DeepViewRT User Manual

Release 2.4.39

Au-Zone Technologies

Mar 22, 2022

CONTENTS

CHAPTER

ONE

INTRODUCTION

Au-Zone's DeepViewRT implements a neural network inference engine suitable for embedded processors, graphics processors, and microcontrollers. The engine is also fully supported on standard workstations running Windows, macOS, and Linux for development purposes. The supporting tools target the standard workstation environments for generating models and interfacing with remote targets for test framework integration.

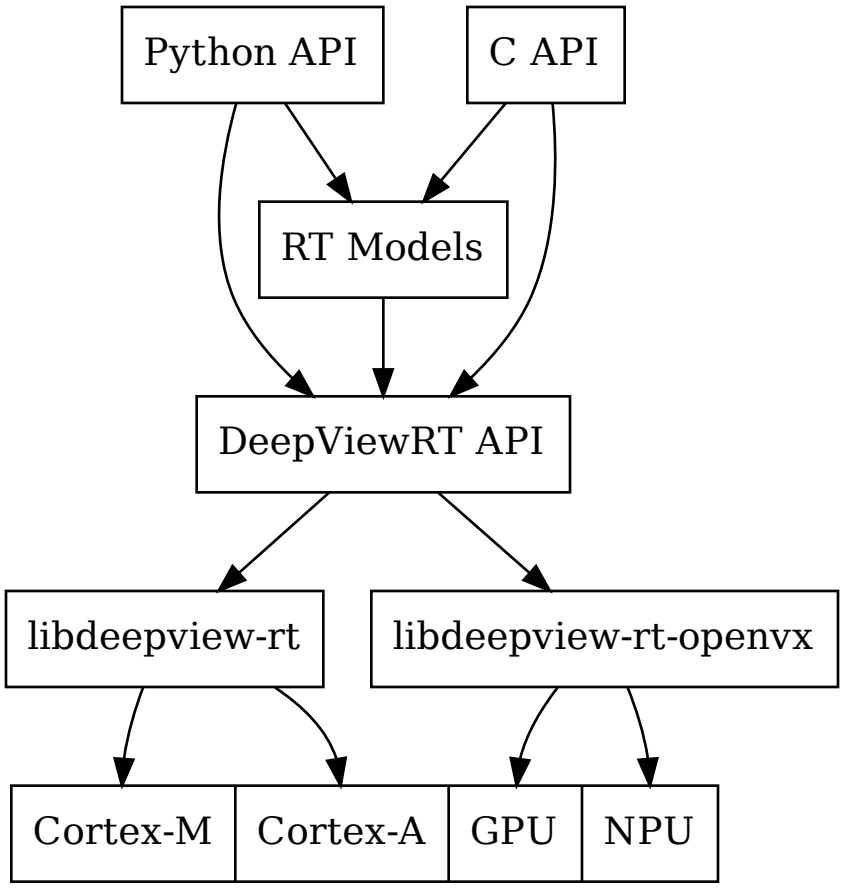
1.1 Compute Engines

The DeepViewRT API is implemented through a native C/ASM library for Cortex-M MCU and Cortex-A CPU and x86_64 for testing on the host. DeepViewRT also implements an OpenVX based engine to enable processing on GPU and NPU accelerators. This provides a common API across all processing engines.

The base engine which is always available is the CPU engine and this is the default NN API implementation. The CPU engine can leverage SIMD extensions of the CPU if they are available, these are probed at runtime so the library binary is portable across processors of the same familly.

The DeepViewRT Models are implemented using the NN API and allow for models to be deployed to all targets without explicitly programming the computation graph.

DeepViewRT can be programmed from a C API or from a Python API. The Python API is a direct wrapper around the C API (NN API) implemented using CFFI and is portable from Python 2.7 through Python 3.x.


Python API
C API
RT Models
DeepViewRT API
libdeepview-rt
libdeepview-rt-openvx
Cortex-M
Cortex-A
GPU
NPU

CHAPTER

TWO

QUICKSTART

DeepViewRT quickstart covers a few examples start from the Python API. For additional Python introduction refer to the ModelRunner chapter under the ModelClient Python API section for how to interface with the ModelRunner from a Python API.

2.1 Python

The DeepViewRT Python API provides a high-level "Pythonic" interface to the low level C API. It is appropriate for development and testing, and possibly deployment depending on the performance and integration requirements of a target platform. Refer to the Python chapter for further details.

2.1.1 Primitives

The primitive operations (or layers) can be accessed through the deepview.rt.ops namespace. The functions all follow the same pattern of taking Tensors for data and possibly lists and/or scalars as parameters before returning the resulting data as a Tensor. All operation functions accept an optional parameter "out" which, if provided, will be used as the output Tensor thus avoiding creation of a new object at evaluation. This out tensor must be appropriately sized to hold the result, a useful pattern is to have it "lazy loaded" as follows.

```
import deepview.rt as rt
import numpy as np

# Our result tensor is initially defined as None
result = None

a = np.random.rand(100)

# Here out is set to result, which is initially None so will be created.
result = rt.ops.relu(a, out=result)

# Here out is set to result, which is now the result of the previous call.
# This could've been a loop where the result Tensor is initialized on the
```

(continues on next page)

(continued from previous page)

```
# first loop then reused in following runs.
result = rt.ops.relu(a, out=result)
```

Though the primitives all operate on Tensors, Numpy arrays can be used in their place in which case a Tensor wrapper will be created on demand. This results in a Tensor object being created and released in the call though the Numpy array memory will be used directly.

