Eötvös Loránd University

Faculty of Informatics

Dept. of Programming Languages and Compilers

# Transformation of Neural Networks

*Author:*
Ágoston Mátyás Czobor
Computer Science BSc

*Internal supervisor:*
Norbert Pataki
Docent

*External supervisor:*
Viktor Gyenes
AiMotive
Lead AI Research Engineer

*Budapest, 2024*

# Contents

# Acknowledgement

I would like to express my gratitude to my external supervisor, Viktor Gyenes, for his support for this topic, helping to reinterpret the operations with his immense knowledge of neural networks. Also aiMotive, for allowing me, to bring this project, as my thesis topic, and for their support with the paperwork needed.

I am also grateful to my internal supervisor, Norbert Pataki, whose tenacity in correcting my mistakes was endless, helping me write a better documentation. His ideas and sensible thinking helped me see a clearer goal.

# 1 Introduction

With the astonishing growth of AI, its presence has been growing more and more. This caused multiple representations to emerge, all setting out to create something new, something different or specialized. This forking caused high variance in the representative languages, making translation between the representative languages difficult and direct export-import even more so. Multiple groups and developers recognized this fragmentation and aim to solve this issue, either by creating an intermediary software or by developing APIs to support different frontends and/or backends.

The two pieces of software that this paper will be exploring are the Neural Network Exchange Format (referred to as NNEF) at The Khronos Group [1] and Apache TVM (henceforth TVM) at The Apache Software Foundation [2, 3].

The goal was to create an NNEF frontend to TVM, using the provided Python API for frontends. For that, I had to create a suitable conversion library that mapped the functions, nodes, and operators of NNEF to their equivalent counterparts in TVM, or provide a method of calculating the desired results. For this, I created the NNEF-to-TVM-converter package, which, at the time of writing, is still in the process of being merged into the official TVM GitHub repository [4]. Currently, the package can be used outside of the TVM source code by importing the necessary packages, but this will be a deprecated use case after it is part of the TVM repository.

This project was completed as part of my job duties at aiMotive [5]. The company has already integrated NNEF as their network representation and needed a suitable compiler and optimizer suite, which TVM provides. Because of this, the development contained a more in-depth review process, with unit, integration, lint, and complete model tests.

I worked on this project alone, under the supervision of my lead, Viktor Gyenes.

# 2 Fundamentals of NNEF and TVM

This chapter will give a basic overview of NNEF and TVM, the two software that were used for the transformation.

## 2.1 Terminology

This section will present a quick overview of the terms used in the thesis. These will contain the neural network, NNEF, and TVM related terms.

**neural network**
: A computational graph, consisting of tensor operations. It has input(s) and output(s), it's nodes, the operations consist of tensor inputs and outputs, along with scalar attributes.

**tensor**
: A multi-dimensional array of scalars (or other types) that represents data flow in the graph. The number of dimensions is conceptually infinite; the insignificant trailing dimensions are 1 (singleton dimension).

**attribute**
: A non-tensor parameter to operations that define further details of the operation. Attributes are of primitive types whose values are known at graph compilation-time, and hence expressions of attributes can be evaluated at graph compilation-time.

**compound operation**
: An operation that is defined in terms of other operations. Its semantics are defined via the composition of the operations that

**computational graph**
: A graph with nodes that are either operations or tensors. Operation nodes are connected to tensor nodes only, and vice versa.

**graph compilation-time**
: The time when the graph is built before execution. Analogous to compilation-time for programming languages.

**graph execution-time**
: The time when the graph is run (possibly multiple times) after building. Analogous to run-time for programming languages.

**operation**
: A mapping of input tensors to output tensors. Refers to NNEF operations.

**rank (of a tensor)**
: The number dimensions of a tensor explicitly specified in a shape or implicitly defined by shape propagation. Note that a shape explicitly defined as (1,1) has rank 2, even though its dimensions are singular.

**shape (of a tensor)**
: A list of integers defining the extents of a tensor in each relevant dimension.

**volume (of a tensor)**
: An integer value that is the product of extents of a shape.

## 2.2 Neural Network Exchange Format

The section presents the NNEF, by The Khronos Group [1].

### 2.2.1 General overview

NNEF aims to lessen the fragmentation of the industry by creating an intermediary representation between the standard neural network formats (such as Torch, Caffe, TensorFlow, Theano, Chainer, Caffe2, PyTorch, and MXNet) and inference engines, making interoperability easier while standardizing a representation, providing tools for importing and exporting models.

NNEF is only a representative format; it does not contain the proper tools for network execution. It has a C++ execution module, which is not optimized for large graphs, only written for test cases, and a Python interpreter (that was used as a baseline NNEF output for the test cases) which is a PyTorch wrapper, unsuited for proper use on large scales. That is why the task of integrating it into TVM was crucial, that being a proper compiler framework, with runtime modules, and a complex toolset to train, compile, optimize, and run models for a wide range of targets.

**Alternative** NNEF is similar to another format, Open Neural Network Exchange (henceforth ONNX), initiated by Microsoft and Meta, which also tries to stand as a general network format. The difference is in the aim of the development of the projects.

ONNX updates quickly, keeping up with every new framework update, while NNEF provides a stable, strong-standing foundation for compatibility and reliability. They rather complement each other, than compete.

This paper will mainly discuss the Python package of NNEF, therefore moving forward the discussion will refer to Python concepts, not the C++ backend.

### 2.2.2 Data representation

NNEF provides a way to store the model's files in a contained manner. It uses a multiple-file approach, where one file contains the operations, and their parameters (the nodes and edges, creating a simple graph without edge weights), and multiple weight files. The weights can be read into memory, called variables in the NNEF standard. They are usually pre-trained tensors - the weights of kernels, biases, constant multiplicators, etc. for the model in question.

#### Model

An NNEF model consists of an enclosing directory, a `graph.nnef` file, and optionally any number of subdirectories, containing weights - tensor data files. This graph can then be loaded in, e.g. via `nnef.load_graph`, for use in a program.

An example model can be seen in Figures 2.2.2.1 and 2.2.2.2.

**graph.nnef** The graph file is a textual representation of the model in question. Among others, it specifies the input(s) and output(s) of the model, lists its operations, including input handling, variable loading operations, and their parameters and attributes.

**weight files** The weights are binary `.dat` files. It is capable of storing the values of the tensor, that can be read into the graph as variables, its parameters.

```
NNEF_Model_Directory
 ├── graph.nnef
 └── weights
      ├── conv_kernel.dat
      ├── conv_bias.dat
      └── ...
```

**Figure 2.2.2.1:** *Directory example of a simple NNEF model*

```
version 1.0;

graph main ( input ) -> ( output )
{
    input = external<scalar>(shape = [4, 3, 256, 256]);
    kernel = variable<scalar>(shape = [12, 3, 4, 4], label = 'weights/
        conv_kernel');
    bias = variable<scalar>(shape = [1, 12], label = 'weights/conv_bias
        ');
    conv = conv(input, kernel, bias);
    output = relu(conv);
}
```

**Figure 2.2.2.2:** *Basic Network in NNEF*

### 2.2.3 NNEF-Tools

The library supporting the NNEF framework is a toolset, called NNEF-Tools [6]. It contains tools to generate and consume NNEF documents, such as a parser (C++ and Python) that can be included in consumer applications and converters for deep learning frameworks.

**NNEF Tools** contains tools to convert pre-trained models in TensorFlow / Caffe / Caffe2 / ONNX to NNEF format, and vice versa.

**NNEF Parser** contains C++ and Python source code for a sample NNEF graph parser.

## 2.3 Apache TVM

This section presents Apache TVM, an open-source compiler framework [2].

### 2.3.1 General overview

Apache TVM is an open-source project [2, 3], it aims to be a framework for optimizing and deploying machine learning models across various hardware platforms. With the exponential growth in model complexity and the diversity of hardware architectures, there's a pressing need for a unified solution to bridge the gap between cutting-edge research and real-world applications.

TVM provides a compiler stack that translates machine learning models expressed in high-level frameworks such as TensorFlow, PyTorch, MXNet, and NNEF into efficient executable code for diverse hardware targets, including CPUs, GPUs, and specialized accelerators like FPGAs and TPUs. It uses optimization techniques such as graph-level and operator-level optimizations, auto-tuning, and hardware-aware scheduling for reaching peak performance executables.

It currently uses various representations, each corresponding to different stages of the program compilation or development process. In this paper, I will delve into more detail about Relay, one of the highest-level representations TVM uses along with Relax. They both represent the same level in the execution workflow. Relax, the name coming from Relay Next, supersedes Relay, by implementing dynamic-shaped tensor operations. They currently coexist, as Relax is not fully ready yet, so Relay has been placed as a maintain-only target. Additionally, there is the Tensor-level IR (TIR), which represents the next level of abstraction. TIR is still a human-readable language, but it is a low-level representation. The next step is the target-specific representation, only called IR, which is used in a runtime.Module. This is already a near-execute-ready library, containing platform-specific instructions. All Relay, Relax, and TIR utilize IRModule for their internal structure, the signature of which we will delve deeper into later. This paper will not go in-depth about the IR language but will introduce the necessary terminology, objects, and functions. The workflow of TVM can be seen in Figure 2.3.1.1, presenting the conversion between IRs.

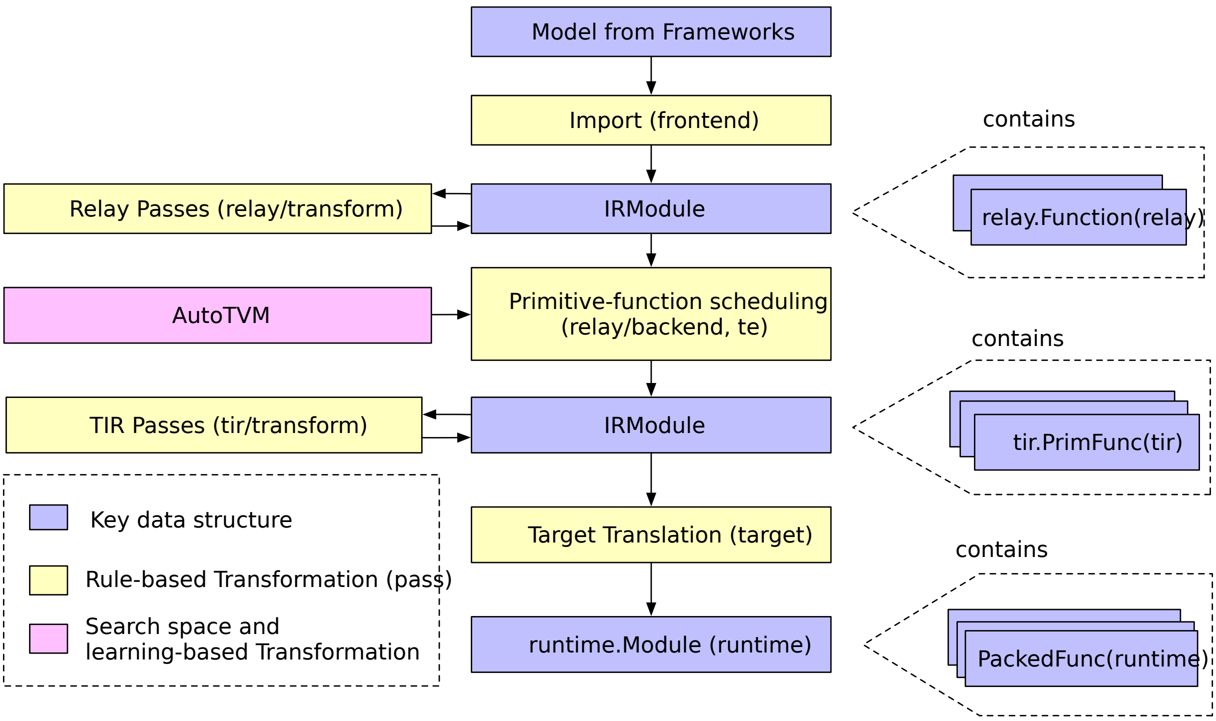


**Figure 2.3.1.1:** *TVM workflow*

## 2.3.2 Data representation

While both Relay and Relax use frontends for their main input source, they both come with a complete Python DSL, with a slightly different approach.

An example model can be seen in Figures 2.3.2.1 and 2.3.2.2.

**Relay** has a Python function library, that allows the easy construction of `relay.Functions`, showing pipelines and dataflow. With the help of this, it is easy to quickly create small graphs. Relay also has a text representation, for visually showing a model. More can be read about Relay IR in the TVM docs [7].

```
input = relay.var("input", "float32")
kernel = relay.const(kernel_data, "float32")
bias = relay.const(bias_data, "float32")
conv = relay.nn.conv2d(input, kernel)
conv = relay.nn.bias_add(conv, bias)
output = relay.nn.relu(conv)
main = relay.Function([input], output)
```

**Figure 2.3.2.1:** *A Basic Network in Relay - Python definition*

```
fn (%input: float32) {
  %0 = nn.conv2d(%input, meta[relay.Constant][0], padding=[0, 0, 0, 0]);
  %1 = nn.bias_add(%0, meta[relay.Constant][1]);
  nn.relu(%1)
}
```

**Figure 2.3.2.2:** *A Basic Network in Relay - Text representation*

**Relax** uses a different approach, its textual representation and language definition being the same, a Python-based scripting language, TVMScript. For further information about Relax, the reader is referred to the docs, hosted on MLC's servers for Relax [8].

The project at the beginning targeted creating an NNEF frontend for Relay, as that seemed the newest representation, but during the development process Relay Next, Relax came out, so the focus shifted towards a Relax frontend. In this Thesis, I will only delve into the Relay implementation of the converter.

# 3 User documentation

This chapter will present the thesis directory structure with a general overview, give installation methods for the required software, NNEF and TVM, and provide documentation for the converter program, along with an example use case.

## 3.1 Directory structure

In this section, I will present the overall folder structure of the thesis, visible in Figure 3.1.0.1. This is an abbreviated display of the directories, only presenting the encapsulating folders for installations and the standalone test folder. For more in-depth information, check Section 4.1.

```
Thesis
 ├─ Standalone
 │   └─ ...
 ├─ Integration
 │   └─ ...
 └─ Tests
     └─ ...
```

**Figure 3.1.0.1:** *Thesis directory structure*

**Standalone** The *Standalone* folder contains the standalone Python package for the program. It contains the necessary install scripts for pip to install the package. More about standalone installation in Section 3.3.1.

**Integration** The *Integration* folder contains the necessary files for integrating the program into TVM. The files that need modification are present in the structure, and need to be merged into a TVM install directory. More about integrated installation in Section 3.3.2.

**Tests** The *Test* folder contains the script file for running standalone tests, `test_standalone.py`, along with the test cases in the sub directory, `cases`. More about testing in Section 5.

## 3.2 Installation of NNEF and TVM

The NNEF-to-TVM converter (henceforth NNEFConverter) is currently in the process of being integrated into the Apache TVM compiler framework as a frontend for Relay. Until the process is finished, integrating into TVM is a more in-depth path, which many may avoid. The standalone installation will not be recommended after integrating officially but will be supported.

