cONNXr

Release 0.0.0

Mar 11, 2021

Documentation

Note: This is project is in a very early stage and we are looking for contributors.

A onnx runtime written in pure C99 with zero dependencies focused on embedded devices. Run inference on your machine learning models no matter which framework you train it with and no matter the device that you use. This is the perfect way to go in old hardware that doesn't support fancy modern C or C++.

CHAPTER 1

Out of the box example

Estimate the number written in an image using the MNIST model. The input image is stored in *input_0.pb* and the MNIST model is stored in onnx format in *model.onnx*.

First compile the project.

```
make all
```

And run inference on the input using the model.

```
build/connxr test/mnist/model.onnx test/mnist/test_data_set_0/input_0.pb
```

1.1 Documentation

1.1.1 Introduction

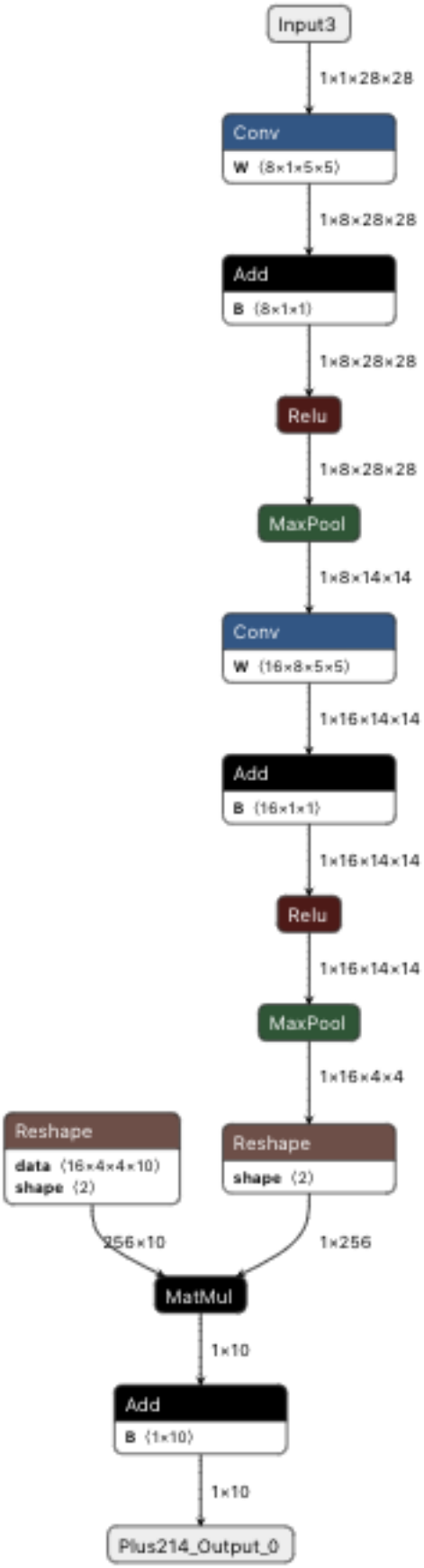
What is ONNX?

If you don't know about `onnx` you might want to read about it before since its the building block of this project. They have a nice website and great repositories with a lot of documentation to read about. Everything is open source, and really big companies in the industry are behind it (AMD, ARM, AWS, Nvidia, IBM) just to name a few. Here you can find a mix of official and non official related repositories:

- https://github.com/onnx/onnx
- https://github.com/onnx/onnx-r
- https://github.com/onnx/models
- https://github.com/owulveryck/onnx-go
- https://github.com/microsoft/onnxruntime

In short, `onnx` provides a **O**pen **N**eural **N**etwork **E**xchange format. This format, describes a huge set of operators, that can be mixed to create every type of machine learning model that you ever heard of, from a simple neural network to complex deep convolutional networks. Some examples of operators are: matrix multiplications, convolutions, adding, maxpool, sin, cosine. They provide a standardised set of operators here. So we can say that `onnx` provides a layer of abstraction to ML models, which makes all frameworks compatible between them. Exporters are provided for a huge variety of frameworks (PyTorch, TensorFlow, Keras, Scikit-Learn) so if you want to convert a model from Keras to TensorFlow, you just have to use Keras exporter to export `Keras->ONNX` and then use the importer to import `ONNX-TensorFlow`.