The following example demonstrates the matrix multiply primitive and comparing the result to Numpy's implementation. Note that Tensors will provide a string representation when evaluated but to use them from a Numpy function the array() function should be called which will return a valid Numpy array copy of the data. Alternatively mapro() (or maprw for a writeable version) can be called which will return a Numpy array mapped to the Tensor's memory, once no longer needed unmap() must be called to release the mapping.

```
import deepview.rt as rt
import numpy as np

a = np.random.rand(3, 4)
b = np.random.rand(4, 2)
C = rt.ops.matmul(a, b)

np.allclose(np.dot(a,b), C.array())
```

2.1.2 Tensor Maps

Tensors can be mapped into Numpy arrays, this works whether the Tensor lives on the CPU or GPU. The map() function provides a Numpy array with data backed by up-to-date tensor data (at time of function call) but on release will not cause the tensor to be updated unless it was mapped with the writeable parameter set to true. Note the behaviour of updates can vary when using a compute device engine plugin (such as OpenCL plugin).

```
import deepview.rt as rt
import numpy as np

t = rt.Tensor(shape=(3, 4))
t.fill(3.14)

n = t.map()
print(n) # should be a [3,4] array filled with 3.14

del n # delete n before unmapping to avoid invalid memory
t.unmap()

n = t.map(True) # mapped writable, will be able to update contents.
n[:] = np.random.rand(3,4).astype(np.float32)
```

(continues on next page)

(continued from previous page)

```
del n
t.unmap()

a = t.array() # return an array copy of the Tensor data
print(a) # should now contain random data instead of 3.14
```

2.2 ModelRunner

ModelRunner provides an HTTP Webservice API for loading and running models on target as well as a high performance UNIX Socket IPC channel appropriate for low latency video processing. Refer to the ModelRunner chapter for further details.

CHAPTER

THREE

DEEPVIEW MODELRUNNER

DeepView ModelRunner is a dedicated service for hosting and evaluating DeepViewRT Models (RTM).

3.1 HTTP REST API

ModelRunner provides an HTTP REST API for loading an evaluating models.

3.1.1 cURL Examples

The following are examples demonstrating the REST API using cURL command line tool.

This first example sends an image (PNG or JPEG) to ModelRunner and expects the response to contain the inferred label as well as additional timing information in the headers.

Note: the –data-binary parameter requires a leading @ to force it to read the file with FILENAME instead of sending FILENAME string as data.

```
curl -D - -XPOST -H 'Accept: text/plain' -H 'Content-Type: image/*' --data-binary
↪'@FILENAME' 'http://localhost:10810/v1?run=1'
```

Removing the '-H "Accept: text/plain"' will result in the default application/json response instead.

```
curl -D - -XPOST -H 'Content-Type: image/*' --data-binary '@FILENAME' 'http://
↪localhost:10810/v1?run=1'
```

If you would like to receive the output tensor's contents as opposed to the label you can use the output parameter, output should be set to the name of the layer you wish to read (typically named output).

```
curl -D - -XPOST -H 'Content-Type: image/*' --data-binary '@FILENAME' 'http://
↪localhost:10810/v1?run=1&output=output'
```

3.1.2 ModelClient API

The ModelClient API is a way for users to communicate with the ModelRunner service through Python. The ModelClient class provides methods for loading a model, running a model, and gathering timing and layer information of the current model. The ModelClient can be initialized with just a URI, or a URI and model in the form of a bytearray or filepath to where the .rtm is located. A new URI can be set through a simple assignment and if no 'http://' is found it will be added to the URI.

```
from deepview.rt.modelclient import ModelClient

modelclient = ModelClient("http://localhost:10818/v1")
modelclient = ModelClient("http://localhost:10819/v1", "mobilenet.rtm")
modelclient.uri = localhost:10818/v1
print(modelclient.uri) # 'http://localhost:10818/v1' will be the new URI
```

Once a ModelClient has been created, a new model can be loaded overwriting the previous and allowing for the new model to be run.

```
from deepview.rt.modelclient import ModelClient

modelclient = ModelClient("http://localhost:10818/v1")
modelclient.load_model("mobilenet.rtm")
```

Once a model has been loaded, it can be run with the ModelClient method run that accepts a dictionary of inputs and a list of outputs that the user wishes to have returned with their associated numpy tensors. There is an additional timeout parameter to avoid the possibility of the script hanging if there is poor connectivity to the ModelRunner.

The return value from run is a dictionary that associates the given output names with their calculated tensor. When using a memory map, some return values may be overwritten if they use memory that a later layer is mapped to use as well.

```
import numpy as np
input_val = np.random.rand(10,160,160,3) * 255
inputs = {'input': input_val.astype(np.float32)}
outputs = ['output']
results = modelclient.run(inputs, outputs)
```

Once a model has been run it is possible to gain access to the timing and layer information of the given model. This provides information on how long it took for the model to be uploaded to the ModelRunner service, the time for the ModelRunner to receive the inputs and return the outputs to the user, as well as the evaluation time of the model. This can be gathered from the get_timing_info method. The ModelClient also provides access to the timing of each type of layer within the graph through the get_op_timing_info method.