#### Dependencies

The program uses NNEF-Tools [6] and TVM [4], both programs are necessary. There will be a source install for NNEF, and either a PyPI or a source method for TVM. For these programs the minimal necessary dependencies are:

- Python (>= 3.8)
- Git

Installing NNEF-Tools is simpler, it only uses a Python package, for TVM we will show two installation methods; while installing from source is recommended, a package installation route will also be provided.

### 3.2.1 Installing NNEF

Since NNEF-Tools is in the process, of publishing a PyPI package, the recommended method is to install it from the GitHub repository [6]. Figure 3.2.1.1 presents how the user can get the repository from GitHub.

```
git clone https://github.com/KhronosGroup/NNEF-Tools.git NNEF-Tools
```

**Figure 3.2.1.1:** *Clone NNEF-Tools repository*

Figure 3.2.1.2 presents how to install the NNEF Python packages, `nnef` and `nnef_tools`.

```
# Step into NNEF-Tools directory
cd NNEF-Tools
# install NNEF-Tools
pip install .
# install NNEF-Parser
pip install ./parser/python/
```

**Figure 3.2.1.2:** *Install NNEF-Tools and NNEF-Parser*

With this, NNEF-Tools and NNEF-Parser are installed in the active Python environment.

#### PyPI package

During the last phases of the thesis, I published the first packages to PyPI, and from now on `nnef_tools`, and `nnef` should be installable via pip, as presented in Figure 3.2.1.3.

```
pip install nnef nnef_tools
```

**Figure 3.2.1.3:** *Install NNEF-Tools and Parser via pip*

Note that for a more stable version, the main branch of the repository should be used.

### 3.2.2 Installing TVM

As mentioned above, there will be both a PyPI and a source install method presented for TVM.

#### Package install

TVM has Python Packages built by the community for ease of use, which are a quick and easy way to start using TVM. This method has no additional dependencies; pip will automatically collect the required libraries. TVM only has partial support via pip packages, the build is strongly advised from source. For that, the reader is referred to the next section, Install from source, for more information.

These packages are mostly built for demo purposes by the community and can be reached via TLCPack [9].

Figure 3.2.2.1 presents an example (default) install command.

```
# Linux/MacOS CPU build only!
pip install apache-tvm
```

**Figure 3.2.2.1:** *Pip command for default PyPI package*

Note that this command only installs Linux/macOS, CPU build only. For commands for Windows or other packages, check the TLCPack website, `https://tlcpack.ai`. Some version, system, and CUDA combinations are not supported via this method and need to be built from source.

**Install from source**

TVM has a dedicated page for installation instructions. The paper will provide a summarization, but the user is strongly recommended to read the instructions in the official documentation at `https://tvm.apache.org/docs/install/from_source.html`.

This paper will provide the instructions for a CPU build, which uses LLVM codegen.

Installing TVM from source has additional dependencies the user has to provide, the minimal requirements are:

- CMake (>= 3.24.0)
- LLVM (>= 15)
- A recent C++ compiler supporting C++ 17, at the minimum
  - GCC 7.1
  - Clang 5.0
  - Apple Clang 9.3
  - Visual Studio 2019 (v16.7)

The user needs to clone the GitHub repository as visible in Figure 3.2.2.2.

```
# clone needs to be recursive
git clone --recursive https://github.com/apache/tvm tvm
```

**Figure 3.2.2.2:** *Clone TVM repository*

Figure 3.2.2.3 presents how to prepare the configuration file for CMake:

```
cd tvm
mkdir build
cp cmake/config.cmake build
```

**Figure 3.2.2.3:** *Prepare build directory*

The user can now edit the `build/config.cmake` file, changing the setting `set(USE_LLVM ON)` to use LLVM codegen. Then build TVM via the process presented in Figure 3.2.2.4

```
cd build
cmake ..
# use of parallel processes is recommended
make -j$(nproc)
```

**Figure 3.2.2.4:** *Make TVM*

Then, Figure 3.2.2.5 shows how to install the TVM Python package:

```
# path/to/tvm is the root directory of tvm
pip install path/to/tv/python/
```

**Figure 3.2.2.5:** *Install TVM Python package*

## 3.3 Installation of NNEFConverter

The program can be used as both a standalone package or can be manually integrated into TVM, provided the user installed from source. The functionality does not differ between the installation methods. For in-depth testing, integration into TVM is recommended, but standalone tests are also provided.

The paper will use the standalone installation's package notation, `NNEFConverter.from_nnef`, instead of the TVM package notation, `tvm.relay.frontend.from_nnef`, but they are interchangeable, depending on the install type.

### 3.3.1 Standalone install

The standalone version of the program can be found under the `Standalone` directory of the project. It needs to be installed via pip and can be used via the `NNEFConverter` package as shown in Figure 3.3.1.1

```
# path/to/project is the root directory of the project
pip install Standalone/
```

**Figure 3.3.1.1:** *Install the NNEFConverter package*

Installing via this method adds the NNEFConverter module to Python. The module has a single public function, `from_nnef`.

### 3.3.2 Integrate to TVM

The integration can be easily done by copying the `Integration` folder into TVM's root directory. The directory contains all files that need modification; therefore, the operating system will warn the user that files will be overwritten.

It is necessary to reinstall the Python package of TVM as presented in Figure 3.3.2.1.

```
# path/to/tvm is the root directory of tvm
pip install path/to/tvm/python/
```

**Figure 3.3.2.1:** *Install TVM Python package*

Later, this install method will be deprecated, as the pull request to the TVM repository will be accepted, and then this project will be supported as an official TVM frontend.

## 3.4 Use of NNEFConverter

As mentioned previously, the following example codes in the documentation will use the standalone version's notation, `NNEFConverter`, as the Python module name. However, the provided example scripts should run in the case the integrated version is used, as long as the module name `tvm.relay.frontend` is utilized.

Currently, the use of NNEFConverter is only possible through Python.

**NNEFConverter.from_nnef** is the public function to convert an NNEF model into a TVM IRModule and an accompanying parameter dictionary.

**Parameters**

- `model`: The function grants the possibility of either passing an `os.PathLike`, or str of the path of the NNEF model directory, or an `NNEF.Graph` object as well.
- `freeze_vars`: It also has a parameter that informs the converter whether it should convert the NNEF variables into Relay constants, and fold them into the resulting IRModule, or leave them as external variables, returned as a dictionary.

**Return value**

- The function returns a tuple, that has two elements.
  - The first element is always a `tvm.IRModule`, containing the computational (or dataflow) graph, and constant variables.
  - The second is either a dictionary of str and `tvm.nd.array`, representing the variables. The string name is the key for the dict, and values are contained in an array. The value can also be `None` if there were no parameters in the graph, or the `freeze_vars` switch was used.

### 3.4.1 Example use

In this section, I will present an example use of the program, from loading the graph to execution, showing the Python script file in Figure 3.4.1.1. The model I will be using is an InceptionNet v1. The NNEF model can be downloaded from the NNEF ModelZoo [10]. We will use the `graph_executor` submodule of TVM for compiling, and build for CPU with LLVM codegen.

We will use some assumptions for values:

- `model` : As mentioned above, the model is an Inception v1 model. We will assume that its model directory was placed in the working directory.
- `X` : X will represent a `numpy.ndarray`, containing an image, which will be the input of the model. The image is a single RGB separated float value matrix of 224x224 pixels, so the array has a shape of [1, 3, 224, 224].

```python
import NNEFConverter
import tvm
import tvm.relay as relay
from tvm.contrib import graph_executor

# Call the NNEFConverter function, to generate TVM module, and parameters, containing the weights from NNEF
# Provide the path of the directory of the NNEF model
mod, params = NNEFConverter.from_nnef("inception_v1.nnef")


# tvm uses target to specify build target(s)
target = tvm.target.Target("llvm")


# relay.build generates a tvm library with target specific instructions
with tvm.transform.PassContext(opt_level=3):
    lib = relay.build(mod, target=target, params=params)

# tvm uses device to specify the device used to compile the module
dev = tvm.device(str(target), 0)

module = graph_executor.GraphModule(lib["default"](dev))

# we set the input, X, for the module, loading it into memory
module.set_input("data_0", X)


# Execute model
module.run()

# extract the output
tvm_output = module.get_output(0).numpy()
```

**Figure 3.4.1.1:** *Conversion and execution of Inception v1*

This example is just a very basic example, as the strength of using TVM as a compiler is the ability to tune the model or create executables for different targets, but unfortunately discussion about that is outside the scope of this paper.

### 3.4.2 Possible errors and warnings during use

**Warnings**

With some operators that use border styles, there can be warnings, that `’Currently ignore border is not supported, used ‘constant‘ border’`. While NNEF supports different border styles (constant, replicate, reflect, reflect-even) for every sliding window operation, TVM mostly supports a constant border. This should not cause much difference in most operators, and where it does, they have been implemented to work correctly.

In NNEF, some operators (avg_pool, LRN) have the possibility to nominate multiple axes to apply the operation over, but even the specification does not require Inference APIs to implement them for over one active axis at a time. In this case, the user will be notified that multiple axes are not supported, the operation used the first valid axis.

**Errors**

As the conversion does not fully cover every NNEF operation, there can arise a case when a network can not be converted via NNEFConverter. The coverage is above 90%, the remaining part being not frequently used operations, which do not even have a TVM counterpart. NNEF supports these operations because of its development strategy, for backward compatibility, they have no real-world usage anymore, the models use better operations for the same goal. If encountered, the conversion will fail with an `Exception`, containing the message `’Not supported operator was called, please check for compatibility’`.

If the user were to try to use `matmul` on tensors whose dimensions are not broadcastable, the conversion will also fail with the message `’Batch dimensions are not broadcastable’`.

Other conversion errors should not arise, given that the input `nnef.Graph` is a valid, correct graph. Future development of NNEF might raise errors, but those are not user errors as they derive from the development of said software.

# 4 Developer documentation

In this chapter, I abbreviate the name of `relay` to `relay`, as it is recommended and used by the TVM docs.

## 4.1 Directory structure

In this section, I present the directory structure of NNEFConverter.

### 4.1.1 Standalone Python package

The `Standalone` directory, shown in Figure 4.1.1.1, has the script files necessary to install the Python package, NNEFConverter. No additional files are required for use.

Note that the integrated version uses the same `from_nnef.py`, renamed to `nnef.py` and `nnef_ops.py` files, just in a different folder structure.

```
Standalone
├── NNEFConverter
│   ├── __init__.py
│   ├── from_nnef.py
│   └── nnef_ops.py
└── setup.py
```

**Figure 4.1.1.1:** *NNEFConverter package structure*

`setup.py` provides the install information for pip, with `setuptools`.

`__init__.py` is the package initialization script. It uncovers the main method of NNEFConverter, `from_nnef`, so it can be called from the package level.

`from_nnef.py` contains the graph-level logic of NNEFConverter. It provides the necessary methods to deconstruct an `nnef.Graph` and construct a relay `tvm.IRModule`. This file contains the `from_nnef` function as well. It imports `nnef_ops.py` for the operation converting functions. In-depth documentation for the function can be found in Section 4.2.3.

`nnef_ops.py` contains the operation converter functions as a library. It contains only the methods needed to convert the NNEF data structures into TVM ones. In-depth documentation for the conversion functions can be found in Section 4.3.

### 4.1.2 Integration directory

The `Integration` directory, shown in Figure 4.1.2.1, contains the frontend itself, and the files that are vital to be modified for the frontend to work properly. By merging the folder into the TVM root directory, the NNEF frontend will be added to the Python API of TVM. The folder structure is important, as it mirrors TVM's, making the integration trivial.

Note that the standalone version uses the same `nnef.py`, renamed to `from_nnef.py` and `nnef_ops.py` files, just in a different folder structure.

**python**
This directory contains the new files that are added to TVM, the frontend itself, the rest being install scripts, testing scripts, linter corrections.

The files have been placed on the correct path for frontends in TVM, `tvm/relay/frontend/`. `__init__.py` scripts had to be modified, to import the NNEF frontend to the package, uncovering for use. The `nnef.py` and `nnef_ops.py` files contain the functions for the frontend, their contents identical to their standalone counterparts in Section 4.1.1.

**docker**
The `docker` directory contains the necessary scripts to build Docker images of TVM, for ease of deployment. The setup scripts for them must also be changed to include the NNEF-Tools, and NNEF-Parser to the images. The `ubuntu_install_nnef.sh` shell script is responsible for the git cloning and installing of the NNEF software. The other two files are used by TVM's Docker image building script, they specify which requirements are necessary for the task at hand, so running the aforementioned installation script in them is mandatory.

**tests**
TVM has automated tests for Continuous Integration (CI) and Continuous Delivery (CD) pipelines, whose setup files, definition files, and test cases are located in the `tests` folder. There are a multitude of tests, from lint checks to unit tests, on every target type TVM can use.

```
Integration
├── docker
│   ├── install
│   │   └── ubuntu_install_nnef.sh
│   ├── Dockerfile.ci_cpu
│   └── Dockerfile.ci_gpu
├── python
│   └── tvm
│       └── relay
│           └── frontend
│               ├── __init__.py
│               ├── nnef.py
│               └── nnef_ops.py
└── tests
    ├── lint
    │   ├── check_file_type.py
    │   └── rat-excludes
    ├── python
    │   └── frontend
    │       └── nnef
    │           ├── cases
    │           │   ├── test case graph dirs
    │           │   └── ⋮
    │           └── test_forward.py
    └── scripts
        ├── task_python_frontend.sh
        └── task_python_frontend_cpu.sh
```

**Figure 4.1.2.1:** *Integration directory structure*

In my case, I had to modify the lint scripts, so it accepts the `.nnef` graph definition files. For that, both the `check_file_type.py` script and the exclude list for Apache RAT, in `rat-excludes` had to be modified.

For unit tests, I had to add the script file containing the test execution script, `test_forward.py`, and the graphs for test cases themselves (under `cases`).

The scripts that run the tests themselves for CI-CD are `task_python_frontend.sh` for GPU tests, and `task_python_frontend_cpu.sh` for CPU tests. They have been updated as well to include the NNEF frontend tests.

More about tests can be checked in Section 5.

### 4.1.3 Tests

Figure 4.1.3.1 presents the structure of the `Tests` directory, that contains the test suites for NNEFConverter, so testing can be done with the standalone install as well.

```
Tests
├── cases
│   ├── test case graph dirs
│   └── ⋮
└── test_standalone.py
```

**Figure 4.1.3.1:** *Standalone tests directory structure*

**test_standalone.py** file contains the required pytest scripts, for setting up the environment for the tests. It loads in the test graphs from the `cases` folder, then executes them on Relay, the expected output being the results of running the same graph with the NNEF PyTorch interpreter.

**cases** folder contains 301 directories, each representing a NNEF graph, defining a test case. The `test_standalone.py` script collects the graph definitions from this folder. The naming convention of the cases follows:

$operation\_dimension/kernel\ size\_optional(e.g\ padding, groups, standalone)$.