In the following image, you can find an example on how a `onnx` model looks like. Its just a bunch of `nodes` that are connected between them to form a `graph`. Each node has an `operator` that takes some `inputs` with some specific `dimensions` and some `attributes` and calculates some `outputs`. This is how the inference is calculated, just forward propagating the input along every node until the last one is reached.


Input3
1×1×28×28
Conv
W (8×1×5×5)
1×8×28×28
Add
B (8×1×1)
1×8×28×28
Relu
1×8×28×28
MaxPool
1×8×14×14
Conv
W (16×8×5×5)
1×16×14×14
Add
B (16×1×1)
1×16×14×14
Relu
1×16×14×14
MaxPool
1×16×4×4
Reshape
data (16×4×4×10)
shape (2)
Reshape
shape (2)
256×10
1×256
MatMul
1×10
Add
B (1×10)
1×10
Plus214_Output_0

mnist

So this `.onnx` format can describe machine learning models through a set of nodes that contain specific operators connected among them. On top of that, they also offer a set of the so called "runtimes" see link. A runtime allows to run inference on a model, and they offer different ones in a wide variety of languages and hardware. Unfortunately, all the runtimes rely on modern versions of C/C++ with many abstractions and dependancies, which might be a no go for some specific projects. Here is where we come in.

What is cONNXr

Well, now that you know about `onnx`, our project is just a runtime that runs inference on `onnx` models. The `c` means that is implemented in C language and the `r` means that its a runtime for `ONNX`. The only difference between this runtime and the others, is that this one is written in pure `C99` without any dependancy. This means that it should be able to compile with almost any compiler, no matter how old it is. Our goal is to enable embedded devices that doesn't have much resources or fancy features (like GPUs or any type of hardware accelerator) to run inference. No GPUs, no multithreading, no dependancies, just pure C code with the lowest possible footprint. Train your model in whatever ML framework you want, export it to `.onnx` and deploy it wherever you want.

You might also find this project useful if you work with some bare metal hardware with dedicated accelerators. If this is the case, you might find useful to reuse the architecture and replace the specific operators (functions) by your own ones.

Or perhaps you have a different use case. If this is the case, we would love to hear about it.

1.1.2 File structure

The most relevant files and folders are the following ones:

- `include/src`: Source and header C files.
- `include/operators/onnx/`: Autogenerated header files for all operators and versions. There is one header file per operator, and within each file there is one function declaration for each data type. For example, for the operator `conv` version 11 there is one header file, that contains 3 function declarations (double, float and float16). Note that these functions are defined as extern and not implemented.
- `src/operators/implementation`: Here is where all the operators implementations are located. The functions that are defined here must use the prototype defined in `include/operators/onnx` mentioned above. See some examples.
- `src/operators/resolve`: This is autogenerated code, that resolves the mapping between an operator + data type and the function that should be executed.
- `src/operators/operator_set.c`: Autogenerated file that stores the relationship between an operator name (i.e. `Conv`) and the function that resolves its data type. Operator version is also taken into account.
- `scripts/onnx_generator`: Set of Python scripts that generates the header and C files existing in `include/operators/onnx`, `src/operators/resolve` and `src/operators/operator_set.c`.
- `src/test` and `include/test`: Test code divided into two levels: operator level and model level.
- `test`: Test vectors on operator and model level. Bunch of `.onnx` models and `.pb` files (input + expected output)

1.1.3 Operators interface

All ONNX operators comply with the following interface. This struct contains all the data that an operator needs to run, like its inputs and attributes. In `operator_executer` it contains the pointer to the function that runs that

specific node. For example, if node `i` contains a `Conv`, that `operator_executer` will point to the `conv` function defined in `src/operators/implementation`.

