]]*) – Dictionary of values for each of the arguments for the Function, e.g. if the in-built function is linear(slope), the args here could be {"slope":3} or {"slope":"input\_port\_0 + 2"}
- **value** (*Optional[Union[modelspec.base\_types.EvaluableExpression, List, Dict, numpy.ndarray, int, float, str]]*) – If the function is a value expression, this attribute will contain the expression and the function and args attributes will be None.

Method generated by attrs for class Function.

## Methods

### 25.5.8 modeci\_mdf.mdf.Graph

```
class modeci_mdf.mdf.Graph(id: str, nodes: List[modeci_mdf.mdf.Node] = NOTHING,
 edges: List[modeci_mdf.mdf.Edge] = NOTHING, parameters: Optional[Dict[str, Any]] = None, conditions: Optional[modeci_mdf.mdf.ConditionSet] = None, *, metadata: Optional[Dict[str, Any]] = None)
```

Bases: modeci\_mdf.mdf.MdfBase

A directed graph consisting of :class:`~Node`'s (with :class:`~Parameter`'s and :class:`~Function`'s evaluated internally) connected via :class:`~Edge`'s.

#### Variables

- **id** (*str*) – A unique identifier for this Graph
- **nodes** (*List*[modeci\_mdf.mdf.Node]) – One or more :class:`~Node`'s present in the graph
- **edges** (*List*[modeci\_mdf.mdf.Edge]) – Zero or more :class:`~Edge`'s present in the graph
- **parameters** (*Optional*[*Dict*[*str*, *Any*]]) – Dictionary of global parameters for the Graph
- **conditions** (*Optional*[modeci\_mdf.mdf.ConditionSet]) – The Condition-Set stored as dictionary for scheduling of the Graph

Method generated by attrs for class Graph.

## Methods

---

|                           |                                                    |
|---------------------------|----------------------------------------------------|
| <code>get_node(id)</code> | Retrieve Node object corresponding to the given id |
|---------------------------|----------------------------------------------------|

---

**get\_node** (*id*: *str*) → *Optional*[modeci\_mdf.mdf.Node]

Retrieve Node object corresponding to the given id

**Parameters** **id** – Unique identifier of Node object

**Returns** *Node* object if the entered *id* matches with the *id* of node present in the *Graph*.  
None if a node is not found with that *id*.

**property dependency\_dict**

Returns the dependency among nodes as dictionary

Key: receiver, Value: Set of senders imparting information to the receiver

**Returns** Returns the dependency dictionary

**property inputs**

Enumerate all Node-InputPort pairs that specify no incoming edge. These are input ports for the graph itself and must be provided values to evaluate

**Returns** A list of Node, InputPort tuples

### 25.5.9 modeci\_mdf.mdf.InputPort

```
class modeci_mdf.mdf.InputPort (id: str, shape=None, type: Optional[str] = None, *, metadata: Optional[Dict[str, Any]] = None)
```

Bases: modeci\_mdf.mdf.MdfBase

The *InputPort* is an attribute of a Node which allows external information to be input to the Node

#### Variables

- **id** (*str*) – The unique (for this Node) id of the input port,
- **shape** (*Optional[Tuple[int, ...]]*) – The shape of the input port. This uses the same syntax as numpy ndarray shapes (e.g., `numpy.zeros(shape)` would produce an array with the correct shape
- **type** (*Optional[str]*) – The data type of the input received at a port.

Method generated by attrs for class InputPort.

#### Methods

---

### 25.5.10 modeci\_mdf.mdf.Model

```
class modeci_mdf.mdf.Model (id: str, graphs: List[modeci_mdf.mdf.Graph] = NOTHING, format: str = 'ModECI MDF v0.4', generating_application: str = 'Python modeci-mdf v0.4.4', onnx_opset_version: Optional[str] = None, *, metadata: Optional[Dict[str, Any]] = None)
```

Bases: modeci\_mdf.mdf.MdfBase

The top level construct in MDF is Model, which may contain multiple *Graph* objects and model attribute(s)

#### Variables

- **id** (*str*) – A unique identifier for this Model
- **graphs** (*List[modeci\_mdf.mdf.Graph]*) – The collection of graphs that make up the MDF model.
- **format** (*str*) – Information on the version of MDF used in this file
- **generating\_application** (*str*) – Information on what application generated/saved this file
- **onnx\_opset\_version** (*Optional[str]*) – The ONNX opset used for any ONNX functions in this model.

Method generated by attrs for class Model.

## Methods

---

|                                                           |                                                                      |
|-----------------------------------------------------------|----------------------------------------------------------------------|
| <code>to_graph_image([engine, output_format, ...])</code> | Convert MDF graph to an image (png or svg) using the Graphviz export |
|-----------------------------------------------------------|----------------------------------------------------------------------|

---

**to\_graph\_image** (*engine: str = 'dot', output\_format: str = 'png', view\_on\_render: bool = False, level: int = 2, filename\_root: Optional[str] = None, only\_warn\_on\_fail: bool = False, is\_horizontal: bool = False, solid\_color=False*)  
 Convert MDF graph to an image (png or svg) using the Graphviz export

### Parameters

- **engine** – dot or other Graphviz formats
- **output\_format** – e.g. png (default) or svg
- **view\_on\_render** – if True, will open generated image in system viewer
- **level** – 1,2,3, depending on how much detail to include
- **filename\_root** – will change name of file generated to filename\_root.png, etc.
- **only\_warn\_on\_fail** – just give a warning if this fails, e.g. no dot executable. Useful for preventing errors in automated tests
- **solid\_color** – if True, will return a solid color for the graphviz node. default is False

## 25.5.11 modeci\_mdf.mdf.Node

**class** modeci\_mdf.mdf.Node (*id: str, input\_ports: List[modeci\_mdf.mdf.InputPort] = NOTHING, functions: List[modeci\_mdf.mdf.Function] = NOTHING, parameters: List[modeci\_mdf.mdf.Parameter] = NOTHING, output\_ports: List[modeci\_mdf.mdf.OutputPort] = NOTHING, \*, metadata: Optional[Dict[str, Any]] = None*)

Bases: modeci\_mdf.mdf.MdfBase

A self contained unit of evaluation receiving input from other nodes on :class:`InputPort` (s). The values from these are processed via a number of :class:`Function` (s) and one or more final values are calculated on the :class:`OutputPort` (s)

### Variables

- **id** (*str*) – A unique identifier for the node.
- **input\_ports** (*List [modeci\_mdf.mdf.InputPort]*) – Dictionary of the *InputPort* objects in the Node
- **parameters** (*List [modeci\_mdf.mdf.Parameter]*) – Dictionary of :class:`Parameter` (s) for the node
- **functions** (*List [modeci\_mdf.mdf.Function]*) – The :class:`Function` (s) for computation the node
- **output\_ports** (*List [modeci\_mdf.mdf.OutputPort]*) – The :class:`OutputPort` (s) containing evaluated quantities from the node

Method generated by attrs for class Node.