More about testing can be read in Section 5.

## 4.2 Graph conversion

In this section, I will introduce the graph representations of the two external software, as well as the logic of converting between them. It will detail the motivations for the converting strategy and the structure of the file, the `from_nnef.py`.

### 4.2.1 NNEF Graph

NNEF Parser loads graphs into `nnef.graph` objects, presented as class diagram in Figure 4.2.1.1, containing the operations, their inputs and outputs, and the tensors, with their data.

| Graph |
| --- |
| _operations : Operation[]<br>_tensors : Tensor[]<br>_inputs : Tensor[]<br>_outputs : Tensor[]<br>_name : str |
| + __init__(name: str)<br>+ name : str<br>+ operations : Operation[]<br>+ tensors : Tensor[]<br>+ inputs : Tensor[]<br>+ outputs : Tensor[]<br>⋮ |

**Figure 4.2.1.1:** *nnef.Graph class diagram*

`name` The string name of the graph.

`operation` A `list`, containing `nnef.Operation` instances, that represent an operation, with its name, attributes, and data type (dtype).

`tensors` A `dict` containing string identifiers as keys, and their `nnef.Tensor` value pairs. This can contain the data of the tensors, loaded from external `.dat` data files.

`inputs` A `list` containing every graph level input by the identifiers' string representation.

`outputs` A `list` containing every graph level output by the identifiers' string representation.

For more in-depth information about the `nnef.Graph`, refer to NNEF Specification on the Khronos Registry [11], or the NNEF-Tools GitHub repository [6].

### 4.2.2 Relay Function

As mentioned previously, `relay` will refer to `relay`, as per the TVM conventions. Relay approaches neural networks by using functions as subgraphs, which purely have dataflow fragments that change the data, but not the topology of the graph. Changing the control flow is the responsibility of control flow fragments. It forms a `relay.Function`, presented as class diagram in Figure 4.2.2.1. This class contains every operation required for the network in its body, then wraps that into a `tvm.IRModule`, presented as class diagram in Figure 4.2.2.2.

```
Function
-------------------------------------------------
params : List[relay.Var]
body : relay.Expr
ret_type : Optional[relay.Type]
type_params : Optional[List[relay.TypeParam]]
attrs : Optional[DictAttrs]
span : Optional[relay.Span]
-------------------------------------------------
+ __init__(params: List[relay.Var],
           body: relay.Expr,
           ret_type: Optional[relay.Type],
           type_params: Optional[List[relay.TypeParam]],
           attrs: Optional[DictAttrs],
           span: Optional[relay.Span])
+ __call__(args: List[relay.Expr]) : Call
⋮
```

**Figure 4.2.2.1:** *relay.Function class diagram*

In the NNEFConverter I create the outputs of the graph, built up from `relay.Expr` calls, which becomes the `body` of the `Function`. The inputs of the network will be given as the `params` of the `Function`.

A necessary concept to know in TVM Relay is the `span`. In short, `relay.Span` refers to information that points back to the original source code, the source of the node. In practice, the NNEFConverter sets this attribute, such as for a specific node, the span information containing the node(s) that uses it as an input. This has to be done, but only TVM uses it, for debugging, and analysis purposes, the NNEFConverter does not utilize it further.

```
IRModule
---------------------------------------------
functions : Optional[dict]
type_definitions : Optional[dict]
attrs : Optional[DictAttrs]
global_infos : Optional[dict]
---------------------------------------------
+ from_expr(expr, functions, type_defs)
+ astext(show_meta_data: bool,
        annotate: Optional[Object->str) : str
⋮
```

**Figure 4.2.2.2:** *tvm.IRModule class diagram*

As discussed before, TVM uses `IRModule`s to wrap networks in. For the Relay frontend, I only had to get acquainted with the `from_expr()` function, as that is what I use in the NNEFConverter, to create the `IRModule`, passing the `Function` representing the NNEF graph into it. The `astext` function returns the textual representation of the graph contained in the `IRModule`.

For further information about `relay.Function` or `tvm.IRModule`, refer to the TVM GitHub repository [4], or the TVM documentation [12].

### 4.2.3 NNEFConverter class

For converting an NNEF graph into a TVM Relay, I used the, `NNEFConverter` class, approaching the problem with an Object Oriented method. Its class diagram is shown in Figure 4.2.3.1. The class contains the crucial functions to traverse the `nnef.Graph`, consume the vital parts, and transform it into the equivalent `tvm.Expr` calls. After every node has been converted, it analyses the result to create the `relay.Function` with the appropriate inputs, and function body. Then constructs the `IRModule` and returns it in a tuple with the parameter dictionary.

#### Attributes

This part will give a short description of the attributes of the NNEFConverter class.

`_freeze_vars` The `_freeze_vars` attribute contains the only parameter of the constructor, other than the graph itself, `freeze_vars`. This changes whether the converter should fold the NNEF variables into the resulting `IRModule` as constants.

- True: In this mode, the data from the `nnef.Tensor`s will be loaded in as constants `relay.const`, which can be folded into the graph for optimizations. The compile-time might increase as a result, but run-time should decrease.

```
NNEFConverter
-------------------------------------------------------------
_freeze_vars : bool
_nodes : dict
_consts : dict
_inputs : dict
_num_inputs : int
_params : dict
_num_params : int
-------------------------------------------------------------
# Constructor
+ __init__(freeze_vars: bool=False)

# The main function that is called
+ from_nnef(graph: nnef.Graph) : Tuple[tvm.IRModule, dict]

# nnef.Grpah traversal functions
- _parse_inputs(graph: nnef.Graph)
- _construct_nodes(graph: nnef.Graph)

# Single node converter handling functions
- _set_operator(node)
- _set_const(node)
- _set_variable(tensor)

# span handling functions
- _set_parameter_span(node, node_source_name)
- _set_par_span_helper(node, node_source_name, name, field_name)

# Helper functions
- _get_relay_op_call(name, inputs, attrs)
- _set_literal_inputs(node)
```

**Figure 4.2.3.1:** *NNEFConverter class diagram*

- False: But for use cases where modification of variables might be desired (e.g. training), having the variables in a separate object saves on compile time. The parameters, and variables will still be loaded into memory, but only converted into `relay.Var` objects, that are modifiable.

`_nodes` The `_nodes` attribute contains a dictionary, where the key is the NNEF identifier for the node, and the value is the equivalent TVM node, which can be `relay.const`, `relay.var`, or `tvm.realy.Expr`. This contains every node of the network, including inputs, outputs, constants, and parameters.

`_consts` The `_consts` attribute contains a dictionary with a structure similar to `_nodes`, but contains only the constant values in the `nnef.Graph`. This is used to reach the tensor data more easily.

`_inputs` The `_inputs` attribute contains the elements of `_nodes` which are the inputs of the `nnef.Graph`.

`_num_inputs` The `_num_inputs` attribute contains the number of inputs. This is provided as a convenience attribute.

`_params` The `_params` attribute contains the NNEF variables, if `freeze_vars` is `False`, in the same format as `_nodes`. Otherwise, it is empty.

`_num_params` The `_num_params` attribute contains the number of parameters. This is provided as a convenience attribute.

### Functions

This section gives an overview of the methods of the NNEFConverter class and provide the motivation for them. If complexity requires it, the subsequent section gives further information about the structure.

**__init__** The constructor of the class takes one parameter, the `freeze_vars` switch, which is saved to `self._freeze_vars`. Then it initializes the other attributes to either an empty `dict`, or 0.

**from_nnef** This is the main method that is used internally, in the enclosing frontend function, having the same name, `NNEFConverter.from_nnef()`. It calls the `nnef.Graph` traversal methods, converting its nodes into the attributes. After reading the graph, it analyzes the resulting `tvm.Exprs`, to build up the `relay.Function`, then the `tvm.IRModule` as well. Its return value is the resulting `tvm.IRModule`, `dict` tuple, the result of the conversion. A detailed explanation can be found in the following section.

**_parse_inputs** This method traverses the `nnef.Graph.inputs`, populating the `_inputs`, updating the `_num_inputs` attributes as well. The method also loads the input variables into the `_nodes` attribute, as it needs to contain every node.

**_construct_nodes** This method traverses the `nnef.Graph.operations`, converting every node in the network. It skips inputs, called external variables in NNEF, as they have been already handled in `_parse_inputs`. If it encounters a `variable` or `constant` operation, it calls the corresponding converter functions, `_set_variable` or `_set_const` respectively. For any other node, the conversion can be generalized. The NNEFConverter uses an operation library, `NNEFConverter.nnef_ops.py`, to properly convert the operation calls.

**_set_operator** This is the general operation converting method, that takes an `nnef.Operation` as its parameter, extracts its inputs, parameters, and attributes, and converts the operation into a `relay.Call`, from the operation library. The conversion has to be able to handle both literal numerical, or other expressions as inputs, for this, the `_set_literal_inputs` method converts the former into `relay.consts`. After the conversion is done, the method tries to fold the constants into the model, then saves the output(s) to `_nodes`.

**_set_const** This method handles the NNEF `constant` operations. The tensor data is extracted from the `nnef.Tensor` object, if it's a singular fill value, it creates an array with `numpy.full`, otherwise a `numpy.array` is called, and the created `numpy.ndarray` is saved in `_consts`, in a `relay.const` object. The node is also saved in `_nodes`.

**_set_variable** This method handles the NNEF `variable` operations. Depending on the `_freeze_vars` switch, the data can be saved either as a constant in `_consts` or saved as a variable parameter, and added to `_params`.

**_set_parameter_span** This method sets the `span` attribute of every TVM operation. It is used in `_construct_nodes` to set the operation's inputs' attribute. This method traverses every input of the node and calls the following function, `set_par_span_helper` on them.

**_set_par_span_helper** This method generates the proper span attribute, by getting the input node information from `_nodes`, and generating the span information with TVM's built-in functions. Then it updates the node in `_nodes`, and in `inputs` or `_consts` if applicable.

**_get_relay_op_call** This method is responsible for handling the lookup of the conversion function from the operation library. This thinly wraps the access to the dict of `operation name - conversion function call`, which contains the converter functions from the file `nnef_ops.py`, checking if the operation is known to NNEFConverter.

**_set_literal_inputs** This method is used in `_construct_nodes`, as NNEF defines using literal values in some functions (e.g. a literal `2.0` for a `mul` operation, literal `true` for a `select` operation), which TVM can not handle. For this, I have to convert them into `relay.consts`.

### Functions - detailed analysis

In this section, I will give detailed information, with the help of code snippets for some functions introduced above, where complexity requires it for full understanding.

**from_nnef** In this section, I walk through the source code, which can be seen in Figure 4.2.3.2. I explain step by step the output analysis.

```python
def from_nnef(self, graph: nnef.Graph) -> typing.Tuple[tvm.IRModule, dict]:
    self._parse_inputs(graph)
    self._construct_nodes(graph)

    outputs = [self._nodes[n] for n in graph.outputs]
    outputs = outputs[0] if len(outputs) == 1 else tvm_expr.Tuple(outputs)

    nodes = {v: k for k, v in self._nodes.items()}
    free_vars = analysis.free_vars(outputs)
    free_vars = [nodes[var] for var in free_vars]
    for i_name in self._params.keys():
        if i_name in free_vars and i_name not in self._inputs:
            self._inputs[i_name] = self._nodes[i_name]
    func = function.Function(list(self._inputs.values()), outputs)
    return IRModule.from_expr(func), self._params
```

**Figure 4.2.3.2:** *Source code of the self.from_nnef method*

As visible in the function signature, `from_nnef` takes a single argument other than `self`, an `nnef.Graph` to convert. In this section, `graph` will refer to this function parameter.

In lines 2-3 the `graph` is traversed, filling the attributes of the `NNEFConverter` class. In the following lines first the output is collected from `_nodes`. If there is a singular output, its type will be `relay.Call`, the output node itself, otherwise `relay.Tuple`, which contains `relay.Call`. Subsequently, I use the `relay.analysis` package to collect the free variables, the variables that are used in the workflow of the expected output, but are not defined in that sequence in Line 9. If there are free variables, they have to be contained in `_params`, and I had to add them as inputs of the `relay.Function`. By adding the correct node from `_nodes` to `_inputs`, I solved this issue. Then I can build the `relay.Function`, via providing the inputs as its first argument, then the previously defined `outputs`, as the function's body. Finally, I can construct the desired `IRModule` from this function, and also return with the `_params`.

**_set_operator** As mentioned before, general operations are converted with `_set_operator`. This handles all NNEF operations other than `external`, `variable`, and `constant`.

Its source code is shown in Figure 4.2.3.3.

As visible in the function signature, `_set_operation` takes a single argument other than `self`, an `nnef.Operation`, which it converts. In this section, `node` will refer to this function parameter.

First, the function has to convert every remaining non-TVM type object into a class, derived of `tvm.Expr`. In Line 2 the remaining literal inputs are converted into `relay.consts` via `_set_literal_inputs`. Then in Line 3 the aforementioned span setting takes place, setting the `node`'s inputs' span to the name of `node`. After that, I had to collect the converted values of the inputs of `node`, which happens through Lines 5-21. This needs to take all `list` type, previous node type, or literal types into account. In Line 23, by calling `_get_relay_op_call`, I can get the TVM equivalent of the operation `node`, and convert its inputs and attributes via the conversion function. More about the conversion operation library can be read in Section 4.3.

The rest of the function handles the future output of this `converted` object. To insert into `_nodes` correctly, I had to determine the number of expected outputs of `covnerted`. This derives from its type, as TVM handles tuples in a `relay.expr.TupleWrapper`. For the case that `converted` is an output node, I can call `set_span` on `converted` in Line 38. Then I had to add every output of `node` to `_nodes`, so I could refer to them independently.

```python
def _set_operator(self, node):
    self._set_literal_inputs(node)
    self._set_parameter_span(node, node.name)
    inputs = []
    for ink, inv in node.inputs.items():
        if isinstance(inv, list):
            for i, linv in enumerate(inv):
                if linv in self._nodes.keys():
                    inputs.append(self._nodes[linv])
                else:  # handle literal inputs
                    name = f"{node.name}_{ink}_{i}"
                    assert name in self._nodes, f"{name} has not been properly handled"
                    inputs.append(self._nodes[name])

        else:
            if inv in self._nodes.keys():
                inputs.append(self._nodes[inv])
            else:  # handle literal inputs
                name = f"{node.name}_{ink}"
                assert name in self._nodes, f"{name} has not been properly handled"
                inputs.append(self._nodes[name])

    converted = self._get_relay_op_call(node.name, inputs, node.attribs)

    if not isinstance(converted, tvm_expr.TupleWrapper):
        outputs_num = 1
    else:
        outputs_num = len(converted)

    if outputs_num == 1:
        if not isinstance(converted, tvm_expr.TupleWrapper):
            converted = fold_constant(converted)
        else:
            converted = fold_constant(converted.astuple())
    else:
        converted = tvm_expr.TupleWrapper(fold_constant(converted.astuple()), len(converted))

    converted = set_span(converted, node.name)

    if outputs_num == 1:
        # check if the singular ret val is a list of only one element
        ret_val = list(node.outputs.values())[0]
        if isinstance(ret_val, list):
            self._nodes[ret_val[0]] = converted
        else:
            self._nodes[ret_val] = converted
    else:
        for i, out in zip(range(outputs_num), node.outputs["values"]):
            self._nodes[out] = converted[i]
```

**Figure 4.2.3.3:** *Source code of the self._set_operation function*

### 4.2.4 Miscellaneous functions in `from_nnef.py`

In the `from_nnef.py` file, there are three additional functions next to the `NNEFConverter` class, `get_type`, `make_parameter_span`, and `from_nnef`.