The return value from get_op_timing_info is a dictionary of operation names, where each value is a list of the average time for that operation type, the total time taken by that operation and the number of times that operation was encountered within the model.

```
put_time, post_time, eval_time = modelclient.get_timing_info()
op_timing = modelclient.get_op_timing_info()
```

The ModelClient also provides access to the raw layer information through the get_layers method.

This returns a dictionary of all of the layer names with their associated inputs, type, tensor shape, datatype, and timing info in nanoseconds.

```
model_layers = modelclient.get_layers()
```

CHAPTER

FOUR

MEMORY MANAGEMENT

4.1 NN API Memory

The NN API (C API) collection of functions will only allocate memory from the heap from the _init functions, this will always be documented and will also always allow for the user to provide a custom buffer to avoid heap allocations within the library. The library can be used without a heap, avoiding all usage of malloc/free. On 32bit platforms stack usage of any function will be documented when above 256 bytes, and those cases are restricted to remain under 2KB of stack usage (a few exceptional cases are documented).

Generally speaking, functions which would be called continuously such as the primitive operations and model evaluation never allocate from the heap. This means heap allocations, if used, can be constrained to initialization time then runtime memory usage will not change outside of stack usage.

4.1.1 Opaque Objects

Data structures are implemented as opaque objects, where the public headers will define an empty struct type definition (typedef struct {} NNObject) which the client can only interact with through a pointer. To allow heapless or finer grained control of memory each object can be initialized over user defined memory, the pattern is to have nn_object_init(void memory) accept a pointer to at least nn_object_sizeof() bytes (static define NN_OBJECT_SIZEOF also provided) with which the object will use to store its internal data structure. If the memory parameter is NULL then the library will internally allocate the required memory from the heap using malloc.

The static size definitions and custom memory parameters are mainly useful for MCU development where fine grained memory memory control is important. It allows for constructs such as the following to be defined, in this case both the tensor object as well as the data are user provided and statically defined.

```
NNTensor *tensor = NULL;
int32_t tensor_shape[] = { 3, 4 };
uint8_t tensor_memory[NN_TENSOR_SIZEOF];
float tensor_data[3 * 4];

main()
```

(continues on next page)

(continued from previous page)

```
{
    // Initialize the tensor structure.
    tensor = nn_tensor_init(tensor_memory, NULL);

    // Allocate the tensor data using a pre-defined buffer.
    nn_tensor_assign(tensor,            // tensor object being assigned
                     NNTensorType_F32,  // datatype, F32 is float32.
                     2,                 // dimensions in the shape array
                     tensor_shape,      // shape array, also defines minimum size of␣
↪tensor_data.
                     tensor_data);      // user supplied buffer

    // Releases the tensor, tensor_memory and tensor_data remain valid.
    nn_tensor_release(tensor);
}
```

The definition NN_OBJECT_SIZEOF will be padded out with extra memory to allow for future extensions to the data structures. By contrast nn_object_sizeof() will return the actual size of the structure, so this requires slightly less memory but requires runtime probing. Which is most appropriate is up to the implementor to decide.

The following example demonstrates an alternate version of the above but using dynamic allocations.

```
NNTensor *tensor = NULL;
int32_t tensor_shape[] = { 3, 4 };

main()
{
    // Initialize the tensor structure, nn_tensor_init will allocate the
    // required memory from the heap.
    tensor = nn_tensor_init(NULL, NULL);

    // Allocate the tensor data using a buffer from the heap.
    nn_tensor_alloc(tensor,             // tensor object being assigned
                    NNTensorType_F32,   // datatype, F32 is float32.
                    2,                  // dimensions in the shape array
                    tensor_shape);      // shape array, internal data size␣
↪calculated from this shape.

    // Releases the tensor, internal memory and data buffer will also be released.
    nn_tensor_release(tensor);
}
```

CHAPTER

FIVE

TOOLCHAIN SUPPORT

This chapter outlines the toolchain requirements to use the DeepViewRT library across the various supported platforms, including MCU targets.

5.1 Cortex-M

DeepViewRT is provided for Cortex-M4F and Cortex-M7F (technically all are F) MCU devices. Currently the FPU is a requirement and both the FPv4 and FPv5 variants are supported for M4 and M7, respectively.

The library is packaged following guidelines from the ARM ABI Compatibility documents, with the following parameters.

- **wchar_t uses 4-bytes.**
  - gcc -fno-short-wchar
  - note wchar_t is not used anywhere in deepview.
- **enum uses ints.**
  - gcc -fno-short-enums
- fpu fpv4-sp and fpv5-sp
- **hard-float calling conventions**
  - gcc -mfloat-abi=hard
  - this means floating point arguments are passed using registers.

APPENDIX

A

PYTHON API REFERENCE

The Python API using the CFFI module to interact directly with the C ABI but providing a Pythonic interface. As we're using CFFI in on-line ABI mode the bindings are portable across various Python versions from 2.7 and 3.4 onwards.

A.1 DeepViewRT Core

DeepViewRT for Python provides Python bindings for the target run-time of the DeepView Machine Learning Toolkit implementing neural network primitives and model inference support. The Python bindings are implemented using the CFFI module to interact directly with the DeepViewRT C ABI. To use this module, libdeepview-rt.so needs to be available in the library search path along with any engines that are to be used (ex: deepview-rt-opencl.so). On Windows the library will be called deepview-rt.dll and on MacOSX it will be called libdeepview-rt.dylib, otherwise the usage remains the same.