## Methods

|                                  |                                                                                                                                 |
|----------------------------------|---------------------------------------------------------------------------------------------------------------------------------|
| <code>get_input_port(id)</code>  | Retrieve <i>InputPort</i> object corresponding to the given id :param id: Unique identifier of class: <i>InputPort</i> object   |
| <code>get_output_port(id)</code> | Retrieve <i>OutputPort</i> object corresponding to the given id :param id: Unique identifier of class: <i>OutputPort</i> object |
| <code>get_parameter(id)</code>   | Get a parameter by its string id                                                                                                |

**get\_parameter** (*id: str*) → *Optional[modeci\_mdf.mdf.Parameter]*

Get a parameter by its string id

**Parameters** *id* – The unique string id of the *Parameter*

**Returns** The *Parameter* object stored on this node. None if not found.

**get\_input\_port** (*id: str*) → *modeci\_mdf.mdf.InputPort*

Retrieve *InputPort* object corresponding to the given id :param id: Unique identifier of class:*InputPort* object

**Returns** *InputPort* object if the entered id matches with the id of class:*InputPort* present in the class:*Node*

**Return type** class

**get\_output\_port** (*id: str*) → *modeci\_mdf.mdf.OutputPort*

Retrieve *OutputPort* object corresponding to the given id :param id: Unique identifier of class:*OutputPort* object

**Returns** *OutputPort* object if the entered id matches with the id of class:*OutputPort* present in the class:*Node*

**Return type** class

### 25.5.12 modeci\_mdf.mdf.OutputPort

**class** *modeci\_mdf.mdf.OutputPort* (*id: str, value: Optional[str] = None, shape=None, type: Optional[str] = None, \*, metadata: Optional[Dict[str, Any]] = None*)

Bases: *modeci\_mdf.mdf.MdfBase*

The *OutputPort* is an attribute of a *Node* which exports information to another *Node* connected by an *Edge*

#### Variables

- **id** (*str*) – Unique identifier for the output port.
- **value** (*Optional[str]*) – The value of the *OutputPort* in terms of the *InputPort*, *Function* values, and *Parameter* values.
- **shape** (*Optional[Tuple[int, ...]]*) – The shape of the output port. This uses the same syntax as numpy ndarray shapes (e.g., `numpy.zeros(shape)` would produce an array with the correct shape
- **type** (*Optional[str]*) – The data type of the output sent by a port.

Method generated by attrs for class *OutputPort*.



## Methods

## 25.5.13 modeci\_mdf.mdf.Parameter

```
class modeci_mdf.mdf.Parameter (id: str, value: Optional[Union[modelspec.base_types.EvaluableExpression,
List, Dict, numpy.ndarray, int, float, str]] = None, default_initial_value: Optional[Union[modelspec.base_types.EvaluableExpression, List,
Dict, numpy.ndarray, int, float, str]] = None, time_derivative: Optional[str] = None, function: Optional[str] = None,
args: Optional[Dict[str, Any]] = None, conditions: List[modeci_mdf.mdf.ParameterCondition] = NOTHING,
*, metadata: Optional[Dict[str, Any]] = None)
```

Bases: modeci\_mdf.mdf.MdfBase

A parameter of the [Node](#), which can be: 1) a specific fixed value (a constant (int/float) or an array) 2) a string expression for the value referencing other named Parameter` (s). which may be stateful (i.e. can change value over multiple executions of the :class:`Node`); 3) be evaluated by an inbuilt function with args; 4) or change from a default\_initial\_value with a time\_derivative.

## Variables

- **value** (Optional[Union[modelspec.base\_types.EvaluableExpression, List, Dict, numpy.ndarray, int, float, str]]) – The next value of the parameter, in terms of the inputs, functions and PREVIOUS parameter values
- **default\_initial\_value** (Optional[Union[modelspec.base\_types.EvaluableExpression, List, Dict, numpy.ndarray, int, float, str]]) – The initial value of the parameter, only used when parameter is stateful.
- **time\_derivative** (Optional[str]) – How the parameter changes with time, i.e. ds/dt. Units of time are seconds.
- **function** (Optional[str]) – Which of the in-built MDF functions (linear etc.) this uses, See
- **https** – [//mdf.readthedocs.io/en/latest/api/MDF\\_function\\_specifications.html](https://mdf.readthedocs.io/en/latest/api/MDF_function_specifications.html)
- **args** (Optional[Dict[str, Any]]) – Dictionary of values for each of the arguments for the function of the parameter, e.g. if the in-built function is `linear(slope)`, the args here could be `{"slope": 3}` or `{"slope": "input_port_0 + 2"}`
- **conditions** (List[modeci\_mdf.mdf.ParameterCondition]) – Parameter specific conditions

Method generated by attrs for class Parameter.

## Methods

|                            |                               |
|----------------------------|-------------------------------|
| <code>is_stateful()</code> | Is the parameter stateful?    |
| <code>summary()</code>     | Short summary of Parameter... |

**summary()**

Short summary of Parameter...

**is\_stateful()** → bool

Is the parameter stateful?

A parameter is considered stateful if it has a `time_derivative`, `default_initial_value`, or its `id` is referenced in its value expression.