**get_type** is a simple string matching function, which takes the NNEF data types (scalar, integer, logical, string), and returns the equivalent TVM data types (float32, int32, bool, string respectively).

**make_parameter_span** is a basic string generating function, in which, I format the inputs of a node into the TVM span attribute style string.

**from_nnef** is the main function of the package, the single public method, that handles the setup and call of the `NNEFConverter` class. The source code is presented in Figure 4.2.4.1.

```python
def from_nnef(
        model: typing.Union[str, os.PathLike, nnef.Graph],
        freeze_vars: bool = False,
) -> typing.Tuple[IRModule, dict]:
    conv_cls = NNEFConverter(freeze_vars)

    if not isinstance(model, nnef.Graph):
        model = nnef.load_graph(model)

    # fills in the nnef graph's shape information
    nnef.infer_shapes(model)

    return conv_cls.from_nnef(graph=model)
```

**Figure 4.2.4.1:** *Source code of the from_nnef function*

In this function, I check whether the given `model` parameter is an instance of `nnef.Graph` or path like (string, or os.PathLike), in which case, I load the model via `nnef.load_graph`. For conversion, I had to infer the shapes of the `nnef.Graph`, which is accomplished with the `nnef.infer_shapes` pass. This fills in the `shape` attributes of the `nnef.Operations` inside the model. Finally, I call the aforementioned `NNEFConverter` class' `from_nnef` with the model.

## 4.3 Operation conversion

In this section, I will list the converted functions I implemented in the frontend, along with the helper functions found in the `nnef_ops.py` file. For the most part, this only includes function signatures, as the conversion is trivial from the viewpoint of this frontend, but in the case of the more complex functions, more in-depth explanations will be included.

### 4.3.1 Helper functions

For most of the operation conversions, there exists a TVM equivalent to the NNEF operations. The parameterization differs for most higher complexity functions, but unary and binary functions usually are similar in structure.

To get the correct TVM functions, I used the TVM subpackage provided, `relay.frontend.common`. There exists a TVM function, `relay.frontend.common.get_relay_op`, which returns the corresponding TVM call to the function, whose string name has been given as the `get_relay_op`'s parameter. Almost all converter functions use this method, to get the correct TVM operator, but sometimes modification of the NNEF data is necessary.

As NNEF infers the dimension of operations from the data, it does not separate operations into separate dimensional ones, unlike TVM, where, for optimization's sake, they defined separate 1, 2, and 3-dimensional variants for some functions. For example for convolution, NNEF only has `conv`, TVM separates it into `conv_1d`, `conv_2d`, and `conv_3d`, not even detailing the additional possibilities, like implementations using for example the Winograd algorithm. To handle this, I implemented the `dimension_picker` function, which attaches the dimension suffix to the name of the operation, for use in the `relay.frontend.common.get_relay_op` function. It requires a parameter, `rank` to determine the correct number, adds error handling, and possible additional suffix extension (which is used for NNEF's `deconv`, whose equivalent is `conv_nd_transpose`, where n is the size of spatial dimensions, and `_transpose` is the suffix). The source code can be found in Figure 4.3.1.1.

```python
def dimension_picker(prefix, kernel_shape, suffix=""):
    rank = len(kernel_shape[2:])
    if rank == 1:
        return prefix + "1d" + suffix
    if rank == 2:
        return prefix + "2d" + suffix
    if rank == 3:
        return prefix + "3d" + suffix
    op_name = prefix + "1d/2d/3d"
    msg = f"Only 1D, 2D, and 3D kernels are supported for operator {op_name}."
    raise tvm.error.OpAttributeInvalid(msg)
```

**Figure 4.3.1.1:** *Source code of the dimension_picker function*

### Attribute shape conversion

Even if TVM supports an operation, that does not imply that it is trivially convertible. The most distinct difference is in approach to sliding-window-style operations, like convolution, and pooling; more importantly, their attributes - padding, stride, and pool size. Usually, as a consequence of NNEF's approach, NNEF attributes contain all dimensions, meaning the rank of the padding matrix for the operation is the same as the data tensor's. In contrast, TVM mostly uses attributes up to the active dimensions' rank, leaving out the batch and channel dimensions. For calculating these, I have created the operations `_size_conv`, `_stride_conv`, and `_padding_conv` methods.

#### _size_conv

I implemented the `_size_conv` function, which converts an NNEF pooling operation's pool size attribute, which indicates how large the sliding window should be in the operation.

The source code can be seen in Figure 4.3.1.2.

```python
def _size_conv(size, rank):
    if rank == 3:
        if len(size) == 1:
            return size
        if len(size) == 3:
            assert (
                size[0] == 1 and size[1] == 1
            ), "Incorrect window dimensions, first two dimensions must be 1"
            return size[2]
    if rank == 4:
        if len(size) == 2:
            return size
        if len(size) == 4:
            assert (
                size[0] == 1 and size[1] == 1
            ), "Incorrect window dimensions, first two dimensions must be 1"
            return size[2:]
    if rank == 5:
        if len(size) == 3:
            return size
        if len(size) == 5:
            assert (
                size[0] == 1 and size[1] == 1
            ), "Incorrect window dimensions, first two dimensions must be 1"
            return size[2:]

    raise ValueError(f"Unexpected window size, got {len(size)}")
```

**Figure 4.3.1.2:** *Source code of the _size_conv function*

Both average and max pooling uses this, as NNEF sets the rank of pool size to the rank of the input tensor, but TVM pooling only requires the active dimensions.

The function takes two parameters, `size` and `rank`. The `rank` parameter refers to the rank of the input tensor, the number of its non-zero dimensions. It cannot be inferred from the length of `size`, as both 1 and 3-dimensional data can have a pool size with a length of 3. The `size` is the pool size, that can either contain the batch and channel dimensions, if called from an NNEF operation or can only contain the active dimension sizes if a call was made from TVM nodes' output.

Lines 6, 14, and 22 provide some checks, ensuring that the NNEF batch and channel dimensions must be 1, to mean the same operation in TVM. Although pooling along those axes is recognized in the NNEF format, the standard does not require APIs to support that, as that operation is not used, and can be solved with permutations.

### _stride_conv

I implement the `_stride_conv` function, which converts an NNEF stride attribute to a valid TVM stride attribute. The source code can be seen in Figure 4.3.1.3.

```
1  def _stride_conv(stride, rank):
2      if rank == 3:
3          if len(stride) == 1:
4              return stride
5          if len(stride) == 3:
6              assert (
7                  stride[0] == 1 and stride[1] == 1
8              ), "Not supported stride dimensions, first two dimensions
                   must be 1"
9              return stride[2:]
10     if rank == 4:
11         if len(stride) == 2:
12             return stride
13         if len(stride) == 4:
14             assert (
15                 stride[0] == 1 and stride[1] == 1
16             ), "Not supported stride dimensions, first two dimensions
                   must be 1"
17             return stride[2:]
18     if rank == 5:
19         if len(stride) == 3:
20             return stride
21         if len(stride) == 5:
22             assert (
23                 stride[0] == 1 and stride[1] == 1
24             ), "Not supported stride dimensions, first two dimensions
                   must be 1"
25             return stride[2:]
26     raise ValueError(f"Unexpected stride in {rank - 2}D, got {len(stride
          )}: {stride}")
```

**Figure 4.3.1.3:** *Source code of the _stride_conv function*

Stride is the displacement of the kernel in sliding-window operations. In NNEF pooling operations the stride is the same rank as the tensor, while in convolutions it is only the spatial dimensions' rank. In TVM, it is only the rank of the spatial dimensions' size.

The function takes two parameters, `stride` and `rank`. The `rank` parameter refers to the rank of the input tensor. It cannot be inferred from the length of `stride`, as both 1 and 3-dimensional data can have stride with a length of 3. The `stride` parameter is the strides of the kernel, either containing the batch and channel dimensions for pooling operations or only consisting of strides for spatial dimensions, for convolution and box operations.

### _padding_conv

The `_padding_conv` function converts an NNEF padding attribute to a valid TVM padding attribute, as shown by the source code in Figure 4.3.1.4. As mentioned before, NNEF supports padding on all dimensions, while TVM does not, only supporting spatial or active dimensions. For this, the padding has to be truncated in the case of pooling operations, as they include padding on the batch and channel dimensions. In the case of padding, the structure of the attribute differs as well. NNEF uses a list of tuples, where a tuple contains the 'before' and 'after' padding for the corresponding dimension, for example in 2D, it has the structure:

$$[(up, down), (left, right)].$$

But in the case of TVM, the padding is in a single list, and the structure is the 'before' padding values first, for all dimensions, then the 'after' values, for example in 2D:

$$[up, left, down, right].$$

The function takes three parameters, two positional, `padding` and `rank`, and one keyword, `keepdims`. The `rank` parameter refers to the rank of the input tensor. It cannot be inferred from the length of `padding`, as both 1 and 3 dimensional data can have stride with a length of 3. `padding` contains the sizes of padding for each dimension, in a list of tuples, to convert to a singular list. The `keepdims` parameter is used in the case of some rare TVM operations, where the batch and channel dimensions are required (for example in `relay.sliding_window`, which generates a tensor, where it simulates a sliding window operation, without any reduction operations). In this frontend it is not used, but outside of the scope of this thesis, it is needed.

```python
def _padding_conv(padding, rank, keepdims=False):
    if isinstance(padding[0], (tuple, list)):
        # 1D
        if rank == 3:
            if len(padding) == 1:
                return padding[0]
            if len(padding) == 3:
                if not keepdims:
                    assert padding[0] == (0, 0) and padding[1] == (0, 0), (
                        "Incorrect padding. " "Padding on C,I dimensions not supported"
                    )
                    return padding[2]
                else:
                    return padding

        # 2D
        if rank == 4:
            if len(padding) == 2:
                return [x[i] for i in [0, 1] for x in padding]
            if len(padding) == 4:
                if not keepdims:
                    assert padding[0] == (0, 0) and padding[1] == (0, 0), (
                        "Incorrect padding. " "Padding on C,I dimensions not supported"
                    )
                    return list(itertools.chain.from_iterable(zip(padding[2], padding[3])))
                else:
                    return padding

        # 3D
        if rank == 5:
            # shortened, because of  triviality
            #   please refer to nnef_ops.py for the full code

        raise ValueError(
            f"Incorrect padding style for {rank - 2}D operand. Only length of {rank - 2}, {rank} "
            f"supported, got {len(padding)}: {padding}"
        )

    raise ValueError("nnef should not have singular padding")
```

**Figure 4.3.1.4:** *Source code of the _padding_conv function*

### Padding calculations

Both NNEF and TVM support padding as an optional attribute, meaning that in some cases it has to be calculated automatically by the program. In these cases, NNEF and TVM differ. The amount of padding is the same, the only problem arises when it is asymmetric, as the placement of the padding does not align between the two definitions. Solving this, I have created `_calculate_nnef_padding` and `_calculate_nnef_padding_deconv`. In these methods, I implement the NNEF style padding calculation and splitting, to be able to give them as exact values to TVM, so the automatic calculation is skipped there. The source code can be seen in Figure 4.3.1.5.

```
def _calculate_nnef_padding(active_shape, strides, kernel_shape, dilation):
    output = [(ui+(s-1))//s for ui, s in zip(active_shape, strides)]
    dilated = [(f-1)*d+1 for f, d in zip(kernel_shape, dilation)]
    total = [
        max(0, (di-1)*s+df-ui)
        for di, s, df, ui in zip(output, strides, dilated, active_shape)
    ]
    padding = [(pad//2, (pad+1)//2) for pad in total]
    return padding
```

**Figure 4.3.1.5:** *Source code of the _calculate_nnef_padding function*

The function takes the attributes necessary to calculate the required padding, namely `active_shape`, sizes of the active shapes in the operation, `strides`, the strides for the operation, `kernel_shape`, the shape of the kernel in the sliding-window operation, and `dilation`, for the dilation, meaning the amount with which the kernel entries are displaced while matching the kernel to the input in a single input position. The function returns the NNEF format for padding, so further conversion may be required in the calling function.

This function works exclusively in cases when there is no expected output shape, unlike in the case of deconvolution, debox. For those, I had to create a separate function, presented in Figure 4.3.1.6.

```
def _calculate_nnef_padding_deconv(data_sh, strides, kernel_active_sh, dilation, output_shape):
    out_sh = output_shape[2:] if output_shape else [ui*s for ui, s in zip(data_sh, strides)]
    dilated = [(f-1)*d+1 for f, d in zip(kernel_active_sh[2:], dilation)]
    total = [
        max(0, (di-1)*s+df-ui)
        for di, s, df, ui in zip(data_sh, strides, dilated, out_sh)
    ]
    return total, out_sh
```

**Figure 4.3.1.6:** *Source code of the _calculate_nnef_padding_deconv function*

As presented, the function takes an additional parameter, `output_shape`, which is the desired output shape in NNEF format. The result is a tuple, whose first element is the padding to apply, and the second is the TVM formatted output shape, which only contains the active dimensions' shape.

`_get_converter_map`

For looking up the correct functions, I have created the `_get_converter_map` function, which returns a dictionary, used as a lookup table. The keys of the dictionary are the NNEF string name of the operations, as they can be accessed from the `nnef.Operation.name` attribute. The values are the conversion functions, in Python function objects, so the external process can call the conversion function with the attributes. For brevity's sake, the source code is severely truncated in Figure 4.3.1.7, check the `nnef_ops.py` file for the complete implementation.

```python
def _get_converter_map():
    return {
        "copy": copy_converter,
        "neg": neg_converter,
        "rcp": rcp_converter,
        # etc.
    }
```

**Figure 4.3.1.7:** *Source code of the _get_converter_map function*

#### Future proofing

In preparation for the case, where NNEF expands and adds a new attribute, causing any of the conversion functions to receive an attribute that they do not expect, NNEFConverter raises a new Error via `__unexpected_attrs`.

Additionally, in the case where a currently not supported operation is called, another Exception is raised. For efficiency, not deployed operations can call the `ndop` function, that I created for this case, which raises the Exception in question.

These are for development purposes, a typical end user should not run into these repeatedly. Not implemented operations might be more common in older, less-used models, as some of those operations do not fit in the scope of this frontend.