A.2 ModelClient

The deepview.rt.ModelClient class provides a Python interface to the ModelRunner service. It offers a main routine which is callable from the command-line along with the Python API.

class ModelClient(*uri*, *rtm=None*, *timeout=100*, *callback=<built-in function print>*, *upload_here=True*)

Bases: `object`

The ModelClient class is responsible for all interactions with the modelrunner application through Python.

default_input_quant_scale = 0.0078125

default_input_zero_point = 128

default_output_quant_scale = 0.00390625

default_output_zero_point = 0

__init__(*uri*, *rtm=None*, *timeout=100*, *callback=<built-in function print>*, *upload_here=True*)
Initializes the ModelClient class. If given an rtm it attempts to load it at the given URI, where a modelrunner application should be running.

@param uri A string that contains the URI. @param rtm A string, bytearray, or Path that represents the actual rtm model or a path to a .rtm file (default=None).

upload_model()

stop_uploading()

property uri
Returns the current URI that the ModelClient is trying to communicate with.

get_model_name()
Returns the name of the current model, if one exists, at the given URI.

get_layers()
Returns all layer information of the current model, if one exists, at the current URI.

get_io_quantization()

get_labels(*id=None*)

get_layer_timing_info()

get_op_timing_info()
Returns the timing information for all layer types that exist within the current model, if one exists, at the current URI.

@return A dictionary where the keys are the names of the operations within the model and the value is a list of [avg_time, total_time, number of layers of the given type].

get_timing_info()
Returns the timing information regarding how long it took to send the most recent rtm to the modelrunner application, the total time to run the model (includes the post request and evaluation), and the time it took for the modelrunner application to run the model.

get_runner_timings()
Returns the timings values from modelruuner/v1/model {timings key} these thre values make reference to the image decode time, input time and output time

load_model(*rtm*, *uri=None*, *timeout=100*, *callback=<built-in function print>*)
Attempts to load a given rtm to the modelrunner application at the given URI, or the current URI if none is provided.

@param rtm A string, bytearray, or Path that represents the actual rtm model or a path to a .rtm file. @param uri A string that contains the URI (default=None).

run(*inputs=None*, *outputs=None*, *timeout=None*, *headers=None*, *params=None*)
Attempts to run the model that is currently loaded into the modelrunner application at the current URI, using the inputs provided.

@param inputs A dictionary of inputs that correspond to tensors within the model. @param outputs A list of tensor names that exist within the model that the user wants

to have returned as tensors from the modelrunner. @param timeout A timeout value to provide to the post request. @return Given that outputs is not None, a dictionary of the tensor names and their associated tensors.

A.3 Tensor Class

class Tensor(*copy_from=None*, *shape=None*, *engine=None*, *dtype=<class 'numpy.float32'>*, *wrap=None*)

Bases: `object`

Tensor objects are the primary means of holding data within DeepViewRT. All operations use data through tensor objects.

Shapes are currently limited to a maximum of 4 dimensions, attempting to use more dimensions will raise an exception.

__init__(*copy_from=None*, *shape=None*, *engine=None*, *dtype=<class 'numpy.float32'>*, *wrap=None*)

Constructs a DeepViewRT tensor. If copy_from is set to a valid Numpy array it will be mapped into the Tensor and a reference will be held to ensure validity across the lifespan of the tensor, if shape is also provided the tensor will be reshaped accordingly.

Note that though copy_from will be mapped to the Tensor there's no guarantee that the memory will be used beyond initialization of a Engine backed tensor as this is defined by the specific engine implementation. As an alternative consider a role reversal where the tensor owns the data and a numpy array is retrieved through tensor's map/unmap methods.

If copy_from is None then shape can be used to allocate a zero initialized tensor. If neither copy_from nor shape is provided then an empty tensor is created.

The engine parameter can be used to create a Tensor using the specified engine plugin.

Parameters

- **copy_from** (`Optional[numpy.ndarray]`) – Create a tensor mapped from the specified numpy array.
- **shape** (`Optional[List[int]]`) – Shape (or reshape, if copy_from is provided) of the tensor.
- **engine** (`Optional[Engine]`) – Create the tensor using this engine.
- **dtype** (`numpy.dtype`) – Sets the underlying datatype, only numpy.float32 is currently supported.

Raises

- **TensorTypeUnsupportedError** – If dtype is an unknown or unsupported type.
- **Error** – If an internal error occurs.

engine_name()
The tensor's engine's name.

Returns Engine name or None if no engine is associated to tensor.

Return type str

engine_version()
The tensor's engine's version number.

Returns Engine version string or None if no engine is associated to tensor.

Return type str

volume()
Volume of the tensor's data which is the inner product of it's shape.

Returns The volume of the tensor.

Return type int

element_size()
Size in bytes of an individual tensor data element.

Returns Element size of the tensor.

Return type int

property ntype

Return type int

property dtype

Return type dtype

property time

Return type int

size()
Size in bytes of the tensor's data, this is equivalent to volume * element_size.

Returns The tensor's data size in bytes.

Return type int

property dims
Number of dimensions for this tensor.

Returns the number of dimensions for the tensor.

Return type int

property shape
Shape of the tensor's data.

Returns The shape of the tensor.

Return type Tuple[int, . . .]

reshape(*newshape*)
Reshapes the tensor representation, new shape must have the same volume as previous.

Raises **Error** – If reshape operation fails.

Return type Tuple[int, ...]

sync()
Synchronize the tensor and preceeding operations.

Raises **Error** – If synchronization fails.