**Returns** True if stateful, False if not.

### 25.5.14 modeci\_mdf.mdf.ParameterCondition

```
class modeci_mdf.mdf.ParameterCondition(id: str, test: Optional[Union[modelspec.base_types.EvaluableExpression,
List, Dict, numpy.ndarray, int, float, str]] = None, value: Optional[Union[modelspec.base_types.EvaluableExpression,
List, Dict, numpy.ndarray, int, float, str]] = None)
```

Bases: `modelspec.base_types.Base`

A condition to test on a Node's parameters, which if true, sets the value of this Parameter

#### Variables

- **id** (`str`) – A unique identifier for the ParameterCondition
- **test** (`Optional[Union[modelspec.base_types.EvaluableExpression, List, Dict, numpy.ndarray, int, float, str]]`) – The boolean expression to evaluate
- **value** (`Optional[Union[modelspec.base_types.EvaluableExpression, List, Dict, numpy.ndarray, int, float, str]]`) – The new value of the Parameter if the test is true

Method generated by attrs for class ParameterCondition.

## Methods

## 25.6 modeci\_mdf.utils

Useful utility functions for dealing with MDF objects.

### Functions

|                                                         |                                                                          |
|---------------------------------------------------------|--------------------------------------------------------------------------|
| <code>color_rgb_to_hex(rgb)</code>                      | Convert a rgb color to hexadecimal format.                               |
| <code>create_example_node(node_id, graph)</code>        | Create a simple example node with Input inside a graph                   |
| <code>is_number(s)</code>                               | Return True if cast to float does not throw ValueError, False otherwise. |
| <code>load_mdf(filename)</code>                         | Load an MDF file from JSON or YAML.                                      |
| <code>load_mdf_json(filename)</code>                    | Load an MDF JSON file                                                    |
| <code>load_mdf_yaml(filename)</code>                    | Load an MDF YAML file                                                    |
| <code>print_summary(graph)</code>                       | Print a summary of a graph to standard out.                              |
| <code>simple_connect(pre_node, post_node, graph)</code> | Create an edge connecting two nodes in a graph.                          |

### 25.6.1 modeci\_mdf.utils.color\_rgb\_to\_hex

`modeci_mdf.utils.color_rgb_to_hex(rgb)`  
Convert a rgb color to hexadecimal format.

### 25.6.2 modeci\_mdf.utils.create\_example\_node

`modeci_mdf.utils.create_example_node(node_id: str, graph: modeci_mdf.mdf.Graph) → modeci_mdf.mdf.Node`

Create a simple example node with Input inside a graph

#### Parameters

- **node\_id** – The unique id for the first node in the graph.
- **graph** – The graph to add the example node.

**Returns** The node (with id=node\_id) created in the graph.

### 25.6.3 modeci\_mdf.utils.is\_number

`modeci_mdf.utils.is_number(s)`  
Return True if cast to float does not throw ValueError, False otherwise.

### 25.6.4 modeci\_mdf.utils.load\_mdf

`modeci_mdf.utils.load_mdf(filename: str) → modeci_mdf.mdf.Model`  
Load an MDF file from JSON or YAML. File type is detected automatically based on extension.

### 25.6.5 modeci\_mdffutils.load\_mdf\_json

`modeci_mdffutils.load_mdf_json(filename: str) → modeci_mdffmdf.Model`  
Load an MDF JSON file

### 25.6.6 modeci\_mdffutils.load\_mdf\_yaml

`modeci_mdffutils.load_mdf_yaml(filename: str) → modeci_mdffmdf.Model`  
Load an MDF YAML file

### 25.6.7 modeci\_mdffutils.print\_summary

`modeci_mdffutils.print_summary(graph: modeci_mdffmdf.Graph)`  
Print a summary of a graph to standard out.

### 25.6.8 modeci\_mdffutils.simple\_connect

`modeci_mdffutils.simple_connect(pre_node, post_node, graph) → modeci_mdffmdf.Edge`  
Create an edge connecting two nodes in a graph.

#### Parameters

- **pre\_node** – The source node.
- **post\_node** – The destination node.
- **graph** – The graph to add the edge.

**Returns** The edge that has been added to the graph.

## INDICES AND TABLES

- `genindex`
- `modindex`
- `search`



## PYTHON MODULE INDEX

### m

- `modeci_mdf`, 115
- `modeci_mdf.execution_engine`, 115
- `modeci_mdf.full_translator`, 122
- `modeci_mdf.functions`, 122
  - `modeci_mdf.functions.actr`, 123
    - `modeci_mdf.functions.actr.ccm`, 126
      - `modeci_mdf.functions.actr.ccm.buffer`, 126
      - `modeci_mdf.functions.actr.ccm.dm`, 127
      - `modeci_mdf.functions.actr.ccm.logger`, 133
      - `modeci_mdf.functions.actr.ccm.model`, 135
      - `modeci_mdf.functions.actr.ccm.pattern`, 137
      - `modeci_mdf.functions.actr.ccm.scheduler`, 138
  - `modeci_mdf.functions.onnx`, 139
  - `modeci_mdf.functions.standard`, 197
- `modeci_mdf.interfaces`, 199
  - `modeci_mdf.interfaces.actr`, 199
    - `modeci_mdf.interfaces.actr.importer`, 199
  - `modeci_mdf.interfaces.graphviz`, 200
    - `modeci_mdf.interfaces.graphviz.exporter`, 200
  - `modeci_mdf.interfaces.onnx`, 202
    - `modeci_mdf.interfaces.onnx.exporter`, 203
    - `modeci_mdf.interfaces.onnx.importer`, 204
  - `modeci_mdf.interfaces.pytorch`, 206
    - `modeci_mdf.interfaces.pytorch.exporter`, 207
    - `modeci_mdf.interfaces.pytorch.importer`, 209
- `modeci_mdf.mdf`, 213
- `modeci_mdf.utils`, 223





## Symbols

`__call__()` (*in module `modeci_mdf.functions.actr.ccm.model.MethodGeneratorWrapper`*), 136  
`__call__()` (*in module `modeci_mdf.functions.actr.ccm.model.MethodWrapper`*), 136