You can check more information about `Onnx__NodeProto` and `Onnx__TensorProto` structs in `onnx.pb-c.h`, which is a file that is generated from the ONNX common interface.

```
struct node_context{
  Onnx__NodeProto     *onnx_node;
  Onnx__TensorProto  **inputs;
  Onnx__TensorProto  **outputs;
  operator_executer resolved_op;
};
```

1.1.4 Types and Structures

We can divide the types and structures that this repo uses into two:

- ONNX structures: These structures are exactly (or almost) the same as the ones that ONNX defines in its onnx.proto. See `onnx.pb-c.h`. We list the most relevant ones below.
- Custom structures: These structures are defined *ad hoc* for this project.

ONNX structures

Onnx__GraphProto

Defines the graph of a model, made of different nodes `Onnx__NodeProto`. See `onnx.pb-c.h`.

```
struct  _Onnx__GraphProto
{
  ProtobufCMessage base;
  size_t n_node;
  Onnx__NodeProto **node;
  char *name;
  size_t n_initializer;
  Onnx__TensorProto **initializer;
  size_t n_sparse_initializer;
  Onnx__SparseTensorProto **sparse_initializer;
  char *doc_string;
  size_t n_input;
  Onnx__ValueInfoProto **input;
  size_t n_output;
  Onnx__ValueInfoProto **output;
  size_t n_value_info;
  Onnx__ValueInfoProto **value_info;
  size_t n_quantization_annotation;
  Onnx__TensorAnnotation **quantization_annotation;
};
```

Onnx__NodeProto

Information for a given node (a node has inputs and attributes and provides an output using a given operator). See `onnx.pb-c.h`.

```
struct  _Onnx__NodeProto
{
  ProtobufCMessage base;
  size_t n_input;
  char **input;
  size_t n_output;
  char **output;
  char *name;
  char *op_type;
  char *domain;
  size_t n_attribute;
  Onnx__AttributeProto **attribute;
  char *doc_string;
};
```

Onnx__TensorProto

One of the most important types that you will see, is the `Onnx__TensorProto`. It just defines a vector, an array, a matrix, or whatever you want to call it. It is quite convinient to use, because it is quite generic. You can store different types of values, with different sizes. As an example, lets say that we want to store a 3 dimension vector. In that case `n_dims=3` and `dims[0]`, `dims[1]`, `dims[2]` will store some values. Lets store some `float` values, so `data_type=ONNX__TENSOR_PROTO__DATA_TYPE__FLOAT`. In this case `n_float_data=dims[0]*dims[1]*dims[2]` and `float_data` will contain all the values in a single dimension array. The tensor has also a `name`.

```
struct  _Onnx__TensorProto
{
  ProtobufCMessage base;
  size_t n_dims;
  int64_t *dims;

  protobuf_c_boolean has_data_type;
  int32_t data_type;
  Onnx__TensorProto__Segment *segment;

  size_t n_float_data;
  float *float_data;

  size_t n_int32_data;
  int32_t *int32_data;

  size_t n_string_data;
  ProtobufCBinaryData *string_data;

  size_t n_int64_data;
  int64_t *int64_data;

  char *name;
  char *doc_string;

  protobuf_c_boolean has_raw_data;
  ProtobufCBinaryData raw_data;

  size_t n_external_data;
  Onnx__StringStringEntryProto **external_data;
```

(continues on next page)

(continued from previous page)

```
  protobuf_c_boolean has_data_location;
  Onnx__TensorProto__DataLocation data_location;

  size_t n_double_data;
  double *double_data;

  size_t n_uint64_data;
  uint64_t *uint64_data;
};
```

Custom structures

Since we are at a very early stage, structures are constantly evolving. Have a look to the code in order to know more. At some point we will also document them here.