### 4.3.2 Non-trivial conversions

In this section, I will describe the conversion strategy for the functions developed, whose equivalent is not trivially usable. Some of them are easily expressible with other operations (these are called compound operations), in those cases, the formula used can be seen in the NNEF Specification [11].

### rcp_converter

Reciprocal operation is nonexistent in TVM; therefore, for `rcp_converter`, I solved this issue, by using `div_converter` to divide a constant 1 numerator with the input tensor. The source code is presented in Figure 4.3.2.1.

```python
def rcp_converter(data, **kwargs):
    if kwargs:
        __unexpected_attrs("rcp", kwargs)

    if isinstance(data, relay.Call):
        d_type = infer_type(data).checked_type.dtype
    else:
        d_type = data.type_annotation.dtype

    return div_converter(tvm_expr.const(1, dtype=d_type), data)
```

**Figure 4.3.2.1:** *Source code of the rcp_converter function*

### sqr_converter

TVM does not provide a convenience operator for squaring a tensor, so I used the `power` operation to raise the input operator to the power of 2. The source code is presented in Figure 4.3.2.2.

```python
def sqr_converter(data, **kwargs):
    if kwargs:
        __unexpected_attrs("sqr", kwargs)

    if isinstance(data, relay.Call):
        d_type = infer_type(data).checked_type.dtype
    else:
        d_type = data.type_annotation.dtype

    return get_relay_op("power")(data, tvm_expr.const(2, dtype=d_type))
```

**Figure 4.3.2.2:** *Source code of the sqr_converter function*

### rsqr_converter

TVM does not provide a convenience operator for reverse squaring a tensor, so I used the `power` operation to raise the input operator to the power of -2. The source code is presented in Figure 4.3.2.3.

```
def rsqr_converter(data, **kwargs):
    if kwargs:
        __unexpected_attrs("rsqr", kwargs)

    if isinstance(data, relay.Call):
        d_type = infer_type(data).checked_type.dtype
    else:
        d_type = data.type_annotation.dtype

    return get_relay_op("power")(data, tvm_expr.const(-2, dtype=d_type))
```

**Figure 4.3.2.3:** *Source code of the rsqr_converter function*

### clamp_converter

NNEF defines clamping with either single, global limit values or tensors of the same shape as input, to create an element-wise clamp over the input. TVM only provides the function for the first case, so for the second, it had to be expressed by consecutive max and min over the data. The source code is presented in Figure 4.3.2.4.

```
def clamp_converter(x, a, b, **kwargs):
    if kwargs:
        __unexpected_attrs("clamp", kwargs)

    # only works if b and a are Constant floats, not tensors
    if isinstance(a, tvm_expr.Constant) and isinstance(b, tvm_expr.
   Constant):
        return get_relay_op("clip")(x, float(a.data.numpy()), float(b.
           data.numpy()))

    return max_converter(min_converter(x, b), a)
```

**Figure 4.3.2.4:** *Source code of the clamp_converter function*

### conv_converter

As briefly mentioned, NNEF provides one operation for any dimensional convolution, and also includes bias in that, while TVM has separate functions for it. The source code is presented in Figure 4.3.2.5.

```python
def conv_converter(data, kernel, bias, border, stride, padding, dilation, groups, **kwargs):
    if kwargs:
        __unexpected_attrs("conv", kwargs)

    if border != "constant":
        print(f"Currently {border} border is not supported, used `constant` border")

    kernel_shape = infer_shape(kernel)
    dshape = infer_shape(data)

    strides = _stride_conv(stride, len(kernel_shape)) if stride else (1,) * (len(kernel_shape) - 2)

    dilation = dilation if dilation else ((1,) * (len(kernel_shape)-2))

    if not padding:
        padding = _calculate_nnef_padding(dshape[2:], strides, kernel_shape[2:], dilation)

    pad = _padding_conv(padding, len(kernel_shape))

    channels = kernel_shape[0]

    if groups == 0:
        groups = channels

    op = get_relay_op(dimension_picker("conv", kernel_shape))
    conv_out = op(
        data=data,
        weight=kernel,
        strides=strides,
        padding=pad,
        dilation=dilation,
        groups=groups,
        channels=channels,
        kernel_size=kernel_shape[2:],
    )

    res = None
    if isinstance(bias, tvm_expr.Constant):
        if (bias.data.numpy() == 0).all():
            res = conv_out

    if not res:
        res = tvm_op.nn.bias_add(conv_out, relay.squeeze(bias, axis=0))

    return res
```

**Figure 4.3.2.5:** *Source code of the conv_converter function*

In the conversion, I print a warning if the border mode is not constant, as TVM does not support any other border mode in this operation. The converter has to either convert the attributes with the previously introduced operations or create default values for them.

Then the number of channels and groups is inferred. In NNEF, setting the attribute for the number of groups to 0 is a shorthand for saying to use the same number of groups as there are channels. Afterwards, the proper TVM method is collected and called with the proper attributes to generate the output, a `relay.Call`.

If NNEF adds bias to the convolution, then the function has to add it separately, with a `bias_add` function. Squeezing the NNEF bias is necessary, as TVM takes bias of size `[bias_dim]`, while NNEF provides `[1,bias_dim]`.

**deconv_converter**

As briefly mentioned, NNEF provides one operation for any dimensional deconvolution, and also includes bias in that, while TVM has separate functions for it. The source code is presented in Figure 4.3.2.6.

In the conversion, I print a warning if the border mode is not constant, as TVM does not support any other border mode in this operation.

The converter has to either convert the attributes with the previously introduced operations or create default values for them. It has to use the NNEF calculation method to get the correct padding and output shape for TVM.

Then the number of channels and groups is inferred. In NNEF, setting the attribute for the number of groups to 0 is a shorthand for using the same number of groups as the number of batches. But in deconv, the number of channels has to be multiplied by the number of groups to get the desired TVM channel number.

Then the proper TVM method is collected and called with the proper attributes to generate the output, a `relay.Call`.

If NNEF adds bias to the convolution, then the function has to add it separately, with a `bias_add` function. Squeezing the NNEF bias is necessary, as TVM takes bias of size `[bias_dim]`, while NNEF provides `[1,bias_dim]`.

```python
def deconv_converter(
    data, kernel, bias, border, stride, padding, dilation, output_shape,
        groups, **kwargs
):
    # kwargs check ommitted for brevity

    kernel_shape = infer_shape(kernel)
    rank = len(kernel_shape)
    strides = _stride_conv(stride, rank) if stride else (1,) * (rank -
        2)
    dilation = dilation if dilation else ((1,) * (rank - 2))

    total, out_sh = _calculate_nnef_padding_deconv(
        infer_shape(data), strides, kernel_shape, dilation, output_shape
    )

    if padding:
        pad = _padding_conv(padding, rank)
    else:
        pad = _padding_conv([(pad // 2, (pad + 1) // 2) for pad in total
            ], rank)

    if groups == 0:
        groups = kernel_shape[0]
    channels = kernel_shape[1] * groups

    out_pad = (
        [(x - (y - t)) % s for x, y, t, s in zip(output_shape[2:],
            out_sh, total, stride)]
        if output_shape else \
        (0, 0)
    )

    op = get_relay_op(dimension_picker("conv", kernel_shape, suffix="
        _transpose"))
    deconv_out = op(
        data=data,
        weight=kernel,
        strides=strides,
        padding=pad,
        dilation=dilation,
        groups=groups,
        channels=channels,
        kernel_size=kernel_shape[2:],
        output_padding=out_pad )

    res = None
    if isinstance(bias, tvm_expr.Constant):
        if bias.data.numpy() == np.array([0.0]):
            res = deconv_out

    if not res:
        res = tvm_op.nn.bias_add(deconv_out, relay.squeeze(bias,axis=0))

    return res
```

**Figure 4.3.2.6:** *Source code of the deconv_converter function*

### box_converter

The box operation performs summation over a local window. TVM has no equivalent to this operation, but box can be interpreted as a convolution with a constant 1 kernel, in the shape of the desired window. The source code is presented in Figure 4.3.2.7.

For a variation, where the attribute `normalize` is `True`, the local window's values have to be divided by the volume of the kernel. This can be substituted with a convolution with a kernel of the desired window, with the value 1 / *volume of window*.

```python
def box_converter(data, size, border, padding, stride, dilation, normalize, **kwargs):
    if kwargs:
        __unexpected_attrs("box", kwargs)

    dshape = infer_shape(data)

    if isinstance(data, relay.Call):
        d_type = infer_type(data).checked_type.dtype
    else:
        d_type = data.type_annotation.dtype

    size[0] = dshape[1]
    if normalize:
        kernel = relay.full(tvm_op.const(1/math.prod(size[2:]), d_type), size, d_type)
    else:
        kernel = relay.ones(size, d_type)

    out = conv_converter(
        data, kernel, tvm_expr.const(0, dtype=d_type), border, stride, padding, dilation, dshape[1]
    )
    return out
```

**Figure 4.3.2.7:** *Source code of the box_converter function*

I generated the proper filter for the aforementioned convolution in the converter function. I used the array generation functions, `relay.full` and `relay.ones` for that, with the correct dtype, and kernel size. Then I was able to call the convolution with `conv_converter`, and the output returned by it is the desired operation.

### debox_converter

The debox operation performs the inverse of box. TVM has no equivalent to this operation either, but debox can be interpreted as a deconvolution with a constant 1 kernel, in the shape of the desired window. The source code is presented in Figure 4.3.2.8.

For a variation, where the attribute `normalize` is `True`, the local window's values have to be divided by the volume of the kernel. This can be substituted with a convolution with a kernel of the desired window, with the value 1 / *volume of window*.

```
def debox_converter(
    data, size, border, padding, stride, dilation, normalize, output_shape, **kwargs
):
    if kwargs:
        __unexpected_attrs("debox", kwargs)

    dshape = infer_shape(data)

    if isinstance(data, relay.Call):
        d_type = infer_type(data).checked_type.dtype
    else:
        d_type = data.type_annotation.dtype

    size[0] = dshape[1]
    if normalize:
        kernel = relay.full(tvm_op.const(1 / math.prod(size[2:]), d_type), size, d_type)
    else:
        kernel = relay.ones(size, d_type)

    out = deconv_converter(
        data,
        kernel,
        tvm_expr.const(0, dtype=d_type),
        border,
        stride,
        padding,
        dilation,
        output_shape,
        groups=dshape[1],
    )
    return out
```

**Figure 4.3.2.8:** *Source code of the debox_ converter function*

The converter function has to generate the proper filter for the aforementioned convolution. It uses the array generation functions, `relay.full` and `relay.ones` for that, with the correct dtype, and kernel size. Subsequently, the deconvolution can be called with `deconv_converter`, and the output returned by it will be the desired operation.

### nearest_downsample_converter

NNEF has an operator for nearest neighbor down-sampling, which, because of the lack of an equivalent in TVM, has to be interpreted as a box operation. The source code is presented in Figure 4.3.2.9.

```python
def nearest_downsample_converter(data, factor, **kwargs):
    if kwargs:
        __unexpected_attrs("nearest_downsample", kwargs)

    dims = 2 + len(factor)

    return box_converter(
        data,
        size=[1] * dims,
        border="constant",
        padding=[(0, 0)] * dims,
        stride=[1, 1] + factor,
        dilation=(1,) * (dims - 2),
        normalize=False,
    )
```

**Figure 4.3.2.9:** *Source code of the nearest_downsample_converter function*

The `factor` parameter is a list of the sampling factors. The corresponding kernel for box is where the `size` parameter has the same rank as the input tensor and consists of ones, and the striding is the same in the active dimensions as the factor.

### area_downsample_converter

NNEF has an operator for area interpolation down-sampling, which, because of the lack of an equivalent in TVM, has to be interpreted as a box operation. The source code is presented in Figure 4.3.2.10.

```python
def area_downsample_converter(data, factor, **kwargs):
    if kwargs:
        __unexpected_attrs("area_downsample", kwargs)

    dims = 2 + len(factor)

    return box_converter(
        data,
        size=[1, 1] + factor,
        border="constant",
        padding=[(0, 0)] * dims,
        stride=[1, 1] + factor,
        dilation=(1,) * (dims - 2),
        normalize=True,
    )
```

**Figure 4.3.2.10:** *Source code of the area_downsample_converter function*

The `factor` parameter is a list of the sampling factors. The corresponding kernel for box is where the `size` parameter has the same rank as the input tensor, the active dimensions being the same as the factor, and the striding is the same in the active dimensions as the factor.

### nearest_upsample_converter

NNEF provides an operator for nearest neighbor up-sampling, which, because of the lack of an equivalent in TVM, has to be interpreted differently. I used a resize operation, which provides the necessary parameters, to create an equivalent operation. The source code is presented in Figure 4.3.2.11.

```
def nearest_upsample_converter(data, factor, **kwargs):
    if kwargs:
        __unexpected_attrs("nearest_upsample", kwargs)

    dshape = infer_shape(data)
    new_size = [d * f for d, f in zip(dshape[2:], factor)]
    return get_relay_op(dimension_picker("resize", dshape))(
        data,
        new_size,
        method="nearest_neighbor",
        rounding_method="round",
    )
```

**Figure 4.3.2.11:** *Source code of the nearest_upsample_converter function*

For using the `relay.image.resize_nd` function, the desired size has to be calculated beforehand, by multiplying the current size of the spatial dimensions with the corresponding factor. Then the operation can be called with the proper parameters, setting the `method` to *nearest_neighbor*, and `rounding_method` to *round*.

### multilinear_upsample_converter

NNEF provides an operator for multi-linear interpolation-based up-sampling, which, because of the lack of an equivalent in TVM, has to be interpreted differently. Converting this is very lengthy, as the different methods, and borders require different approaches. The function contains 4 return statements, as shown in Figure 4.3.2.12. I am going to present them in two parts, the simpler `aligned` method, along with `symmetric` with `replicate` border is one group, that can be solved with `relay.image.resize_nd`, while the rest has to be solved with deconvolution, which will be grouped as well.

```
def multilinear_upsample_converter(data, factor, method, border, **
    kwargs):
    if kwargs:
        __unexpected_attrs("linear_upsample", kwargs)

    dshape = infer_shape(data)
    new_size = [d * f for d, f in zip(dshape[2:], factor)]
    if method == "aligned":
        # conversion from nn.upsampling to image.resizexd, re: discuss
            :11650
        return get_relay_op(dimension_picker("resize", dshape))(
            data,
            new_size,
            method="linear",
            coordinate_transformation_mode="align_corners",
        )
    if method == "symmetric" and border == "replicate":
        return get_relay_op(dimension_picker("resize", dshape))(
            data,
            new_size,
            method="linear",
            coordinate_transformation_mode="half_pixel",
        )
```

**Figure 4.3.2.12:** *Source code of the multilinear_upsample_converter function - first part*

As mentioned, these two cases can be converted into a resize operation, with proper parameters. The method is similar to `nearest_upsample_converter`, in Figure 4.3.2.11.