## A

`abs()` (*in module `modeci_mdf.functions.onnx`*), 149  
`acos()` (*in module `modeci_mdf.functions.onnx`*), 149  
`acosh()` (*in module `modeci_mdf.functions.onnx`*), 149  
`actr_to_mdf()` (*in module `mod-eci_mdf.interfaces.actr.importer`*), 200  
`add()` (*in module `modeci_mdf.functions.onnx`*), 149  
`add_mdf_function()` (*in module `mod-eci_mdf.functions.standard`*), 198  
`and()` (*in module `modeci_mdf.functions.onnx`*), 149  
`argmax()` (*in module `modeci_mdf.functions.onnx`*), 149  
`argmin()` (*in module `modeci_mdf.functions.onnx`*), 150  
`asin()` (*in module `modeci_mdf.functions.onnx`*), 150  
`asinh()` (*in module `modeci_mdf.functions.onnx`*), 150  
`Associated` (*class in module `mod-eci_mdf.functions.actr.ccm.dm`*), 128  
`atan()` (*in module `modeci_mdf.functions.onnx`*), 150  
`atanh()` (*in module `modeci_mdf.functions.onnx`*), 150  
`averagepool()` (*in module `mod-eci_mdf.functions.onnx`*), 150

## B

`batchnormalization()` (*in module `mod-eci_mdf.functions.onnx`*), 151  
`bernoulli()` (*in module `modeci_mdf.functions.onnx`*), 151  
`bitshift()` (*in module `modeci_mdf.functions.onnx`*), 151  
`BlendingMemory` (*class in module `mod-eci_mdf.functions.actr.ccm.dm`*), 128  
`Buffer` (*class in module `modeci_mdf.functions.actr.ccm.buffer`*), 126  
`build_model()` (*in module `mod-eci_mdf.interfaces.actr.importer`*), 200  
`build_script()` (*in module `mod-eci_mdf.interfaces.pytorch.exporter`*), 207

## C

`castlike()` (*in module `modeci_mdf.functions.onnx`*), 152  
`ceil()` (*in module `modeci_mdf.functions.onnx`*), 153  
`celu()` (*in module `modeci_mdf.functions.onnx`*), 153  
`change_goal()` (*in module `mod-eci_mdf.functions.actr`*), 123  
`check_termination()` (*in module `mod-eci_mdf.functions.actr`*), 123  
`Chunk` (*class in module `modeci_mdf.functions.actr.ccm.buffer`*), 127  
`chunk_to_string()` (*in module `mod-eci_mdf.functions.actr`*), 124  
`clip()` (*in module `modeci_mdf.functions.onnx`*), 153  
`color_rgb_to_hex()` (*in module `modeci_mdf.utils`*), 223  
`compress()` (*in module `modeci_mdf.functions.onnx`*), 153  
`concat()` (*in module `modeci_mdf.functions.onnx`*), 153  
`concatfromsequence()` (*in module `mod-eci_mdf.functions.onnx`*), 153  
`Condition` (*class in module `modeci_mdf.mdf`*), 215  
`ConditionSet` (*class in module `modeci_mdf.mdf`*), 215  
`conflict_resolution_function()` (*in module `modeci_mdf.functions.actr`*), 124  
`constant()` (*in module `modeci_mdf.functions.onnx`*), 153  
`constantofshape()` (*in module `mod-eci_mdf.functions.onnx`*), 154  
`conv()` (*in module `modeci_mdf.functions.onnx`*), 154  
`convert_file()` (*in module `mod-eci_mdf.interfaces.onnx.importer`*), 204  
`convert_mdf_file_to_onnx()` (*in module `mod-eci_mdf.interfaces.onnx.exporter`*), 203  
`convert_states_to_stateful_parameters()` (*in module `modeci_mdf.full_translator`*), 122  
`convert_to_serializable()` (*in module `mod-eci_mdf.interfaces.pytorch.importer`*), 210  
`convert_type()` (*in module `mod-eci_mdf.functions.onnx`*), 154  
`convinteger()` (*in module `mod-`*

`eci_mdf.functions.onnx`), 154  
`convtranspose()` (in module `mod-eci_mdf.functions.onnx`), 154  
`cos()` (in module `modeci_mdf.functions.onnx`), 154  
`cosh()` (in module `modeci_mdf.functions.onnx`), 155  
`create_example_node()` (in module `mod-eci_mdf.utils`), 223  
`create_python_expression()` (in module `mod-eci_mdf.functions.standard`), 198  
`create_python_function()` (in module `mod-eci_mdf.functions.standard`), 198  
`cumsum()` (in module `modeci_mdf.functions.onnx`), 155

## D

`dependency_dict()` (`modeci_mdf.mdf.Graph` property), 217  
`depthtospace()` (in module `mod-eci_mdf.functions.onnx`), 155  
`dequantizelinear()` (in module `mod-eci_mdf.functions.onnx`), 155  
`det()` (in module `modeci_mdf.functions.onnx`), 156  
`div()` (in module `modeci_mdf.functions.onnx`), 156  
`DMAssociate` (class in `mod-eci_mdf.functions.actr.ccm.dm`), 128  
`DMBaseLevel` (class in `mod-eci_mdf.functions.actr.ccm.dm`), 129  
`DMFixed` (class in `modeci_mdf.functions.actr.ccm.dm`), 129  
`DMInhibition` (class in `mod-eci_mdf.functions.actr.ccm.dm`), 129  
`DMNoise` (class in `modeci_mdf.functions.actr.ccm.dm`), 130  
`DMSalience` (class in `mod-eci_mdf.functions.actr.ccm.dm`), 130  
`DMSpacing` (class in `mod-eci_mdf.functions.actr.ccm.dm`), 130  
`DMSpreading` (class in `mod-eci_mdf.functions.actr.ccm.dm`), 131  
`dropout()` (in module `modeci_mdf.functions.onnx`), 156  
`DummyLog` (class in `mod-eci_mdf.functions.actr.ccm.logger`), 133  
`dynamicquantizelinear()` (in module `mod-eci_mdf.functions.onnx`), 156

## E

`Edge` (class in `modeci_mdf.mdf`), 215  
`einsum()` (in module `modeci_mdf.functions.onnx`), 157  
`elu()` (in module `modeci_mdf.functions.onnx`), 157  
`equal()` (in module `modeci_mdf.functions.onnx`), 157  
`erf()` (in module `modeci_mdf.functions.onnx`), 157  
`EvaluableFunction` (class in `mod-eci_mdf.execution_engine`), 117