Return type None

map(*writeable=True*)
Maps the tensor and returns a Numpy array map over the underlying buffer. The tensor must be unmapped once no longer needed, note that extreme care must be taken if calling unmap while the Numpy array continues to exist with a now invalid pointer. Another option is to use the array() method which will return a copy of the tensor's data which is safer though slower.

Parameters **writeable** (bool) – If true the tensor is mapped as writeable otherwise read-only

Returns A Numpy array mapped to the tensor's data.

Return type numpy.ndarray

Raises **TensorNoDataError** – If the map failed, likely because the tensor has no data.

unmap()
Unmaps the tensor.

Raises **Error** – on internal error.

Return type None

array()
Returns a copy of the tensor's data as a new numpy array.

Returns Numpy array of the tensor's data

Return type numpy.ndarray

Raises **TensorNoDataError** – if the tensor has no data.

copy_from(*x*)
Copies the Numpy array into the Tensor

Return type None

fill(*value*)
Fills the tensor with the provided value which will be casted to double then eventually to the tensor's data type.

Return type None

pad()

Return type Tensor

shuffle(*order*, *out=None*)

slice(*axes*, *head=None*, *tail=None*, *out=None*)

view(*shape*, *offset=0*, *dtype=<class 'numpy.float32'>*)

A.4 Engine Class

class Engine

Bases: object

The Engine class provides the interface to loading DeepViewRT Engine plugins such as the OpenCL engine (deepview-rt-opencl.so). It can then be used to initialize Tensors with this engine.

load(*plugin*)

unload()

name()

A.5 Operations

The available operations are more fully documented in the C API Operations reference though not all functions are implemented in the Python API.

The pattern to all functions is to return the output tensor and optionally accept the output tensor as parameter "out", when this parameter is not provided it will be created with the appropriate setup (shape, datatype, and engine). Since the function will return the out parameter regardless it can be used for lazy initialization as in the following example, here the c tensor is created by the add() function and reused by the subtract() function.

Example:

In cases where we wish to experiment with custom graphs without using RTM models they could be represented, for example, using the networkx module by having vertices representing the ops and edges representing the tensors. In this case the out tensors would be the edge leaving a vertex which would then be reused by the vertex on the other side of the edge as an input.

A.5.1 Array

concat(*inputs*, *axis*, *out=None*)

Concatenate the tensors from list into a single tensor along the axis.

Returns the concatenated tensor.

Return type *Tensor*

A.5.2 Math

add(*a*, *b*, *out=None*)

Add two tensors together, storing the results into the out tensor. If out is None it will be created with the expected shape of the output. In either case the resulting out tensor is returned.

Parameters

- **a** (Tensor) – The left-hand tensor.
- **b** (Tensor) – The right-hand tensor.
- **out** (Optional[Tensor]) – Optional result tensor, if not provided it will be created.

Returns The out tensor holding the results of the operation.

Return type *Tensor*

Raises **Error** – if an error happened.

subtract(*a*, *b*, *out=None*)

Subtract the b tensor from the a tensor, storing the results into the out tensor. If out is None it will be created with the expected shape of the output. In either case the resulting out tensor is returned.

Parameters

- **a** (Tensor) – The left-hand tensor.
- **b** (Tensor) – The right-hand tensor.
- **out** (Optional[Tensor]) – Optional result tensor, if not provided it will be created.

Returns The out tensor holding the results of the operation.

Return type *Tensor*

Raises **Error** – if an error happened.

multiply(*a*, *b*, *out=None*)

Multiply the a and b tensors, storing the results into the out tensor. If out is None it will be created with the expected shape of the output. In either case the resulting out tensor is returned.

Parameters

- **a** (Tensor) – The left-hand tensor.
- **b** (Tensor) – The right-hand tensor.
- **out** (Optional[Tensor]) – Optional result tensor, if not provided it will be created.

Returns The out tensor holding the results of the operation.

Return type *Tensor*

Raises **Error** – if an error happened.

divide(*a*, *b*, *out=None*)

$$c = \frac{a}{b}$$

Divide the a tensor by the b tensor, storing the results into the out tensor. If out is None it will be created with the expected shape of the output. In either case the resulting out tensor is returned.

Parameters

- **a** (Tensor) – The left-hand tensor.
- **b** (Tensor) – The right-hand tensor.
- **out** (Optional[Tensor]) – Optional result tensor, if not provided it will be created.

Returns The out tensor holding the results of the operation.

Return type *Tensor*

Raises **Error** – if an error happened.

abs(*x*, *out=None*)

$$out = |x|$$

Calculate the absolute value of each element of the tensor.

Parameters

- **x** (Tensor) – The input tensor.
- **out** (Optional[Tensor]) – Optional result tensor, if not provided it will be created.

Returns The out tensor holding the results of the operation.

Return type *Tensor*

Raises **Error** – if an error happened.

sqrt(*x*, *out=None*)

$$out = sqrt(x)$$

Calculate the square root of each element of the tensor.

Parameters

- **x** (Tensor) – The input tensor.
- **out** (Optional[Tensor]) – Optional result tensor, if not provided it will be created.

Returns The out tensor holding the results of the operation.

Return type *Tensor*

Raises **Error** – if an error happened.

exp(*x*, *out=None*)

$$out = sqrt(x)$$

Calculate the square root of each element of the tensor.