For the other part, I will introduce two functions that will be used, presented in Figure 4.3.2.13.

```
def _upsample_weights_1d(fact, symm):
    if symm:
        _weights = [1 - (i + 0.5) / fact for i in range(fact)]
        _weights = list(reversed(_weights)) + _weights
    else:
        _weights = [1 - abs(i) / float(fact) for i in range(-fact + 1,
            fact)]
    return np.array(_weights)

def _upsample_weights_nd(fact, symm):
    _weights = [_upsample_weights_1d(f, symm) for f in fact]
    return reduce(np.multiply, np.ix_(*_weights))
```

**Figure 4.3.2.13:** *Source code of the multilinear_upsample_converter helper functions*

These functions help by generating the weights of the kernels for the deconvolution to be used for upsampling.

```python
    n, c = dshape[:2]
    symmetric = method == "symmetric"
    weights = _upsample_weights_nd(factor, symmetric)
    weights = np.reshape(weights, newshape=(1, 1) + weights.shape)
    kernel = tile_converter(tvm_expr.const(weights), (c, 1) + (1,) * len(factor))

    output_shape = [n, c] + [f * s for f, s in zip(factor, dshape[2:])]

    if symmetric:
        return deconv_converter(
            data,
            kernel,
            tvm_expr.const(0.0),
            border="constant",
            stride=factor,
            padding=[(f - 1, f - 1) for f in factor],
            dilation=[],
            groups=c,
            output_shape=output_shape,
        )
    else:
        replicate = border == "replicate"
        if replicate:
            data = pad_converter(
                data, [(0, 0), (0, 0)] + [(1, 0)] * len(factor), border, tvm_expr.const(0.0)
            )
            padding = factor
        else:
            padding = [f // 2 for f in factor]

        return deconv_converter(
            data,
            kernel,
            tvm_expr.const(0.0),
            border="constant",
            stride=factor,
            padding=[(p, p - 1) for p in padding],
            dilation=[],
            groups=c,
            output_shape=output_shape,
        )
```

**Figure 4.3.2.14:** *Source code of the multilinear_upsample_converter function - second part*

These cases require the use of deconvolution, as shown in Figure 4.3.2.14. I could create the necessary kernel for it, with the help of the methods introduced in Figure 4.3.2.13. Then in the case of `symmetric` method, the use of deconvolution with the original input data, and created kernel is possible. Otherwise, the padding has to be adjusted to handle the different border modes.

### sum_reduce_converter

For sum reduction, NNEF provides a parameter, `normalize`, which TVM does not, and I had to handle it separately. The source code is presented in Figure 4.3.2.15.

```
def sum_reduce_converter(data, axes, normalize, keepdims=True, **kwargs):
    if kwargs:
        __unexpected_attrs("sum_reduce", kwargs)

    out = get_relay_op("sum")(data, axes, keepdims=keepdims)
    if normalize:
        return l2_normalization_converter(out, 0, [x - 2 for x in axes], 0.0)
    return out
```

**Figure 4.3.2.15:** *Source code of the sum_reduce_converter function*

If normalize is set to `True`, then the result will be equal to the result of a regular sum reduce operation, followed by an L2 normalization, with an epsilon of 0.0, as that is equivalent to the normalization method used by NNEF. Otherwise, the TVM `sum` operation is sufficient.

### reshape_converter

With the reshaping operator, the different approach to dimensions shows up again, where NNEF defines reshaping such that it requires a start index, from which it should modify the shapes, the shape to modify to, and the number of dimensions that should be affected. In contrast, TVM requires the resulting shape only, containing every dimension, even the unmodified ones.

```
def reshape_converter(data, shape, axis_start, axis_count, **kwargs):
    if kwargs:
        __unexpected_attrs("reshape", kwargs)

    dshape = list(infer_shape(data))
    if axis_count == -1:
        newshape = dshape[:axis_start] + shape
    else:
        newshape = dshape
        newshape[axis_start : axis_start + axis_count] = shape

    return get_relay_op("reshape")(data, newshape)
```

**Figure 4.3.2.16:** *Source code of the reshape_converter function*

Because of the difference, and NNEF's additional shorthand, the resulting shape calculation follows the formula seen in Figure 4.3.2.16. There is some overlap between NNEF and TVM's special values. Therefore, in the converter, I do not have to handle values like

0 separately, as they have the same meaning in both specifications. To read more about NNEF's special values, refer to the specification at the Khronos Registry [11], and about TVM's special values to its documentation [12].

### unsqueeze_converter

In the case of unsqueeze operation, NNEF can handle multiple axes with one operation, while TVM requires consecutive dimension expansions. The source code is presented in Figure 4.3.2.17.

```
def unsqueeze_converter(data, axes, **kwargs):
    if kwargs:
        __unexpected_attrs("unsqueeze", kwargs)

    axes = sorted(axes)
    for axis in axes:
        if axis < 0 and isinstance(data, tvm_expr.Var):
            axis = len(data.type_annotation.concrete_shape) + len(axes)
                + axis

        data = tvm_op.expand_dims(data, axis=axis, num_newaxis=1)
    return data
```

**Figure 4.3.2.17:** *Source code of the unsqueeze_converter function*

It is necessary to sort the axes, as improper dimension expansions may cause interference with each other. In NNEF, negative axis is also possible, which means that the indexing starts from the back of the tensor. To handle that, additional checks are necessary.

### split_converter

For splitting NNEF and TVM's approaches differ again, they both require an axis to split, but NNEF defines a list of ratios to split into, while TVM requires the indices to split on instead. The source code is presented in Figure 4.3.2.18.

For this conversion, I had to change a ratio list into an index list. Then TVM's `split` operation can handle the splitting of the tensor.

```python
def split_converter(data, axis, ratios, **kwargs):
    if kwargs:
        __unexpected_attrs("split", kwargs)

    axis_len = infer_shape(data)[axis]
    rat_mul = axis_len / sum(ratios)
    ratio_list = [(r * rat_mul) for r in ratios]

    s = 0
    indices = []
    for rat in ratio_list[:-1]:
        s += rat
        indices.append(int(s))

    return get_relay_op("split")(data, indices, axis)
```

**Figure 4.3.2.18:** *Source code of the split_ converter function*

### matmul_converter

Matrix multiplication is another operation, where NNEF's all-inclusive strategy makes conversion more difficult. TVM supports either 2D matrix multiplication with the `matmul` function, or `batch_matmul`, as a workaround for any other dimension, which I will present in Figure 4.3.2.19

In this operator, I had to implement a workaround because of TVM's linting checks. TVM uses Python black, that flags parameters with camel case as errors, but NNEF attributes can use it. Because of that, I had to remove `transposeA` and `transposeB` from the function's parameter list, and manually pop from the `kwargs` dictionary. More about testing can be read in Section 5.

The matmul function takes 4 arguments, the 2 matrices to multiply, and a boolean parameter for each, whether they should be transposed beforehand.

If both matrices are 2-dimensional, the normal TVM `matmul` operation suffices, which also has the transpose parameters. In any other case, I had to use batch matrix multiplication, broadcasting the matrices to proper sizes, while checking for broadcast compatibility, then using the operation on the matrices, with the proper transpose parameters. The result has to be reshaped as well, so the shape does not differ because of the batch repetition.

```python
def matmul_converter(a, b, **kwargs):
    transpose_a = kwargs.pop("transposeA")
    transpose_b = kwargs.pop("transposeB")
    if kwargs:
        __unexpected_attrs("matmul", kwargs)

    a_shape = infer_shape(a)
    b_shape = infer_shape(b)
    a_rank = len(a_shape)
    b_rank = len(b_shape)

    if a_rank == 2 and b_rank == 2:
        out = get_relay_op("matmul")(a, b, transpose_a=transpose_a,
            transpose_b=transpose_b)
    else:
        batch_shape = [1] * (max(a_rank, b_rank) - 2)

        for i, j in enumerate(reversed(a_shape[:-2])):
            batch_shape[i] = j

        for i, j in enumerate(reversed(b_shape[:-2])):
            if batch_shape[i] == 1 or j == 1 or batch_shape[i] == j:
                batch_shape[i] = max(batch_shape[i], j)
            else:
                msg = "Batch dimensions are not broadcastable."
                raise AssertionError(msg)

        batch_shape = batch_shape[::-1]

        a = tvm_op.broadcast_to(a, batch_shape + list(a_shape[-2:]))
        b = tvm_op.broadcast_to(b, batch_shape + list(b_shape[-2:]))

        out = get_relay_op("batch_matmul")(
            tvm_op.reshape(a, [-1, *a_shape[-2:]]),
            tvm_op.reshape(b, [-1, *b_shape[-2:]]),
            transpose_b=transpose_b,
            transpose_a=transpose_a,
        )

        out_shape = batch_shape + [a_shape[-2]] + [b_shape[-1]]
        out = tvm_op.reshape(out, out_shape)

    return out
```

**Figure 4.3.2.19:** *Source code of the matmul_converter function*

### avg/max_pool_converter

Pooling operations are very similar in structure, therefore I will discuss them together, they only contrast in how they interact with border styles. The source code presented in Figure 4.3.2.20 is max pooling, but every diverging part will be marked as such.

As shown, both pooling operations follow a similar ceonversion structure, using the shape conversion methods, `_size_conv`, `_stride_conv`, and padding conversion functions, `_padding_conv` and `_calculate_nnef_padding`.

```python
def max_pool_converter(data, size, border, padding, stride, dilation, **kwargs):
    if kwargs:
        __unexpected_attrs("max_pool", kwargs)

    # ignore border is supported in avg pool, but maxpool returns acceptable results for it as well
    if border not in ["constant", "ignore"]:
        print(f"Currently {border} border is not supported, used `constant` border")

    dshape = infer_shape(data)
    rank = len(dshape)

    pool_size = _size_conv(size, rank)
    strides = _stride_conv(stride, rank) if stride else (1,) * (rank - 2)

    dilation = dilation if dilation else ((1,) * (rank - 2))

    if not padding:
        padding = _calculate_nnef_padding(dshape[2:], strides, pool_size, dilation)

    pad = _padding_conv(padding, rank)

    # the following conditional is only necessary in max pool
    if border == "constant":
        padding = [(0, 0), (0, 0)] + padding
        data = pad_converter(data, padding, border, tvm_expr.const(0.0))
        pad = (0, 0)

    op = get_relay_op(dimension_picker("max_pool", dshape))
    return op(
        data,
        pool_size=pool_size,
        strides=strides,
        dilation=dilation,
        padding=pad,
        # the following parameter is only contained in avg pool
        count_include_pad=border != "ignore",
    )
```

**Figure 4.3.2.20:** *Source code of the max_pool_converter function, with parts of the avg_pool_converter function*

The differences arise with border styles, as in the case of maxpooling, the `constant` border style requires a manually added padding to act as border, as TVM would use a `reflect` border style by default.

In the case of avgpool, the operation supports an additional parameter, `count_include_pad`, which decides whether the edge values should be considered. Not using this corresponds basically to the `ignore` border style in NNEF.

### Simplifiable operations

Numerous operations in NNEF and convenience functions, or, by the nature of arithmetic, are expressible by chaining multiple, lower-level operations together. These are called compound operations. I tried to steer clear of these simplifications, as they could be less optimized than their first-order counterparts, but in some cases, it was unavoidable. In those cases, I followed the NNEF specification's formula to express the operation.

Examples of this, but not limited to, are:

- Activation functions:
  - `elu_converter`
  - `selu_converter`
  - `gelu_converter`
  - `silu_converter`
  - `softplus_converter`
- Linear functions:
  - `linear_converter`
  - `separable_conv_converter`
  - `separable_deconv_converter`
- `rms_pool_converter`
- Normalization functions:
  - `local_mean_normalization_converter`
  - `local_variance_normalization_converter`
  - `local_contrast_normalization_converter`
  - `l1_normalization_converter`

For their equation, refer to the NNEF Specification [11].

# 5 Testing

In this chapter, I will detail the tests performed on NNEFConverter, giving an explanation of the test files containing the unit tests. Furthermore, as this was part of an open-source project, it will also present additional integration testing.

## 5.1 Unit tests

This section will give an overview of the unit tests implemented for NNEFConverter. Currently, there are 301 test cases, which cover operation conversions. Some cases also include multiple operation calls, testing the overall graph generation of the API.

The Project provides the unit tests to both the Standalone and the Integration versions:

- For standalone the tests can be found in `{project}/Tests/`.
- For the integration, in `{project}/Integration/tests/python/frontend/nnef/`.

The folders contain an additional directory, `cases`, which contains a graph for each test case, and a file `test_standalone.py` for the standalone tests, or `test_forward.py` for use in TVM. The two script files' structure closely resembles each other, the difference being that TVM provides additional pytest fixtures, `target` and `dev`.

The `target` parameter contains the desired target to run the model on, this can be `llvm` for CPU or `cuda` for GPU for example. The `device` is the actual device, the program should use to translate the network locally, `tvm.cpu` or `tvm.cuda` respectively, for the examples before. Going in-depth with TVM runtimes is outside of the scope of this thesis, so for the full explanation of TVM testing, refer to the TVM docs [12].

If the tests are run in TVM, through these fixtures, TVM automatically runs the test on multiple targets, but this needs a TVM source installation to run locally. Because of this, in the Standalone tests, I had to remove these fixtures, and use a default value of `llvm` as target, and `tvm.cpu(0)` as device, making the tests run only on CPU, namely on the first recognized by the system.

I based the unit tests on a test case generating script. I provide operation names and data types to this script, and it generates the NNEF model, covering a test case. The tests using these cases follow the naming scheme: `test_cts_{test_case}`. There are 234 cases under cts. I had to modify the generated cases, to occupy less space, and to align more to test the conversion itself, rather than the operation. As those tests did not cover every case I felt necessary, I added 67 additional cases, labeled as `test_ats_{test_case}`.

### 5.1.1 The verify_model function

The `verify_model` function is responsible for executing the tests and comparing them.

The function first loads in the model from memory but omits the weights of the files, so the test cases could be packaged without large binary files. Then it generates values for the inputs and weights of the model, taking into account the given operation's valid values. The first part of the source code is presented in Figure 5.1.1.1.