`EvaluableGraph` (class in `mod-eci_mdf.execution_engine`), 118  
`EvaluableInput` (class in `mod-eci_mdf.execution_engine`), 119  
`EvaluableNode` (class in `mod-eci_mdf.execution_engine`), 120  
`EvaluableOutput` (class in `mod-eci_mdf.execution_engine`), 120  
`EvaluableParameter` (class in `mod-eci_mdf.execution_engine`), 121  
`evaluate()` (`modeci_mdf.execution_engine.EvaluableFunction` method), 118  
`evaluate()` (`modeci_mdf.execution_engine.EvaluableGraph` method), 118  
`evaluate()` (`modeci_mdf.execution_engine.EvaluableInput` method), 119  
`evaluate()` (`modeci_mdf.execution_engine.EvaluableNode` method), 120  
`evaluate()` (`modeci_mdf.execution_engine.EvaluableOutput` method), 121  
`evaluate()` (`modeci_mdf.execution_engine.EvaluableParameter` method), 121  
`evaluate_edge()` (`mod-eci_mdf.execution_engine.EvaluableGraph` method), 118  
`evaluate_expr()` (in module `mod-eci_mdf.execution_engine`), 116  
`evaluate_onnx_expr()` (in module `mod-eci_mdf.execution_engine`), 116  
`Event` (class in `mod-eci_mdf.functions.actr.ccm.scheduler`), 138  
`exp()` (in module `modeci_mdf.functions.onnx`), 158  
`expand()` (in module `modeci_mdf.functions.onnx`), 158  
`eyelike()` (in module `modeci_mdf.functions.onnx`), 158

## F

`file_exists()` (in module `mod-eci_mdf.functions.actr.ccm.logger`), 133  
`find_subgraphs()` (in module `mod-eci_mdf.interfaces.onnx.importer`), 204  
`finished()` (in module `mod-eci_mdf.functions.actr.ccm.logger`), 133  
`First` (class in `modeci_mdf.functions.actr.ccm.dm`), 131  
`flatten()` (in module `modeci_mdf.functions.onnx`), 158  
`floor()` (in module `modeci_mdf.functions.onnx`), 158  
`format_bold()` (in module `mod-eci_mdf.interfaces.graphviz.exporter`), 201  
`format_condition()` (in module `mod-eci_mdf.interfaces.graphviz.exporter`), 201  
`format_function()` (in module `mod-eci_mdf.interfaces.graphviz.exporter`), 201

`format_input()` (in module `modeci_mdf.interfaces.graphviz.exporter`), 201  
`format_label()` (in module `modeci_mdf.interfaces.graphviz.exporter`), 201  
`format_num()` (in module `modeci_mdf.interfaces.graphviz.exporter`), 201  
`format_output()` (in module `modeci_mdf.interfaces.graphviz.exporter`), 201  
`format_param()` (in module `modeci_mdf.interfaces.graphviz.exporter`), 202  
`format_standard_func()` (in module `modeci_mdf.interfaces.graphviz.exporter`), 202  
`format_standard_func_long()` (in module `modeci_mdf.interfaces.graphviz.exporter`), 202  
`format_term_condition()` (in module `modeci_mdf.interfaces.graphviz.exporter`), 202  
`func_args()` (in module `modeci_mdf.interfaces.pytorch.exporter`), 208  
`Function` (class in `modeci_mdf.mdf`), 216

## G

`gather()` (in module `modeci_mdf.functions.onnx`), 158  
`gatherelements()` (in module `modeci_mdf.functions.onnx`), 159  
`gathernd()` (in module `modeci_mdf.functions.onnx`), 160  
`gemm()` (in module `modeci_mdf.functions.onnx`), 161  
`generate_main_forward()` (in module `modeci_mdf.interfaces.pytorch.exporter`), 208  
`generate_onnx_graph()` (in module `modeci_mdf.interfaces.onnx.exporter`), 203  
`generate_onnx_node()` (in module `modeci_mdf.interfaces.onnx.exporter`), 203  
`get()` (in module `modeci_mdf.functions.actr.ccm.pattern`), 137  
`get_actr_functions()` (in module `modeci_mdf.functions.actr`), 124  
`get_all_schemas_version()` (in module `modeci_mdf.functions.onnx`), 162  
`get_category_of_onnx_node()` (in module `modeci_mdf.interfaces.onnx.importer`), 205  
`get_color_for_onnx_category()` (in module `modeci_mdf.interfaces.onnx.importer`), 205  
`get_current_value()` (`modeci_mdf.execution_engine.EvaluableParameter` method), 121  
`get_graph_constants()` (in module `modeci_mdf.interfaces.pytorch.importer`), 210  
`get_input_port()` (`modeci_mdf.mdf.Node` method), 220  
`get_module_declaration_text()` (in module `modeci_mdf.interfaces.pytorch.exporter`), 208  
`get_node()` (`modeci_mdf.mdf.Graph` method), 217  
`get_onnx_attribute()` (in module `modeci_mdf.interfaces.onnx.importer`), 205  
`get_onnx_ops()` (in module `modeci_mdf.functions.onnx`), 162  
`get_onnx_schema()` (in module `modeci_mdf.functions.onnx`), 162  
`get_output()` (`modeci_mdf.execution_engine.EvaluableNode` method), 120  
`get_output_port()` (`modeci_mdf.mdf.Node` method), 220  
`get_parameter()` (`modeci_mdf.mdf.Node` method), 220  
`get_required_variables_from_expression()` (in module `modeci_mdf.execution_engine`), 117  
`get_shape_params()` (in module `modeci_mdf.interfaces.onnx.importer`), 205  
`globalaveragepool()` (in module `modeci_mdf.functions.onnx`), 162  
`globallppool()` (in module `modeci_mdf.functions.onnx`), 162  
`globalmaxpool()` (in module `modeci_mdf.functions.onnx`), 163  
`Graph` (class in `modeci_mdf.mdf`), 217  
`greater()` (in module `modeci_mdf.functions.onnx`), 163  
`greaterorequal()` (in module `modeci_mdf.functions.onnx`), 163  
`gru()` (in module `modeci_mdf.functions.onnx`), 163