Parameters

- **x** (Tensor) – The input tensor.
- **out** (Optional[Tensor]) – Optional result tensor, if not provided it will be created.

Returns The out tensor holding the results of the operation.

Return type *Tensor*

Raises **Error** – if an error happened.

log(*x*, *out=None*)

$$out = sqrt(x)$$

Calculate the square root of each element of the tensor.

Parameters

- **x** (Tensor) – The input tensor.
- **out** (Optional[Tensor]) – Optional result tensor, if not provided it will be created.

Returns The out tensor holding the results of the operation.

Return type *Tensor*

Raises **Error** – if an error happened.

matmul(*a*, *b*, *transpose_a=False*, *transpose_b=False*, *out=None*)

$$out = matmul(a, b)$$

Calculate the matrix multiplication of A and B.

Parameters

- **a** (Tensor) – The A tensor.
- **b** (Tensor) – The B tensor.
- **out** (Optional[Tensor]) – Optional result tensor, if not provided it will be created.

Returns The out tensor holding the results of the operation.

Return type *Tensor*

Raises **Error** – if an error happened.

dense(*x*, *w*, *b=None*, *activation=Activation.Linear*, *out=None*)

$$out = dense(x, w, b)$$

Calculate the matrix multiplication of x and w plus bias b.

Parameters

- **x** (Tensor) – The A tensor.
- **w** (Tensor) – The B tensor, transposed.
- **b** (Optional[Tensor]) – The optional bias.
- **activation** (Optional[Activation]) – The optional activation function.
- **out** (Optional[Tensor]) – Optional result tensor, if not provided it will be created.

Returns The out tensor holding the results of the operation.

Return type *Tensor*

Raises **Error** – if an error happened.

linear(*x*, *w*, *b=None*, *activation=Activation.Linear*, *out=None*)

$$out = linear(x, w, b)$$

Calculate the matrix multiplication of x and wT (transposed) plus bias b.

Parameters

- **x** (Tensor) – The A tensor.

- **w** (Tensor) – The B tensor, transposed.
- **b** (Optional[Tensor]) – The optional bias.
- **activation** (Optional[Activation]) – The optional activation function.
- **out** (Optional[Tensor]) – Optional result tensor, if not provided it will be created.

Returns The out tensor holding the results of the operation.

Return type *Tensor*

Raises **Error** – if an error happened.

A.5.3 Convolution

conv(*x*, *k*, *bias=None*, *activation=Activation.Linear*, *cache=None*, *groups=1*, *strides=(1, 1, 1, 1)*, *padding='VALID'*, *dilation=(1, 1, 1, 1)*, *out=None*)

Parameters

- **inp** (Tensor) – The input tensor.
- **out** (Optional[Tensor]) – Optional result tensor, if not provided it will be created.

Returns The out tensor holding the results of the operation.

Return type *Tensor*

Raises **Error** – if an error happened.

A.5.4 Pooling

avgpool(*x*, *window*, *strides=(1, 1, 1, 1)*, *padding='VALID'*, *dilation=(1, 1, 1, 1)*, *out=None*)

Return type *Tensor*

maxpool(*x*, *window*, *strides=(1, 1, 1, 1)*, *padding='VALID'*, *dilation=(1, 1, 1, 1)*, *out=None*)

Return type *Tensor*

reduce_sum(*x*, *axes=None*, *keepdims=False*, *out=None*)

Return type Tensor

reduce_max(*x*, *axes=None*, *keepdims=False*, *out=None*)

Return type Tensor

reduce_min(*x*, *axes=None*, *keepdims=False*, *out=None*)
reduce_max

Return type Tensor

reduce_mean(*x*, *axes=None*, *keepdims=False*, *out=None*)

Return type Tensor

reduce_product(*x*, *axes=None*, *keepdims=False*, *out=None*)

Return type Tensor

A.5.5 Activation

class Activation(*value*)
Bases: enum.Enum

An enumeration.

Linear = 0

Sigmoid = 3

Tanh = 5

ReLU = 1

ReLU6 = 2

sigmoid(*x*, *out=None*)

$$out = \frac{1}{1 + e^{-}x}$$

Calculate the sigmoid of each element of the tensor.

Parameters

- **x** (Tensor) – The input tensor.
- **out** (Optional[Tensor]) – Optional result tensor, if not provided it will be created.

Returns The out tensor holding the results of the operation.

Return type *Tensor*

Raises **Error** – if an error happened.

sigmoid_fast(*x*, *out=None*)

$$out = \frac{1}{1 + |x|}$$

Calculate the approximate sigmoid of each element of the tensor.

Parameters

- **x** (Tensor) – The input tensor.
- **out** (Optional[Tensor]) – Optional result tensor, if not provided it will be created.

Returns The out tensor holding the results of the operation.

Return type *Tensor*

Raises **Error** – if an error happened.

tanh(*x*, *out=None*)

$$out = tanh(x)$$

Calculate the tanh of each element of the tensor.

Parameters

- **x** (Tensor) – The input tensor.
- **out** (Optional[Tensor]) – Optional result tensor, if not provided it will be created.

Returns The out tensor holding the results of the operation.

Return type *Tensor*

Raises **Error** – if an error happened.

relu(*x*, *out=None*)

$$out = max(0, x)$$

Calculate the relu of each element of the tensor.

Parameters

- **x** (Tensor) – The input tensor.
- **out** (Optional[Tensor]) – Optional result tensor, if not provided it will be created.

Returns The out tensor holding the results of the operation.