1.1.5 Protocol Buffers

In order to convert from the ONNX interface defined in `onnx.proto`, the `protoc` library is used. `onnx` uses protocol buffers to serialize the models data. Note that `protobuf-c` is used to generate the `pb/onnx.pb-c.c` and `pb/onnx.pb-c.h`. Files are already provided, but you can generate it like this:

```
protoc --c_out=. onnx.proto
```

1.1.6 Autogenerated code

In order to avoid boilerplate code, some `.c` and `.h` files are autogenerated from Python. You can find these Python scripts and more information under `scrips/onnx_generator` folder. These Python scripts take into account a given ONNX version and generates all the "infrastructure" from it.

1.1.7 Versioning

ONNX is constantly evolving and new operators and versions are regularly being released. This project aims to keep up with that using the autogenerated code mentioned above.

1.1.8 Operators

All `onnx` operators are defined in the official doc. Our goal is to implement as many as possible, but if you have a model with a custom operator, that also fine.

All operators share a common interface. This structure is available as an input parameter to all operators, and it contains all the data you need, like the input tensors and attributes and a pointer to the output.

```
struct node_context{
  Onnx__NodeProto     *onnx_node;
  Onnx__TensorProto  **inputs;
  Onnx__TensorProto  **outputs;
  operator_executer resolved_op;
};
```

Lets say that you want to implement the `Add` operator, that just adds two numbers or tensors. You can access to its inputs with the following code. Once you have that, its pretty straightforward to access the data within the tensor (see `Onnx__TensorProto` struct)

```
Onnx__TensorProto *A = searchInputByName(ctx, 0);
Onnx__TensorProto *B = searchInputByName(ctx, 1);
```

On the other hand you will need to store the result in a variable, so that other nodes can reuse that output. Just use the following function and populate the content.

```
Onnx__TensorProto *C = searchOutputByName(ctx, 0);
```

If the operator you are implementing has some attributes, you can also easily get them with. Just replace `auto_pad` by your attribute name.

```
Onnx__AttributeProto *auto_pad = searchAttributeNyName(
  ctx->onnx_node->n_attribute,
  ctx->onnx_node->attribute,
  "auto_pad");
```

Its also important to note that each operator is usually defined for more than one data type like float, double or uint8_t. In order to address this, we have decided to define one function per data type, trying of course to share as many code as possible between different data types implementations.

1.1.9 Tracing

We utilize macros to enable fine tracing of components. Trace macros can act as asserts and therefore may abort execution if an erroneous state is detected.

If not explicitly enabled, all tracing components are stripped before compilation!

Trace prints look like this:

```
[MACRO LEVEL] FILE:LINE SCOPE MSG
```

- MACRO describes which macro produced the print
- LEVEL describes the minimum `TRACE_LEVEL` needed to generate this print
- FILE is the file in which the macro was called
- LINE is the line number in which the macro was called
- SCOPE is an optional block description
- MSG is the actual logged message

To enable tracing you need to specify a default `TRACE_LEVEL` starting from 0.

```
make clean all TRACE_LEVEL=0
```

|`TRACE_LEVEL`|Meaning |————:|:—— | undefined|disable tracing | 0|no prints, except for errors | 1|execution flow related prints (function entry/exists, decisions) | 2|data flow related prints (input/output meta data) | 3+|internal state of algorithms

The `TRACING_LEVEL` can be overwritten at file and function granularity without recompilation. When tracing is enabled, each macro checks the `CONNXR_TRACE_FILE` and `CONNXR_TRACE_FUNCTION` env variable for overrides.

Overrides are specified in following format:

```
CONNXR_TRACE_FILE=<filename>:<trace_level>{;<filename>:<trace_level>}
CONNXR_TRACE_FUNCTION=<function>:<trace_level>{;<function>:<trace_level>}
```

So you may execute

```
make clean all TRACE_LEVEL=0
CONNXR_TRACE_FUNCTION=operator__onnx__maxpool__12__T_tensor_float:3 make test_
↪operators
```

to only generate a detailed trace for a specific operator (maxpool in this case).