## H

`hardmax()` (in module `modeci_mdf.functions.onnx`), 164  
`hardsigmoid()` (in module `modeci_mdf.functions.onnx`), 165  
`hardswish()` (in module `modeci_mdf.functions.onnx`), 165

## I

`id_to_port()` (in module `modeci_mdf.interfaces.onnx.importer`), 205  
`id_to_port()` (`modeci_mdf.interfaces.pytorch.importer.PortMapper` method), 213  
`identity()` (in module `modeci_mdf.functions.onnx`), 165  
`if()` (in module `modeci_mdf.functions.onnx`), 165  
`import_class()` (in module `modeci_mdf.functions.onnx`), 165  
`InputPort` (class in `modeci_mdf.mdf`), 218  
`inputs()` (`modeci_mdf.mdf.Graph` property), 217  
`instancenormalization()` (in module `modeci_mdf.functions.onnx`), 165  
`is_number()` (in module `modeci_mdf.utils`), 223

`is_stateful()` (*in module `modeci_mdf.mdf.Parameter` method*), 222  
`isinf()` (*in module `modeci_mdf.functions.onnx`*), 165  
`isnan()` (*in module `modeci_mdf.functions.onnx`*), 166

## L

`leakyrelu()` (*in module `modeci_mdf.functions.onnx`*), 166  
`less()` (*in module `modeci_mdf.functions.onnx`*), 166  
`lessorequal()` (*in module `modeci_mdf.functions.onnx`*), 166  
`load_mdf()` (*in module `modeci_mdf.utils`*), 223  
`load_mdf_json()` (*in module `modeci_mdf.utils`*), 224  
`load_mdf_yaml()` (*in module `modeci_mdf.utils`*), 224  
`Log` (*class in module `modeci_mdf.functions.actr.ccm.logger`*), 134  
`log()` (*in module `modeci_mdf.functions.actr.ccm.logger`*), 133  
`log()` (*in module `modeci_mdf.functions.onnx`*), 166  
`log_everything()` (*in module `modeci_mdf.functions.actr.ccm.model`*), 135  
`LogProxy` (*class in module `modeci_mdf.functions.actr.ccm.logger`*), 134  
`logsoftmax()` (*in module `modeci_mdf.functions.onnx`*), 166  
`loop()` (*in module `modeci_mdf.functions.onnx`*), 167  
`lpnormalization()` (*in module `modeci_mdf.functions.onnx`*), 169  
`lppool()` (*in module `modeci_mdf.functions.onnx`*), 169  
`lrn()` (*in module `modeci_mdf.functions.onnx`*), 169  
`lstm()` (*in module `modeci_mdf.functions.onnx`*), 169

## M

`main()` (*in module `modeci_mdf.execution_engine`*), 117  
`main()` (*in module `modeci_mdf.interfaces.onnx.exporter`*), 203  
`main()` (*in module `modeci_mdf.interfaces.onnx.importer`*), 205  
`make_func_id()` (*in module `modeci_mdf.interfaces.pytorch.importer`*), 210  
`make_model_graph_name()` (*in module `modeci_mdf.interfaces.pytorch.importer`*), 210  
`make_node_id()` (*in module `modeci_mdf.interfaces.pytorch.importer`*), 210  
`match_in_expr()` (*in module `modeci_mdf.interfaces.graphviz.exporter`*), 202  
`match_production()` (*in module `modeci_mdf.functions.actr`*), 124  
`matmul()` (*in module `modeci_mdf.functions.onnx`*), 171  
`matmulinteger()` (*in module `modeci_mdf.functions.onnx`*), 171  
`max()` (*in module `modeci_mdf.functions.onnx`*), 171  
`maxpool()` (*in module `modeci_mdf.functions.onnx`*), 171  
`maxroi_pool()` (*in module `modeci_mdf.functions.onnx`*), 172  
`maxunpool()` (*in module `modeci_mdf.functions.onnx`*), 172  
`mdf_to_graphviz()` (*in module `modeci_mdf.interfaces.graphviz.exporter`*), 202  
`mdf_to_onnx()` (*in module `modeci_mdf.interfaces.onnx.exporter`*), 204  
`mdf_to_pytorch()` (*in module `modeci_mdf.interfaces.pytorch.exporter`*), 209  
`mean()` (*in module `modeci_mdf.functions.onnx`*), 172  
`meanvariancennormalization()` (*in module `modeci_mdf.functions.onnx`*), 172  
`Memory` (*class in module `modeci_mdf.functions.actr.ccm.dm`*), 131  
`MemorySubModule` (*class in module `modeci_mdf.functions.actr.ccm.dm`*), 132  
`MethodGeneratorWrapper` (*class in module `modeci_mdf.functions.actr.ccm.model`*), 136  
`MethodWrapper` (*class in module `modeci_mdf.functions.actr.ccm.model`*), 136  
`min()` (*in module `modeci_mdf.functions.onnx`*), 172  
`mod()` (*in module `modeci_mdf.functions.onnx`*), 173  
`modeci_mdf`  
   module, 115  
`modeci_mdf.execution_engine`  
   module, 115  
`modeci_mdf.full_translator`  
   module, 122  
`modeci_mdf.functions`  
   module, 122  
`modeci_mdf.functions.actr`  
   module, 123  
`modeci_mdf.functions.actr.ccm`  
   module, 126  
`modeci_mdf.functions.actr.ccm.buffer`  
   module, 126  
`modeci_mdf.functions.actr.ccm.dm`  
   module, 127  
`modeci_mdf.functions.actr.ccm.logger`  
   module, 133  
`modeci_mdf.functions.actr.ccm.model`  
   module, 135  
`modeci_mdf.functions.actr.ccm.pattern`  
   module, 137  
`modeci_mdf.functions.actr.ccm.scheduler`  
   module, 138  
`modeci_mdf.functions.onnx`  
   module, 139  
`modeci_mdf.functions.standard`  
   module, 197  
`modeci_mdf.interfaces`  
   module, 199  
`modeci_mdf.interfaces.actr`