Return type *Tensor*

Raises **Error** – if an error happened.

relu6(*x*, *out=None*)

$$out = min(max(0, x), 6)$$

Calculate the relu6 of each element of the tensor.

Parameters

- **x** (Tensor) – The input tensor.
- **out** (Optional[Tensor]) – Optional result tensor, if not provided it will be created.

Returns The out tensor holding the results of the operation.

Return type *Tensor*

Raises **Error** – if an error happened.

softmax(*x*, *out=None*)

$$out = softmax(x)$$

Calculate the softmax of the tensor.

Parameters

- **x** (Tensor) – The input tensor.
- **out** (Optional[Tensor]) – Optional result tensor, if not provided it will be created.

Returns The out tensor holding the results of the operation.

Return type *Tensor*

Raises **Error** – if an error happened.

APPENDIX

B

C API REFERENCE

The DeepViewRT C API is modelled around opaque structures, referred to as classes, from which objects are created and interfaced through the various access functions. The internals of the structures are not exposed, instead the access functions are documented and carry the pointer to the opaque object.

B.1 Error Codes

enum **NNError**

Enumeration of all errors provided by DeepViewRT.

Most functions will return an NNError with NN_SUCCESS being zero. A common usage pattern for client code is to check for err using `if (err) ...` as any error condition will return non-zero.

Values:

enumerator **NN_SUCCESS**

Successfull operation, no error.

enumerator **NN_ERROR_INTERNAL**

Internal error without a specific error code, catch-all error.

enumerator **NN_ERROR_INVALID_HANDLE**

The provided handle is invalid.

This error is typically used by *NNEngine* when interfacing with another API such as OpenCL or OpenVX which require native handles for their internal API.

enumerator **NN_ERROR_OUT_OF_MEMORY**

Out of memory error, returned if a call to malloc returns NULL or similar error from an underlying engine plugin.

enumerator **NN_ERROR_OUT_OF_RESOURCES**

Out of resources errors are similar to out of memory though sometimes treated separately by underlying engine plugins.

enumerator **NN_ERROR_NOT_IMPLEMENTED**
Signals an API has not been implemented.

Can be caught by the core DeepViewRT library when interfacing with engine plugins to gracefully fallback to the native implementation.

enumerator **NN_ERROR_INVALID_PARAMETER**
A required parameter was missing or NULL or simply invalid.

enumerator **NN_ERROR_TYPE_MISMATCH**
When attempting to run an operation where the input/output tensors are of different types and the operation does not support automatic type conversions.

enumerator **NN_ERROR_SHAPE_MISMATCH**
When attempting to run an operation and the input/output tensors have invalid or unsupported shape combinations.

Some operations require the shapes to be the same while others, such as arithmetic broadcasting operations, will support various shape combinations but if the provided pairs are invalid then the shape mismatch is returned.

enumerator **NN_ERROR_INVALID_SHAPE**
The tensor's shape is invalid for the given operation.

It differs from the shape mismatch in that the shape is invalid on its own and not relative to another related tensor. An example would be a shape with more than one -1 dimension.

enumerator **NN_ERROR_INVALID_ORDER**
The requested ordering was invalid.

enumerator **NN_ERROR_INVALID_AXIS**
The requested axis for an operation was invalid or unsupported.

enumerator **NN_ERROR_MISSING_RESOURCE**
A required resource was missing or the reference invalid.

enumerator **NN_ERROR_INVALID_ENGINE**
The requested engine is invalid.

enumerator **NN_ERROR_TENSOR_NO_DATA**
The tensor has no data or the data is not currently accessible.

An example of the latter would be attempting to call *nn_tensor_maprw* while the tensor was already mapped read-only or write-only.

enumerator **NN_ERROR_KERNEL_MISSING**
The internal kernel or subroutine required to complete an operation using the engine plugin was missing.

An example would be OpenCL or OpenVX operation where the kernel implementation cannot be located.

enumerator **NN_ERROR_TENSOR_TYPE_UNSUPPORTED**
The operation does not support the tensor's type.

enumerator **NN_ERROR_TOO_MANY_INPUTS**
For operations which can operate on an array of inputs, the provided list of inputs was too large.

enumerator **NN_ERROR_SYSTEM_ERROR**
A system error occured when interfacing with an operating system function.

On some systems errno might be updated with the underlying error code.

enumerator **NN_ERROR_INVALID_LAYER**
When working with a model a reference was made to a layer which did not exist.

enumerator **NN_ERROR_MODEL_INVALID**
The model is invalid or corrupted.

enumerator **NN_ERROR_MODEL_MISSING**
An operation referenced a model but the model was not provided.

enumerator **NN_ERROR_STRING_TOO_LARGE**
The string was too large.

enumerator **NN_ERROR_INVALID_QUANT**
The quantization parameters are invalid.

B.2 Base Functions

const char ***nn_version**()
DeepViewRT library version as "MAJOR.MINOR.PATCH".

Since 2.0

Returns library version string

const char *`nn_strerror`(*NNError* error)
Returns the string associated with a given error.

See also:

NNError

Since 2.0

Parameters

- `error` – The NNError to be represented as a string.

Returns The string representation when the error is valid.

Returns NULL when the error is not valid.

NNError `nn_init`(const NNOptions *options)
Initializes the library with optional parameters.

This function *MUST* be called before any others (though nn_version and nn_strerror are safe) and *MUST* not be called again unless care is taken to protect this call.