Then I executed the model first on the NNEF reference Interpreter, which as mentioned before, uses PyTorch, wrapped to work with NNEF. This gives me the baseline, to compare against. This has been moved to a separate helper function, `get_nnef_outputs`, presented in Figure 5.1.1.2.

```
def get_nnef_outputs(path, inputs):
    ip = interpreter.Interpreter(path, None, None)
    inputs = [inputs[tensor.name] for tensor in ip.input_details()]
    return ip(inputs)
```

**Figure 5.1.1.2:** *Source code of the get_nnef_outputs function*

Afterward, I converted the NNEF graph into a Relay model, executed it with `relay.executor`, and compared the values with TVM's comparison function, `tvm.testing.assert_allclose`, as shown in Figure 5.1.1.3. The function works similarly to numpy's `testing.assert_allclose` function, checking shape, and element-wise equality, with an absolute and relative tolerance.

```
def verify_model(
        model_path,
        target='llvm',
        device=tvm.cpu(0),
        rtol=1e-5,
        atol=1e-5,
):
    path = os.path.join(graphs_dir, model_path)
    graph = nnef.load_graph(path, load_variables=False)
    nnef.infer_shapes(graph)
    inputs = {}

    for inp in graph.inputs:
        intensor = graph.tensors[inp]
        shape = intensor.shape
        if any(exc in model_path for exc in \
                ["log", "sqrt", "pow", "batch_norm"]):
            low = 0.0
        else:
            low = -1.0
        high = 1.0
        if "acosh" in model_path:
            high = 2.0
            low = 1.0
        if intensor.dtype == "scalar":
            inputs[inp] = np.random.uniform(low=low, high=high,
            size=shape).astype("float32")
        elif intensor.dtype == "integer":
            inputs[inp] = np.random.randint(0, 64, shape)
        elif intensor.dtype == "logical":
            inputs[inp] = np.random.binomial(1, 0.5, shape)
            .astype("bool")
        elif intensor.dtype == "string":
            inputs[inp] = np.random.uniform(low=low, high=high,
            size=shape).astype("string")


    for operation in graph.operations:
        if operation.name == "variable":
            tensor_name = operation.outputs["output"]

            shape = operation.attribs["shape"]

            assert operation.dtype == 'scalar', \
                f'variable of type {operation.dtype} is not supported,
                   please update verify_model'

            data = np.random.uniform(low=-1.0,
                size=shape).astype("float32")

            tensor = graph.tensors[tensor_name]
            graph.tensors[tensor_name] = _nnef.Tensor(
                tensor.name, tensor.dtype, shape, data,
                tensor.quantization)
```

**Figure 5.1.1.1:** *Source code of the verify_model function - first part*

```
    outputs = get_nnef_outputs(graph, inputs)

    mod, params = NNEFConverter.from_nnef(graph)

    with tvm.transform.PassContext(opt_level=3):
        # dev = tvm.device(target, 0)
        executor = relay.create_executor(
            "graph", mod, device=device, target=target, params=params
        ).evaluate()
        out = executor(**inputs)

        if not isinstance(out, (list, tuple)):
            out = [out]

        for i, base_out in enumerate(outputs):
            tvm.testing.assert_allclose(out[i].numpy(), outputs[base_out
               ], rtol=rtol, atol=atol)
```

**Figure 5.1.1.3:** *Source code of the verify_model function - second part*

### 5.1.2 Executing the tests

Executing the tests requires pytest to be installed in the local Python environment.

#### Standalone

They can be executed from the `{project}` directory, via the command presented in Figure 5.1.2.1.

```
python -m pytest Tests/
```

**Figure 5.1.2.1:** *Running the standalone tests*

Note that If run from the `Tests` directory, the tests will fail, as it looks for the test cases in the relative path, `Tests/cases/`.

#### Integration tests

Running the unit tests in TVM locally has to be done from the `${TVM_HOME}` directory, and can be done by running the command seen in Figure 5.1.2.2.

```
TVM_FFI=ctypes python3 -m pytest -v tests/python/frontend/nnef/
```

**Figure 5.1.2.2:** *Running NNEF unit tests in TVM*

## 5.2 Integration tests

TVM has additional integration tests, above unit testing, for more detailed information check the testing section in the TVM docs [12]. Most notably lint checks, which every pull request must comply with. These can be run either locally, in a Docker image, or with every PR a Jenkins workflow is triggered.

Because of the Docker images, I had to modify the image building scripts to include NNEF installation, and include them in the images that run unit tests. These modifications can be seen in `Integration/docker/`.

Additionally, I had to modify the test tasks themselves:

- The lint process, to whitelist `.nnef` files, so neither a script nor Apache RAT flags them as an error,
- The frontend test tasks, to run NNEF tests, for both CPU and GPU.

These modifications were not significant in the scope of TVM but were necessary for complete CI integration.

#### Lint

The lint process uses numerous linting tools, to check every file in the project. In the case of Python files, for example, it uses black, flake8, MyPY, and pylint for code quality. It also checks for License headers and documentation generation. I had to keep in mind all these while developing the code, that it must comply with these formats.

# 6 Conclusion

The frontend is considered currently done, and put on maintain development. The process to merge it into TVM, as was the original goal, is currently ongoing and is in the review state by the community. The request for change for the project has been accepted, and now we are waiting for final confirmation.

Even though not all the functions in NNEF can be handled by it, the supported functions cover the most modern use cases. So this is not recognized as a grave error, as the leftover functions have newer counterparts, which are preferred nowadays. I have done manual testing as well, and I have not come across a network that had unsupported operations during that, using the most common graphs, used for testing purposes, such as InceptionNet, AlexNet, and ResNet. During these, we were content with the speed of TVM, compared to other alternatives, such as ONNX.

## 6.1 Future plans

For the frontend to continue to grow, the remaining functions may be implemented as well, for completeness, but the converter is already working as is. Additionally, I could create customized operation passes, to further optimize NNEF specifically, but as TVM's optimizer is its strength, it would only be incremental increases.

# Bibliography


[1] The Khronos Group. *NNEF Overview*. URL: https://www.khronos.org/nnef (visited on 03/26/2024).

[2] Apache Software Foundation. *Apache TVM*. URL: https://tvm.apache.org/ (visited on 03/26/2024).

[3] Tianqi Chen et al. "{TVM}: An automated {End-to-End} optimizing compiler for deep learning". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 578–594.

[4] Apache Software Foundation. *TVM*. URL: https://github.com/apache/tvm/ (visited on 03/26/2024).

[5] *aiMotive*. URL: https://aimotive.com/ (visited on 03/26/2024).

[6] The Khronos Group. *NNEF-Tools*. URL: https://github.com/KhronosGroup/NNEF-Tools (visited on 03/26/2024).

[7] TVM. *Introduction to Relay IR*. URL: https://tvm.apache.org/docs/arch/relay_intro.html (visited on 04/02/2024).

[8] TVM. *IRModule*. URL: https://mlc.ai/docs/get_started/tutorials/ir_module.html (visited on 04/02/2024).

[9] *TLCPack*. URL: https://tlcpack.ai/ (visited on 04/02/2024).

[10] The Khronos Group. *NNEF-Model Zoo*. URL: https://github.com/KhronosGroup/NNEF-Tools/tree/main/models#nnef-model-zoo (visited on 04/15/2024).

[11] The Khronos Group. *NNEF Specification - v1.0.5*. URL: https://registry.khronos.org/NNEF/specs/1.0/nnef-1.0.5.html (visited on 04/26/2024).

[12] Apache Software Foundation. *Apache TVM Documentation*. URL: https://tvm.apache.org/docs/ (visited on 04/02/2024).

# Table of Functions

This chapter will list the supported conversion functions with their signature, parameters, their types and return value, group as per the NNEF specification [11].

## Unary operations

`copy_converter`(*data, **kwargs*)

Copies the input tensor.

**Parameters:**
- data(*relay.Expr*): Input tensor
- kwargs: Additional keyword arguments

`neg_converter`(*data, **kwargs*)

Elementwise negation converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- kwargs: Additional keyword arguments

`rcp_converter`(*data, **kwargs*)

Reciprocal converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- kwargs: Additional keyword arguments

`exp_converter`(*data, **kwargs*)

Exponential converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- kwargs: Additional keyword arguments

`log_converter`(*data, **kwargs*)

Logarithm converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- kwargs: Additional keyword arguments

`sin_converter`(*data, **kwargs*)

Sine converter.

**Parameters:** – data(*relay.Expr*): Input tensor
– kwargs: Additional keyword arguments

`cos_converter`(*data, **kwargs*)

Cosine converter.

**Parameters:** – data(*relay.Expr*): Input tensor
– kwargs: Additional keyword arguments

`tan_converter`(*data, **kwargs*)

Tangent converter.

**Parameters:** – data(*relay.Expr*): Input tensor
– kwargs: Additional keyword arguments

`sinh_converter`(*data, **kwargs*)

Hyperbolic sine converter.

**Parameters:** – data(*relay.Expr*): Input tensor
– kwargs: Additional keyword arguments

`cosh_converter`(*data, **kwargs*)

Hyperbolic cosine converter.

**Parameters:** – data(*relay.Expr*): Input tensor
– kwargs: Additional keyword arguments

`tanh_converter`(*data, **kwargs*)

Hyperbolic tangent converter.

**Parameters:** – data(*relay.Expr*): Input tensor
– kwargs: Additional keyword arguments

`asin_converter`(*data, **kwargs*)

Arcsine converter.

**Parameters:** – data(*relay.Expr*): Input tensor
– kwargs: Additional keyword arguments

`acos_converter`(*data, **kwargs*)

Arccosine converter.

**Parameters:** – data(*relay.Expr*): Input tensor
– kwargs: Additional keyword arguments

`atan_converter`(*data, **kwargs*)

Arctangent converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- kwargs: Additional keyword arguments

`asinh_converter`(*data, **kwargs*)

Hyperbolic arcsine converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- kwargs: Additional keyword arguments

`acosh_converter`(*data, **kwargs*)

Hyperbolic arccosine converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- kwargs: Additional keyword arguments

`atanh_converter`(*data, **kwargs*)

Hyperbolic arctangent converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- kwargs: Additional keyword arguments

`abs_converter`(*data, **kwargs*)

Absolute value converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- kwargs: Additional keyword arguments

`sign_converter`(*data, **kwargs*)

Sign function converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- kwargs: Additional keyword arguments

`not_converter`(*data, **kwargs*)

Logical not converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- kwargs: Additional keyword arguments

`floor_converter`(*data, **kwargs*)

Flooring converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- kwargs: Additional keyword arguments

`ceil_converter`(*data, **kwargs*)

Ceiling converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- kwargs: Additional keyword arguments

`round_converter`(*data, **kwargs*)

Rounding converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- kwargs: Additional keyword arguments

## Binary operations

`add_converter`(*lhs, rhs, **kwargs*)

Elementwise addition converter.

**Parameters:**
- lhs(*relay.Expr*): Left-hand side input tensor
- rhs(*relay.Expr*): Right-hand side input tensor
- kwargs: Additional keyword arguments

`sub_converter`(*lhs, rhs, **kwargs*)

Elementwise subtraction converter.

**Parameters:**
- lhs(*relay.Expr*): Left-hand side input tensor
- rhs(*relay.Expr*): Right-hand side input tensor
- kwargs: Additional keyword arguments

`mul_converter`(*lhs, rhs, **kwargs*)

Elementwise multiplication converter.

**Parameters:**
- lhs(*relay.Expr*): Left-hand side input tensor
- rhs(*relay.Expr*): Right-hand side input tensor
- kwargs: Additional keyword arguments

`div_converter`(*lhs, rhs, **kwargs*)

Elementwise division converter.

**Parameters:**
- lhs(*relay.Expr*): Left-hand side input tensor
- rhs(*relay.Expr*): Right-hand side input tensor
- kwargs: Additional keyword arguments

`pow_converter`(*lhs, rhs, **kwargs*)

Elementwise power converter.

**Parameters:**
- lhs(*relay.Expr*): Left-hand side input tensor
- rhs(*relay.Expr*): Left-hand side input tensor
- kwargs: Additional keyword arguments

`lt_converter`(*lhs, rhs, **kwargs*)

Elementwise less-than comparison converter.

**Parameters:**
- lhs(*relay.Expr*): Left-hand side input tensor
- rhs(*relay.Expr*): Right-hand side input tensor
- kwargs: Additional keyword arguments

`gt_converter`(*lhs, rhs, **kwargs*)

Elementwise greater-than comparison converter.

**Parameters:**
- lhs(*relay.Expr*): Left-hand side input tensor
- rhs(*relay.Expr*): Right-hand side input tensor
- kwargs: Additional keyword arguments

`le_converter`(*lhs, rhs, **kwargs*)

Elementwise less-than or equal comparison converter.

**Parameters:**
- lhs(*relay.Expr*): Left-hand side input tensor
- rhs(*relay.Expr*): Right-hand side input tensor
- kwargs: Additional keyword arguments

`ge_converter`(*lhs, rhs, **kwargs*)

Elementwise greater-than or equal comparison converter.

**Parameters:**
- lhs(*relay.Expr*): Left-hand side input tensor
- rhs(*relay.Expr*): Right-hand side input tensor
- kwargs: Additional keyword arguments

`eq_converter`(*lhs, rhs, **kwargs*)

Elementwise equality comparison converter.

**Parameters:**
- lhs(*relay.Expr*): Left-hand side input tensor
- rhs(*relay.Expr*): Right-hand side input tensor
- kwargs: Additional keyword arguments

`ne_converter`(*lhs, rhs, **kwargs*)

Elementwise inequality comparison converter.

**Parameters:**
- lhs(*relay.Expr*): Left-hand side input tensor
- rhs(*relay.Expr*): Right-hand side input tensor
- kwargs: Additional keyword arguments

`and_converter`(*lhs, rhs, **kwargs*)

Elementwise logical AND converter.

**Parameters:**
- lhs(*relay.Expr*): Left-hand side input tensor
- rhs(*relay.Expr*): Right-hand side input tensor
- kwargs: Additional keyword arguments

`or_converter`(*lhs, rhs, **kwargs*)

Elementwise logical OR converter.

**Parameters:**
- lhs(*relay.Expr*): Left-hand side input tensor
- rhs(*relay.Expr*): Right-hand side input tensor
- kwargs: Additional keyword arguments

## Select function

`select_converter`(*condition, t_ val, f_ val, **kwargs*)

Conditional selection converter.

**Parameters:**
- condition(*relay.Expr  bool*): Boolean condition tensor  or value
- t_val(*relay.Expr*): Value tensor if condition is true
- f_val(*relay.Expr*): Value tensor if condition is false
- kwargs: Additional keyword arguments

## Simplifier operations

`sqr_converter`(*data, **kwargs*)

Square converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- kwargs: Additional keyword arguments

`sqrt_converter`(*data, **kwargs*)

Square root converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- kwargs: Additional keyword arguments

`rsqr_converter`(*data, **kwargs*)

Reciprocal square converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- kwargs: Additional keyword arguments

`rsqrt_converter(data, **kwargs)`

Reciprocal square root converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- kwargs: Additional keyword arguments

`log2_converter(data, **kwargs)`

Base-2 logarithm converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- kwargs: Additional keyword arguments

`min_converter(lhs, rhs, **kwargs)`

Elementwise minimum value converter.

**Parameters:**
- lhs(*relay.Expr*): Left-hand side input tensor
- rhs(*relay.Expr*): Right-hand side input tensor
- kwargs: Additional keyword arguments

`max_converter(lhs, rhs, **kwargs)`

Elementwise maximum value converter.

**Parameters:**
- lhs(*relay.Expr*): Left-hand side input tensor
- rhs(*relay.Expr*): Right-hand side input tensor
- kwargs: Additional keyword arguments

`clamp_converter(x, a, b, **kwargs)`

Value clamping converter.