module, 199  
 modeci\_mdf.interfaces.actr.importer  
     module, 199  
 modeci\_mdf.interfaces.graphviz  
     module, 200  
 modeci\_mdf.interfaces.graphviz.exporter  
     module, 200  
 modeci\_mdf.interfaces.onnx  
     module, 202  
 modeci\_mdf.interfaces.onnx.exporter  
     module, 203  
 modeci\_mdf.interfaces.onnx.importer  
     module, 204  
 modeci\_mdf.interfaces.pytorch  
     module, 206  
 modeci\_mdf.interfaces.pytorch.exporter  
     module, 207  
 modeci\_mdf.interfaces.pytorch.importer  
     module, 209  
 modeci\_mdf.mdf  
     module, 213  
 modeci\_mdf.utils  
     module, 223  
 Model (class in modeci\_mdf.functions.actr.ccm.model), 136  
 Model (class in modeci\_mdf.mdf), 218  
 module  
     modeci\_mdf, 115  
     modeci\_mdf.execution\_engine, 115  
     modeci\_mdf.full\_translator, 122  
     modeci\_mdf.functions, 122  
     modeci\_mdf.functions.actr, 123  
     modeci\_mdf.functions.actr.ccm, 126  
     modeci\_mdf.functions.actr.ccm.buffer, 126  
     modeci\_mdf.functions.actr.ccm.dm, 127  
     modeci\_mdf.functions.actr.ccm.logger, 133  
     modeci\_mdf.functions.actr.ccm.model, 135  
     modeci\_mdf.functions.actr.ccm.pattern, 137  
     modeci\_mdf.functions.actr.ccm.scheduler, 138  
     modeci\_mdf.functions.onnx, 139  
     modeci\_mdf.functions.standard, 197  
     modeci\_mdf.interfaces, 199  
     modeci\_mdf.interfaces.actr, 199  
     modeci\_mdf.interfaces.actr.importer, 199  
     modeci\_mdf.interfaces.graphviz, 200  
     modeci\_mdf.interfaces.graphviz.exporter, 200  
     modeci\_mdf.interfaces.onnx, 202  
     modeci\_mdf.interfaces.onnx.exporter, 203  
     modeci\_mdf.interfaces.onnx.importer, 204  
     modeci\_mdf.interfaces.pytorch, 206  
     modeci\_mdf.interfaces.pytorch.exporter, 207  
     modeci\_mdf.interfaces.pytorch.importer, 209  
     modeci\_mdf.mdf, 213  
     modeci\_mdf.utils, 223  
 mul() (in module modeci\_mdf.functions.onnx), 173  
 multinomial() (in module modeci\_mdf.functions.onnx), 173

## N

neg() (in module modeci\_mdf.functions.onnx), 173  
 negativeloglikelihoodloss() (in module modeci\_mdf.functions.onnx), 173  
 Node (class in modeci\_mdf.mdf), 219  
 nonmaxsuppression() (in module modeci\_mdf.functions.onnx), 175  
 nonzero() (in module modeci\_mdf.functions.onnx), 175  
 not() (in module modeci\_mdf.functions.onnx), 175

## O

onehot() (in module modeci\_mdf.functions.onnx), 175  
 onnx\_node\_to\_mdf() (in module modeci\_mdf.interfaces.onnx.importer), 206  
 onnx\_to\_mdf() (in module modeci\_mdf.interfaces.onnx.importer), 206  
 optional() (in module modeci\_mdf.functions.onnx), 175  
 optionalgetelement() (in module modeci\_mdf.functions.onnx), 176  
 optionalhaselement() (in module modeci\_mdf.functions.onnx), 176  
 or() (in module modeci\_mdf.functions.onnx), 176  
 OutputPort (class in modeci\_mdf.mdf), 220

## P

pad() (in module modeci\_mdf.functions.onnx), 176  
 Parameter (class in modeci\_mdf.mdf), 221  
 ParameterCondition (class in modeci\_mdf.mdf), 222  
 parse() (in module modeci\_mdf.functions.actr.ccm.pattern), 137  
 parse\_condition() (in module modeci\_mdf.execution\_engine.EvaluableGraph method), 119  
 parsed\_structure\_factory() (in module modeci\_mdf.mdf), 214



`parsed_unstructure_factory()` (in module `modeci_mdf.mdf`), 214  
`Partial` (class in `modeci_mdf.functions.actr.ccm.dm`), 132  
`partialmatch()` (in module `modeci_mdf.functions.actr.ccm.pattern`), 137  
`Pattern` (class in `modeci_mdf.functions.actr.ccm.pattern`), 137  
`pattern_matching_function()` (in module `modeci_mdf.functions.actr`), 124  
`pattern_to_string()` (in module `modeci_mdf.functions.actr`), 125  
`PatternException`, 138  
`port_to_id()` (in module `modeci_mdf.interfaces.pytorch.importer.PortMapper` method), 213  
`PortMapper` (class in `modeci_mdf.interfaces.pytorch.importer`), 213  
`pow()` (in module `modeci_mdf.functions.onnx`), 177  
`predict_with_onnxruntime()` (in module `modeci_mdf.functions.onnx`), 177  
`prelu()` (in module `modeci_mdf.functions.onnx`), 177  
`print_summary()` (in module `modeci_mdf.utils`), 224  
`process_onnx_schema()` (in module `modeci_mdf.interfaces.pytorch.importer`), 210  
`process_torch_schema()` (in module `modeci_mdf.interfaces.pytorch.importer`), 211  
`pytorch_to_mdf()` (in module `modeci_mdf.interfaces.pytorch.importer`), 211  
`reducelogsum()` (in module `modeci_mdf.functions.onnx`), 180  
`reducelogsumexp()` (in module `modeci_mdf.functions.onnx`), 180  
`reducemax()` (in module `modeci_mdf.functions.onnx`), 181  
`reducemean()` (in module `modeci_mdf.functions.onnx`), 181  
`reducemin()` (in module `modeci_mdf.functions.onnx`), 181  
`reduceprod()` (in module `modeci_mdf.functions.onnx`), 181  
`reducesum()` (in module `modeci_mdf.functions.onnx`), 181  
`reducesumsquare()` (in module `modeci_mdf.functions.onnx`), 182  
`relu()` (in module `modeci_mdf.functions.onnx`), 182  
`reshape()` (in module `modeci_mdf.functions.onnx`), 182  
`resize()` (in module `modeci_mdf.functions.onnx`), 182  
`retrieve_chunk()` (in module `modeci_mdf.functions.actr`), 125  
`reversesequence()` (in module `modeci_mdf.functions.onnx`), 182  
`rnn()` (in module `modeci_mdf.functions.onnx`), 183  
`roialign()` (in module `modeci_mdf.functions.onnx`), 184  
`round()` (in module `modeci_mdf.functions.onnx`), 184  
`run_onnx_op()` (in module `modeci_mdf.functions.onnx`), 184