1.2 Contributing

This project is in very early state so we are looking for contributors. Feel free to open a PR or an issue with questions or suggestions. We would be also happy to guide you if you have an idea. You can contribute in many ways, but the most common is to implement a new operator. The `onnx` specification defines more than 150 operators, that are supported for different types (float, int, double,...) and for some of them there are different versions. Some common ones are implemented with limited functionality like convolutions, matrix multiplications and so on, but there are lots of them remaining.

You can also contribute by improving an existing operator or fixing some bugs. Feel free to have a look to the opened issues, where we also have some simple issues for newcomers.

1.2.1 Add new operator

If you want to add a new operator, we provide a simple and generic interface that you can use, with all the information in place ready to be used. The interface is the following. You can check here more information about each struct. In `onnx_node`, `inputs` and `outputs` you will have all the information that you need such as the inputs of the operators with its names, the attributes and where to write the output. Refer to ONNX Operators to check the inputs/outputs/attributes of the operator you are implementing. Note that some might be optional.

```
struct node_context{
  Onnx__NodeProto     *onnx_node;
  Onnx__TensorProto  **inputs;
  Onnx__TensorProto  **outputs;
  operator_executer resolved_op;
};
```

First of all decide the operator that you want to implement. You can see the official onnx operators list. Lets say that you want to implement the `Abs` operator, fair enough.

Since each operator has different versions, you would need to choose the version that you are implementing. The differences can be rather tiny, but its not the same Abs-13 operator than Abs-6. Note that the number is the onnx version that introduced that operator. We recommend implementing the latest one available at the moment.

Once you know the operator and version, there is one extra "dimension" to take into account. Most of the operators work with different data types, so you must choose which one to implement. Going back to the `Conv` example, we would suggest to start implementing the `float` version. Note that `Conv` is defined for double, float and float16. The code would be of course quite similar, but since we are in C we can't just use the same functions for all types.

Lets see how you can implement a new operator in few simple steps.

1. Generate the files

You don't have to create any files, just do a small modification in the `Makefile` and run a script. The files that you need will be generated automatically:

- If you want to implement a new operator, go to the makefile and add a new line with the operator that you want to generate, i.e. ONNX_INCLUDE+="^Add$$".
- Once you have that, and assuming that you have the latest onnx Python version installed, run `make onnx_generator`.
- The previous step will generate all files that are needed. It will also update the operator_sets.c file and create a resolver for that operator. You don't really need to care about this.
- Now you can populate the .c files with your implementation. Note that there is one implementation per data type (i.e. float, double,...)

Lets see some of the files that `Add` operator has:

- `operator__onnx__add__7__T_tensor_double.c`: The `7` indicates the operator version and `double` indicates the type. Add operator will have other files such as `_int32` or `_int64`. The implementations would be different, but they can share most of the code. You have of course to write in these file the actual implementation of the operator.
- `resolve_operator__onnx__add__7.c`: You don't need to touch this file. It justs maps (resolves) the function that an operator needs based on the input time.

2. Implement the operator

Now that you have generated all the files you are ready to start implementing the operator. Lets say that you want to implement `operator__onnx__add__7__T_tensor_float.c`, which is the `Add` operator for opset version `11` and type `float`.

This operator is adding two values or tensors so first of all you need these values. According to the specification this operator takes two inputs `A` and `B` with no attributes. There are two different ways that you can access them: by index or by name.