**Parameters:**
- x(*relay.Expr*): Input tensor to be clamped
- a(*tvm.Constant  tvm.Expr*): Lower bound constant  or tensor
- b(*tvm.Constant  tvm.Expr*): Upper bound constant  or tensor
- kwargs: Additional keyword arguments

## Sliding-window operations

`conv_converter`(*data, kernel, bias, border, stride, padding, dilation, groups, **kwargs*)

Convolution converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- kernel(*relay.Expr*): Convolutional kernel tensor
- bias(*relay.Expr*): Bias tensor
- border(*str*): Border handling strategy
- stride(*tuple*): Stride size
- padding(*tuple*): Padding size
- dilation(*tuple*): Dilation size
- groups(*int*): Number of groups
- kwargs: Additional keyword arguments

`deconv_converter`(*data, kernel, bias, border, stride, padding, dilation, output_shape, groups, **kwargs*)

Deconvolution converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- kernel(*relay.Expr*): Deconvolutional kernel tensor
- bias(*relay.Expr*): Bias tensor
- border(*str*): Border handling strategy
- stride(*tuple*): Stride size
- padding(*tuple*): Padding size
- dilation(*tuple*): Dilation size
- output_shape(*tuple*): Output shape
- groups(*int*): Number of groups
- kwargs: Additional keyword arguments

`box_converter`(*data, size, border, padding, stride, dilation, normalize, **kwargs*)

Box operator converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- size(*list*): Size of the box filter
- border(*str*): Border handling strategy
- padding(*tuple*): Padding size
- stride(*tuple*): Stride size
- dilation(*tuple*): Dilation size
- normalize(*bool*): Normalize flag
- kwargs: Additional keyword arguments

`debox_converter`(*data, size, border, padding, stride, dilation, normalize, output_ shape, **kwargs*)

Debox operator converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- size(*list*): Size of the box filter
- border(*str*): Border handling strategy
- padding(*tuple*): Padding size
- stride(*tuple*): Stride size
- dilation(*tuple*): Dilation size
- normalize(*bool*): Normalize flag
- output_shape(*tuple*): Output shape
- kwargs: Additional keyword arguments

`nearest_downsample_converter`(*data, factor, **kwargs*)

Nearest neighbour downsample converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- factor(*tuple*): Downsampling factor
- kwargs: Additional keyword arguments

`area_downsample_converter`(*data, factor, **kwargs*)

Area downsample converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- factor(*tuple*): Downsampling factor
- kwargs: Additional keyword arguments

`nearest_upsample_converter`(*data, factor, **kwargs*)

Nearest neighbour upsample converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- factor(*tuple*): Upsampling factor
- kwargs: Additional keyword arguments

`multilinear_upsample_converter`(*data, factor, method, border, **kwargs*)

Multilinear upsampling converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- factor(*tuple*): Upsampling factor
- method(*str*): Interpolation method
- border(*str*): Border handling strategy
- kwargs: Additional keyword arguments

## Reduce operations

`sum_reduce_converter`(*data, axes, normalize, keepdims=True, **kwargs*)

Sum reduction converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- axes(*list*): Axes to reduce
- normalize(*bool*): Flag to normalize the result
- keepdims(*bool*): Keep dimensions after reduction
- kwargs: Additional keyword arguments

`max_reduce_converter`(*data, axes, keepdims=True, **kwargs*)

Max reduction converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- axes(*list*): Axes to reduce
- keepdims(*bool*): Keep dimensions after reduction
- kwargs: Additional keyword arguments

`min_reduce_converter`(*data, axes, keepdims=True, **kwargs*)

Min reduction converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- axes(*list*): Axes to reduce
- keepdims(*bool*): Keep dimensions after reduction
- kwargs: Additional keyword arguments

`argmax_reduce_converter`(*data, axes, keepdims=True, **kwargs*)

Argmax reduction converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- axes(*list*): Axes to reduce
- keepdims(*bool*): Keep dimensions after reduction
- kwargs: Additional keyword arguments

`argmin_reduce_converter`(*data, axes, keepdims=True, **kwargs*)

Argmin reduction converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- axes(*list*): Axes to reduce
- keepdims(*bool*): Keep dimensions after reduction
- kwargs: Additional keyword arguments

`all_reduce_converter`(*data, axes, keepdims=True, **kwargs*)

All reduction converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- axes(*list*): Axes to reduce
- keepdims(*bool*): Keep dimensions after reduction
- kwargs: Additional keyword arguments

`any_reduce_converter`(*data, axes, keepdims=True, **kwargs*)

Any reduction converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- axes(*list*): Axes to reduce
- keepdims(*bool*): Keep dimensions after reduction
- kwargs: Additional keyword arguments

`mean_reduce_converter`(*data, axes, keepdims=True, **kwargs*)

Mean reduction converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- axes(*list*): Axes to reduce
- keepdims(*bool*): Keep dimensions after reduction
- kwargs: Additional keyword arguments

## Tensor shape operations

`reshape_converter`(*data, shape, axis_ start, axis_ count, **kwargs*)

Reshape converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- shape(*list*): New shape
- axis_start(*int*): Start axis for reshaping
- axis_count(*int*): Number of axes to reshape
- kwargs: Additional keyword arguments

`squeeze_converter`(*data, axes, **kwargs*)

Squeeze converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- axes(*list*): Axes to squeeze
- kwargs: Additional keyword arguments

`unsqueeze_converter`(*data, axes, **kwargs*)

Unsqueeze converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- axes(*list*): Axes to unsqueeze
- kwargs: Additional keyword arguments

`transpose_converter`(*data, axes, **kwargs*)

Transpose converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- axes(*list*): Axes order for transposition
- kwargs: Additional keyword arguments

`split_converter`(*data, axis, ratios, **kwargs*)

Split converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- axis(*int*): Axis along which to split
- ratios(*list*): Ratios for splitting
- kwargs: Additional keyword arguments

`concat_converter`(**data, axis, **kwargs*)

Concatanation converter.

**Parameters:**
- data(*tuple of relay.Expr*): Tensors to concatenate
- axis(*int*): Axis along which to concatenate
- kwargs: Additional keyword arguments

`stack_converter`(**data, axis, **kwargs*)

Stack converter.

**Parameters:**
- data(*tuple of relay.Expr*): Tensors to stack
- axis(*int*): Axis along which to stack
- kwargs: Additional keyword arguments

`unstack_converter`(*data, axis, **kwargs*)

Unstack converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- axis(*int*): Axis along which to unstack
- kwargs: Additional keyword arguments

`slice_converter`(*data, axes, begin, end, stride, **kwargs*)

Slice converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- axes(*list*): Axes to slice
- begin(*list*): Start indices for slicing
- end(*list*): End indices for slicing
- stride(*list*): Stride for slicing
- kwargs: Additional keyword arguments

`pad_converter`(*data, padding, border, value, **kwargs*)

Padding converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- padding(*list*): Padding amounts for each dimension
- border(*str*): Padding mode ("constant"
- "replicate"
- "reflect")
- value(*float*): Padding value for constant mode
- kwargs: Additional keyword arguments

`tile_converter`(*data, repeats, **kwargs*)

Tile converter.

**Parameters:**
- data(*relay.Expr*): Input tensor
- repeats(*list*): Repeats for each dimension
- kwargs: Additional keyword arguments

# Matmul function

`matmul_converter`(*a, b, **kwargs*)

Matrix multiplication converter. Real signature: matmul_converter(a, b, transposeA, transposeB)

**Parameters:**
- a(*relay.Expr*): Left-hand matrix
- b(*relay.Expr*): Right-hand matrix
- transposeA(*bool*): Whether to transpose the left-hand matrix
- transposeB(*bool*): Whether to transpose the right-hand matrix
- kwargs: Additional keyword arguments

## Compound operations

`sigmoid_converter`(*data, **kwargs*)

Sigmoid converter

**Parameters:**
- data(*relay.Expr*): Input tensor
- kwargs: Additional keyword arguments

`relu_converter`(*data, **kwargs*)

Rectified Linear Unit converter

**Parameters:**
- data(*relay.Expr*): Input tensor
- kwargs: Additional keyword arguments

`prelu_converter`(*data, alpha, **kwargs*)

Parametric Rectified Linear Unit converter

**Parameters:**
- data(*relay.Expr*): Input tensor
- alpha(*relay.Expr*): Input tensor
- kwargs: Additional keyword arguments

`leaky_relu_converter`(*data, alpha, **kwargs*)

Leaky Rectified Linear Unit converter

**Parameters:**
- data(*relay.Expr*): Input tensor
- alpha(*relay.Expr*): Slope of the negative part of the function
- kwargs: Additional keyword arguments

`elu_converter`(*data, alpha, **kwargs*)

Exponential Linear Unit converter

**Parameters:**
- data(*relay.Expr*): Input tensor
- alpha(*relay.Expr*): The alpha value in the ELU formulation
- kwargs: Additional keyword arguments

`selu_converter`(*data, alpha, **kwargs*)

Scaled Exponential Linear Unit converter

**Parameters:**
- data(*relay.Expr*): Input tensor
- alpha(*relay.Expr*): The alpha value in the SELU formulation
- lambda(*relay.Expr*): The lambda value in the SELU formulation

`gelu_converter`(*data, **kwargs*)

Gaussian Error Linear Unit converter

**Parameters:**
- data(*relay.Expr*): Input tensor
- kwargs: Additional keyword arguments

`silu_converter`(*data, **kwargs*)

Sigmoid Linear Unit converter

**Parameters:**
- data(*relay.Expr*): Input tensor
- kwargs: Additional keyword arguments

`softmax_converter`(*data, axes, **kwargs*)

Softmax converter

**Parameters:**
- data(*relay.Expr*): Input tensor
- axes(*list*): List of axes along which to perform the softmax operation
- kwargs: Additional keyword arguments

`softplus_converter`(*data, **kwargs*)

Softplus converter

**Parameters:**
- data(*relay.Expr*): Input tensor
- kwargs: Additional keyword arguments

## Linear operations

`linear_converter`(*data, _filter, bias, **kwargs*)

Linear converter

**Parameters:**
- data(*relay.Expr*): Input tensor
- _filter(*relay.Expr*): Filter tensor
- bias(*relay.Expr*): Bias tensor
- kwargs: Additional keyword arguments

`separable_conv_converter`(*data, plane_filter, point_filter, bias, border, padding, stride, dilation, groups, **kwargs*)

Separable convolution converter

**Parameters:**
- data(*relay.Expr*): Input tensor
- plane_filter(*relay.Expr*): Filter tensor for the plane-wise convolution
- point_filter(*relay.Expr*): Filter tensor for the point-wise convolution
- bias(*relay.Expr*): Bias tensor
- border(*str*): Type of border padding
- padding(*list*): Padding configuration
- stride(*list*): Stride configuration
- dilation(*list*): Dilation configuration
- groups(*int*): Number of groups
- kwargs: Additional keyword arguments

`separable_deconv_converter`(*data, plane_filter, point_filter, bias, border, padding, stride, dilation, output_shape, groups, **kwargs*)

Separable deconvolution converter

**Parameters:**
- data(*relay.Expr*): Input tensor
- plane_filter(*relay.Expr*): Filter tensor for the plane-wise deconvolution
- point_filter(*relay.Expr*): Filter tensor for the point-wise deconvolution
- bias(*relay.Expr*): Bias tensor
- border(*str*): Type of border padding
- padding(*list*): Padding configuration
- stride(*list*): Stride configuration
- dilation(*list*): Dilation configuration
- output_shape(*list*): Output shape configuration
- groups(*int*): Number of groups
- kwargs: Additional keyword arguments

`max_pool_converter`(*data, size, border, padding, stride, dilation, **kwargs*)

Max pool converter

**Parameters:**
- data(*relay.Expr*): Input tensor
- size(*list*): Pooling window size
- border(*str*): Type of border padding
- padding(*list*): Padding configuration
- stride(*list*): Stride configuration
- dilation(*list*): Dilation configuration
- kwargs: Additional keyword arguments

`avg_pool_converter`(*data, size, border, padding, stride, dilation, **kwargs*)

Avg pool converter

**Parameters:**
- data(*relay.Expr*): Input tensor
- size(*list*): Pooling window size
- border(*str*): Type of border padding
- padding(*list*): Padding configuration
- stride(*list*): Stride configuration
- dilation(*list*): Dilation configuration
- kwargs: Additional keyword arguments

`rms_pool_converter`(*data, size, border, padding, stride, dilation, **kwargs*)

Rms pool converter

**Parameters:**
- data(*relay.Expr*): Input tensor
- size(*list*): Pooling window size
- border(*str*): Type of border padding
- padding(*list*): Padding configuration
- stride(*list*): Stride configuration
- dilation(*list*): Dilation configuration
- kwargs: Additional keyword arguments

`local_response_normalization_converter`(*data, size, alpha, beta, bias*)

Local Response Normalization converter

**Parameters:**
- data(*relay.Expr*): Input tensor
- size(*list*): Size of the local region
- alpha(*relay.Expr*): Scaling parameter
- beta(*relay.Expr*): Power parameter
- bias(*relay.Expr*): Bias tensor

`local_mean_normalization_converter`(*data, size, **kwargs*)

Local Mean Normalization converter

**Parameters:**
- data(*relay.Expr*): Input tensor
- size(*list*): Size of the local region
- kwargs: Additional keyword arguments

`local_variance_normalization_converter`(*data, size, bias, epsilon, **kwargs*)

Local Variance Normalization converter

**Parameters:**
- data(*relay.Expr*): Input tensor
- size(*list*): Size of the local region
- bias(*float*): Bias value
- epsilon(*float*): Epsilon value
- kwargs: Additional keyword arguments

`local_contrast_normalization_converter`(*data, size, bias, epsilon, **kwargs*)

Local Contrast Normalization converter

**Parameters:**
- data(*relay.Expr*): Input tensor
- size(*list*): Size of the local region
- bias(*float*): Bias value
- epsilon(*float*): Epsilon value
- kwargs: Additional keyword arguments

`l1_normalization_converter`(*data, axes, bias, epsilon, **kwargs*)

L1 Norm Converter

**Parameters:**
- data(*relay.Expr*): Input tensor
- axes(*list*): Axes over which to normalize
- bias(*float*): Bias value
- epsilon(*float*): Epsilon value
- kwargs: Additional keyword arguments

`l2_normalization_converter`(*data, axes, bias, epsilon, **kwargs*)

L2 Norm Converter

**Parameters:**
- data(*relay.Expr*): Input tensor
- axes(*list*): Axes over which to normalize
- bias(*float*): Bias value
- epsilon(*float*): Epsilon value
- kwargs: Additional keyword arguments

`batch_normalization_converter`(*data, mean, variance, offset, scale, epsilon, **kwargs*)

Batch Normalization Converter

**Parameters:**
- data(*relay.Expr*): Input tensor
- mean(*relay.Expr*): Mean tensor
- variance(*relay.Expr*): Variance tensor
- offset(*relay.Expr*): Offset tensor
- scale(*relay.Expr*): Scale tensor
- epsilon(*float*): Epsilon value
- kwargs: Additional keyword arguments