## Q

`qlinearconv()` (in module `modeci_mdf.functions.onnx`), 178  
`qlinearmatmul()` (in module `modeci_mdf.functions.onnx`), 178  
`quantizelinear()` (in module `modeci_mdf.functions.onnx`), 178

## R

`randomnormal()` (in module `modeci_mdf.functions.onnx`), 178  
`randomnormallike()` (in module `modeci_mdf.functions.onnx`), 179  
`randomuniform()` (in module `modeci_mdf.functions.onnx`), 179  
`randomuniformlike()` (in module `modeci_mdf.functions.onnx`), 179  
`range()` (in module `modeci_mdf.functions.onnx`), 179  
`reciprocal()` (in module `modeci_mdf.functions.onnx`), 180  
`reducel1()` (in module `modeci_mdf.functions.onnx`), 180  
`reducel2()` (in module `modeci_mdf.functions.onnx`), 180

## S

`safe_comparator()` (in module `modeci_mdf.interfaces.graphviz.exporter`), 202  
`scan()` (in module `modeci_mdf.functions.onnx`), 185  
`scatter()` (in module `modeci_mdf.functions.onnx`), 186  
`scatterelements()` (in module `modeci_mdf.functions.onnx`), 187  
`scatternd()` (in module `modeci_mdf.functions.onnx`), 188  
`Scheduler` (class in `modeci_mdf.functions.actr.ccm.scheduler`), 138  
`SchedulerError`, 139  
`selu()` (in module `modeci_mdf.functions.onnx`), 189  
`sequencecat()` (in module `modeci_mdf.functions.onnx`), 189  
`sequenceconstruct()` (in module `modeci_mdf.functions.onnx`), 189  
`sequenceempty()` (in module `modeci_mdf.functions.onnx`), 189  
`sequenceerase()` (in module `modeci_mdf.functions.onnx`), 189

`sequenceinsert()` (in module `mod-eci_mdf.functions.onnx`), 189  
`sequencelength()` (in module `mod-eci_mdf.functions.onnx`), 189  
`set_input_value()` (`mod-eci_mdf.execution_engine.EvaluableInput` method), 119  
`shape()` (in module `modeci_mdf.functions.onnx`), 190  
`shrink()` (in module `modeci_mdf.functions.onnx`), 190  
`sigmoid()` (in module `modeci_mdf.functions.onnx`), 190  
`sign()` (in module `modeci_mdf.functions.onnx`), 190  
`simple_connect()` (in module `modeci_mdf.utils`), 224  
`sin()` (in module `modeci_mdf.functions.onnx`), 190  
`sinh()` (in module `modeci_mdf.functions.onnx`), 191  
`size()` (in module `modeci_mdf.functions.onnx`), 191  
`slice()` (in module `modeci_mdf.functions.onnx`), 191  
`softmax()` (in module `modeci_mdf.functions.onnx`), 192  
`softmaxcrossentropyloss()` (in module `mod-eci_mdf.functions.onnx`), 192  
`softplus()` (in module `modeci_mdf.functions.onnx`), 192  
`softsign()` (in module `modeci_mdf.functions.onnx`), 192  
`spacetodepth()` (in module `mod-eci_mdf.functions.onnx`), 193  
`split()` (in module `modeci_mdf.functions.onnx`), 193  
`splittosequence()` (in module `mod-eci_mdf.functions.onnx`), 193  
`sqrt()` (in module `modeci_mdf.functions.onnx`), 193  
`squeeze()` (in module `modeci_mdf.functions.onnx`), 193  
`stringnormalizer()` (in module `mod-eci_mdf.functions.onnx`), 193  
`sub()` (in module `modeci_mdf.functions.onnx`), 194  
`substitute_args()` (in module `mod-eci_mdf.functions.standard`), 199  
`sum()` (in module `modeci_mdf.functions.onnx`), 194  
`summary()` (`modeci_mdf.mdf.Parameter` method), 222  
`sym()` (in module `mod-eci_mdf.interfaces.pytorch.exporter`), 209

`topk()` (in module `modeci_mdf.functions.onnx`), 195  
`torchnode_to_mdnode()` (in module `mod-eci_mdf.interfaces.pytorch.importer`), 212  
`Trace` (class in `modeci_mdf.functions.actr.ccm.logger`), 135  
`translate_graph()` (in module `mod-eci_mdf.interfaces.pytorch.importer`), 212  
`transpose()` (in module `modeci_mdf.functions.onnx`), 195  
`Trigger` (class in `mod-eci_mdf.functions.actr.ccm.scheduler`), 139  
`trilu()` (in module `modeci_mdf.functions.onnx`), 195

## U

`unique()` (in module `modeci_mdf.functions.onnx`), 196  
`unsqueeze()` (in module `modeci_mdf.functions.onnx`), 197  
`update_buffer()` (in module `mod-eci_mdf.functions.actr`), 125  
`update_goal()` (in module `mod-eci_mdf.functions.actr`), 125  
`update_retrieval()` (in module `mod-eci_mdf.functions.actr`), 125  
`upsample()` (in module `modeci_mdf.functions.onnx`), 197

## V

`v()` (in module `modeci_mdf.mdf`), 214

## W

`where()` (in module `modeci_mdf.functions.onnx`), 197

## X

`xor()` (in module `modeci_mdf.functions.onnx`), 197

## T

`tan()` (in module `modeci_mdf.functions.onnx`), 194  
`tanh()` (in module `modeci_mdf.functions.onnx`), 194  
`tfidfvectorizer()` (in module `mod-eci_mdf.functions.onnx`), 194  
`thresholdedrelu()` (in module `mod-eci_mdf.functions.onnx`), 195  
`tile()` (in module `modeci_mdf.functions.onnx`), 195  
`to_graph_image()` (`modeci_mdf.mdf.Model` method), 219