Inputs

You can **access by index** as follows. Note that `ctx->onnx_node->input[n]` doesn't contain the tensor, but just a name of the tensor.

```
// Access by index
Onnx__TensorProto *A = ctx->inputs[0];
Onnx__TensorProto *B = ctx->inputs[1];
```

You can also **access by name** with `searchInputByName()`. This function takes an index with the name of the tensor that wants to be accessed.

```
// Access by name
Onnx__TensorProto *A = searchInputByName(ctx, 0);
Onnx__TensorProto *B = searchInputByName(ctx, 1);
```

Both ways are perfectly valid, but the **access by index** is preferred since it doesn't need to search. But note that in some cases it can't be used. Imagine an operator with two inputs, but the second one is optional. Using `ctx->inputs[1]` will only work if the second input is provided, and will fail when it doesn't. For this cases, use `searchInputByName()` that will return NULL.

Note: There is some ongoing work in here, so might change the way it works.

At this point you have `A` and `B` ready to be added. Both variables belong to `Onnx__TensorProto` so feel free to have a look here. You can access the elements of the tensor like this, assuming that the type stored in it is `float`.

```
for (int i = 0; i < A->n_float_data){
        //A->float_data[i]
}
```

Attributes

If the operator you are implementing has some attributes, you can also easily get them with. Just replace `auto_pad` by your attribute name.

```
Onnx__AttributeProto *auto_pad = searchAttributeNyName(
  ctx->onnx_node->n_attribute,
  ctx->onnx_node->attribute,
  "auto_pad");
```

Same than before applies here. You can access the attributes **by name** or **by index**

Access by index:

- If there is only 1 attribute, and that attribute is mandatory, this way can be used. Not the case of `LeakyRelu` because that attribute could be empty (and the default value will be taken).
- If there are more than 1 attribute but all of them are mandatory, we can also use this way. If some attributes are mandatory this can't be done.

```
// Access by index
Onnx__AttributeProto *a_alpha = ctx->onnx_node->attribute[0]->f;
```

And this other way can be used in the rest of the cases.

```
// Access by name
Onnx__AttributeProto *a_alpha = searchAttributeNyName(ctx->onnx_node->n_attribute,ctx-
↪>onnx_node->attribute, "alpha");
if (a_alpha) {
     alpha = a_alpha->f;
}
```

Outputs

Last but not least, you will need to store the result in a variable, so that other nodes can reuse that output. Just use the following function and populate the content.

You can do it also **by index**. This is the way to go if there if only one output that is mandatory.

```
Onnx__TensorProto *C = ctx->outputs[0];
```

And in more complex cases where you can optional outputs or more than one, you can use the following.

```
Onnx__TensorProto *C = searchOutputByName(ctx, 0);
```

3. Test the operator

Once the operator is implemented, you are ready to test it. Luckily, onnx provides a set of test vectors for all operators, and we have taken care of integrating them, so you just need to go to `tests.c` file and uncomment the test case of your operator. For example, if you have implemented the `Shrink` operator, you should uncomment `test_shrink_hard` and `test_shrink_hard`. Its important to note that this test cases are not testing all data types (float, double,...). Most of these test cases run on float, but not always.

Currently `onnx` only provides test cases for the latest operator, so if you want to test and old version of an operator, you will have to do some manual work. However, there is some ongoing work in here to address this issue.

You can have a look to this PR that contains an example of how an operator can be implemented.

1.2.2 Other contributions

You can also contribute in other ways, like improving an existing operator, fixing bugs or writing documentation.

1.3 Testing

1.3.1 Testing

Everything is tested using Python `unittest`, thats runs the C code under the hood. Note that `onnx` provides data for testing a backend implementation. Have a look to https://github.com/onnx/onnx/blob/master/docs/OnnxBackendTest.md and https://github.com/onnx/onnx/tree/master/onnx/backend/test/data/node. In this link you will find sets of inputs/outputs + a `model.onnx` that allows to test both an individual operator or a whole model, hence we divide two different types of tests:

- Operator Tests: Tests an individual operator. Lets see a simple example. We want to test the operator `Sum` over a one dimension array. So we provide a set of two inputs `X1 = [1 1]` and `X2 = [2 2]` and an expected output `O = [3 3]`. We also provides a model with just one two inputs an a node which operator is `Sum`. All these inputs/outputs
- Model Tests: Runs inference using a model. As an example we can use MNIST model (a digit recognition library). The input is an image with a written digit and the output is an int `[0-9]`. This type of tests run several operators.

1.3.2 Operator Tests

You can run all operator tests with the following command. This command just uses `tests/test_operators.py` script under the hood.

```
make test_operators
```

You can run a single test as follows:

```
python3 tests/test_operators.py TestOperators.test_relu
```

1.3.3 Model Tests

Different end to end models are implemented and tested, and you can run them all with the following command (make sure you can compiled before `make all`). Note that this uses `tests/test_models.py` under the hood.

```
make test_models
```

You can also run a single model with the following command:

```
python3 tests/test_models.py TestModels.test_mnist
```

1.3.4 Debugging tools

If you are implementing a new model but you can get the end to end test cases to pass, you might want to use some of the tools that we have. Using the CLI, you can provide the option `--dump-file`. This will dump all the intermediate outputs to a file, that you can use later on to compare against the expected values.

The `dump.txt` file will look something like this.

```
name=output_name
shape=shape1,shape2,...
tensor=value1,value2,value3,...
name=output_name
shape=shape1,shape2,...
tensor=value1,value2,value3,...
...
```

So in this file you have all the data that cONNXr has calculated for every node. Now you can compare this against the expected outputs of that model. For this purpose, we have `scripts/assert_nodes.py`. This script gets the output tensor that cONNXr calculated, and compares it against the one calculated by the official ONNX backend. This is a great tool if you want to find the node that is causing problems. This script will point to the node that is not matching the expected values.

Note that this feature is under development.

Example how to dump data

```
build/connxr test/mobilenetv2-1.0/mobilenetv2-1.0.onnx test/mobilenetv2-1.0/test_data_
↪set_0/input_0.pb --dump-file
```

Example how to use the Python script:

TODO

```
python3 scripts/assert_nodes.py
```

1.4 Onnx Operator Status

1.4.1 Onnx Operator Status

This will be generated using the tests as reference.

- ai.onnx
  - not_implemented Abs
  - not_implemented Acos
  - not_implemented Acosh
  - implemented Add

- not_implemented And
- implemented ArgMax
- not_implemented ArgMin
- not_implemented Asin
- not_implemented Asinh
- not_implemented Atan
- not_implemented Atanh
- not_implemented AveragePool
- implemented BatchNormalization
- not_implemented BitShift
- implemented Cast
- not_implemented Ceil
- implemented Clip
- not_implemented Compress
- not_implemented Concat
- not_implemented ConcatFromSequence
- not_implemented Constant
- not_implemented ConstantOfShape
- implemented Conv
- implemented ConvInteger
- not_implemented ConvTranspose
- not_implemented Cos
- not_implemented Cosh
- not_implemented CumSum
- not_implemented DepthToSpace
- not_implemented DequantizeLinear
- not_implemented Det
- not_implemented Div
- not_implemented Dropout
- not_implemented Einsum
- implemented Elu
- not_implemented Equal
- not_implemented Erf
- not_implemented Exp
- not_implemented Expand
- not_implemented EyeLike

- not_implemented Flatten
- not_implemented Floor
- not_implemented GRU
- not_implemented Gather
- not_implemented GatherElements
- not_implemented GatherND
- not_implemented Gemm
- implemented GlobalAveragePool
- not_implemented GlobalLpPool
- not_implemented GlobalMaxPool
- not_implemented Greater
- not_implemented HardSigmoid
- not_implemented Hardmax
- implemented Identity
- not_implemented If
- not_implemented InstanceNormalization
- not_implemented Inverse
- not_implemented IsInf
- not_implemented IsNaN
- not_implemented LRN
- not_implemented LSTM
- implemented LeakyRelu
- not_implemented Less
- not_implemented Log
- not_implemented LogSoftmax
- not_implemented Loop
- not_implemented LpNormalization
- not_implemented LpPool
- implemented MatMul
- implemented MatMulInteger
- not_implemented Max
- implemented MaxPool
- not_implemented MaxRoiPool
- not_implemented MaxUnpool
- not_implemented Mean
- not_implemented Min

- not_implemented Mod
- implemented Mul
- not_implemented Multinomial
- not_implemented Neg
- not_implemented NonMaxSuppression
- not_implemented NonZero
- not_implemented Not
- not_implemented OneHot
- not_implemented Or
- not_implemented PRelu
- not_implemented Pad
- not_implemented Pow
- not_implemented QLinearConv
- not_implemented QLinearMatMul
- implemented QuantizeLinear
- not_implemented RNN
- not_implemented RandomNormal
- not_implemented RandomNormalLike
- not_implemented RandomUniform
- not_implemented RandomUniformLike
- not_implemented Reciprocal
- not_implemented ReduceL1
- not_implemented ReduceL2
- not_implemented ReduceLogSum
- not_implemented ReduceLogSumExp
- not_implemented ReduceMax
- not_implemented ReduceMean
- not_implemented ReduceMin
- not_implemented ReduceProd
- not_implemented ReduceSum
- not_implemented ReduceSumSquare
- implemented Relu
- implemented Reshape
- not_implemented Resize
- not_implemented ReverseSequence
- not_implemented RoiAlign

- not_implemented Round
- not_implemented Scan
- not_implemented Scatter
- not_implemented ScatterElements
- not_implemented ScatterND
- not_implemented Selu
- not_implemented SequenceAt
- not_implemented SequenceConstruct
- not_implemented SequenceEmpty
- not_implemented SequenceErase
- not_implemented SequenceInsert
- not_implemented SequenceLength
- not_implemented Shape
- not_implemented Shrink
- stubbed Sigmoid
- not_implemented Sign
- not_implemented Sin
- not_implemented Sinh
- not_implemented Size
- not_implemented Slice
- implemented Softmax
- not_implemented Softplus
- not_implemented Softsign
- not_implemented SpaceToDepth
- not_implemented Split
- not_implemented SplitToSequence
- not_implemented Sqrt
- not_implemented Squeeze
- not_implemented StringNormalizer
- not_implemented Sub
- not_implemented Sum
- not_implemented Tan
- not_implemented Tanh
- not_implemented TfIdfVectorizer
- not_implemented ThresholdedRelu
- not_implemented Tile

  - not_implemented TopK
  - not_implemented Transpose
  - not_implemented UnfoldToDepth
  - not_implemented Unique
  - not_implemented Unsqueeze
  - not_implemented Upsample
  - not_implemented Where
  - not_implemented Xor
  - Functions
    - not_implemented Celu
    - not_implemented DynamicQuantizeLinear
    - not_implemented GreaterOrEqual
    - not_implemented LessOrEqual
    - not_implemented MeanSquaredDistance
    - not_implemented MeanVarianceNormalization
    - not_implemented NegativeLogLikelihoodLoss
    - not_implemented Range
    - not_implemented SoftmaxCrossEntropyLoss
- ai.onnx.training
  - not_implemented Adagrad
  - not_implemented Gradient
  - not_implemented GraphCall
  - not_implemented Momentum
- ai.onnx.ml
  - not_implemented ArrayFeatureExtractor
  - not_implemented Binarizer
  - not_implemented CastMap
  - not_implemented CategoryMapper
  - not_implemented DictVectorizer
  - not_implemented FeatureVectorizer
  - not_implemented Imputer
  - not_implemented LabelEncoder
  - not_implemented LinearClassifier
  - not_implemented LinearRegressor
  - not_implemented Normalizer
  - not_implemented OneHotEncoder

- not_implemented SVMClassifier
- not_implemented SVMRegressor
- not_implemented Scaler
- not_implemented TreeEnsembleClassifier
- not_implemented TreeEnsembleRegressor
- stubbed ZipMap