Since 2.4

Note: As of DeepViewRT 2.4.32 this function does not do anything except on RaspberryPi platforms. This could change in the future so it is safer to call the function for future compatibility.

Returns NN_SUCCESS after successfully initializing the library.

Returns NN_ERROR_INTERNAL if the library fails to initialize.

B.3 Tensors

struct `NNTensor`
Tensors are represented by the *NNTensor* class.

The dimensions are variable and can be from 1 to 4 dimensions. Internally the shape of a 1-dimensional tensor would be [N 1 1 1] and a scalar [1 1 1 1].

Tensors can exist locally on the CPU or when initialized using an *NNEngine* object the tensors can be mapped to a buffer on a compute device such as a GPU or NPU using the DeepViewRT OpenCL or OpenVX engine plugins.

Public Functions

size_t **nn_tensor_sizeof**()
Returns the size of the tensor object for preparing memory allocations.

Since 2.0

NNTensor ***nn_tensor_init**(void *memory, *NNEngine* *engine)
Initializes the tensor using provided memory.

The memory MUST be at least the size returned by *nn_tensor_sizeof()*. This size does not include the actual tensor data which is allocated separately, either by requesting the implementation to allocate the buffer or attaching to externally allocated memory.

The tensor created by this function has no data associated to it and is of rank-0.

Since 2.0

Parameters

- **memory** – The pointer to be initialized to a *NNTensor* object.
- **engine** – Pointer to the engine object.

Returns NULL given the memory pointer is a null pointer.

Returns Pointer to the newly created *NNTensor* object.

void **nn_tensor_release**(*NNTensor* *tensor)
Releases the memory used by the tensor object.

Since 2.0

Parameters

- **tensor** – Pointer to the tensor object.

NNEngine ***nn_tensor_engine**(*NNTensor* *tensor)
Returns the engine owning this tensor, could be NULL.

Since 2.0

Parameters

- **tensor** – Pointer to the tensor object.

Returns Pointer to the engine object to which the tensor is associated.

NNError **nn_tensor_sync**(*NNTensor* *tensor)
Synchronize the tensor and all preceeding events in the chain.

This is used for engines which may not immediately evaluate tensor operations but instead pass events around, this call will synchronize the event chain leading to this tensor.

Since 2.0

Parameters

- **tensor** – Pointer to the tensor object.

Returns NN_SUCCESS if the sync was successful or ignored by engines which do not implement this API.

int64_t **nn_tensor_time**(*NNTensor* *tensor)
Returns the time information stored in the tensor.

The time is returned in nanoseconds of the duration of the last operation the wrote into this tensor. causes a nn_tensor_sync on the target tensor.

This is used for measuring the time an operation takes by capturing the time the operation took into the destination tensor of the operation. The time is not the time it takes to write to the tensor, this is captured by the *nn_tensor_io_time()* function, but the time it took the operation to complete (not including map/unmap times).

Since 2.0

Parameters

- **tensor** – Pointer to the tensor object.

Returns Nanoseconds of processing time for the last operation which wrote into this tensor.

int64_t **nn_tensor_io_time**(*NNTensor* *tensor)
Returns the I/O time information stored in the tensor.

The time is returned in nanoseconds of the duration of the last map/unmap pair. When tensors are mapped to the CPU (no accelerator engine is loaded) then times are expected to be zero time as no mapping is actually required and the internal pointer is simply returned. When an accelerator engine is used, such as OpenVX, then the io_time measures the time the map/unmap or copy operations took to complete.

Since 2.1

Parameters

- **tensor** – Pointer to the tensor object.

Returns Nanoseconds of time spent in the map/unmap calls.

void **nn_tensor_printf**(*NNTensor* *tensor, bool data, FILE *out)
Writes the tensor inforamtion to the FILE stream provided.

The format is "[D0 D1 D2 D3]" where D0..D3 are the dimensions provided. If the data parameter is true the format will be followed by ": ..." where ... is the string representation of the tensor's data.

Since 2.0

> **Warning:** Before version 2.4.32 this function always assumes float32 tensors and will therefore lead to segmentation faults when used with integer tensors.

Parameters

- **out** – Pointer to the FILE stream where the tensor will be written to.
- **tensor** – Pointer to the tensor object.

NNError **nn_tensor_assign**(*NNTensor* *tensor, *NNTensorType* type, int32_t n_dims, const int32_t shape[NN_ARRAY_PARAM(*n_dims*)], void *data)
Assigns the tensor parameters and optionally data pointer.

The default implementation uses the data buffer as the internal storage for tensor data and it MUST outlive the tensor. For engine plugins they may choose how to use the data but for the OpenCL example if data is provided it will be copied into the OpenCL buffer then otherwise never used again. If NULL is provided for data the OpenCL engine would create the memory and leave it unassigned.

If using the default implementation and leaving data NULL then all operations which require data will fail. The most dynamic tensor setup with optional data would be to call assign to setup the parameters with NULL data, then calling nn_tensor_native_handle to see if one was created, if not the data buffer can be malloc'ed followed by a call to nn_tensor_set_native_handle with this buffer. One could also call nn_tensor_assign a second time with data set to the malloc'ed data.

Since 2.0

Parameters

- **tensor** – Pointer to the given tensor object.
- **type** – The data type that the tensor is storing (The type of the provided data).
- **n_dims** – The number of dimensions in the provided tensor.
- **shape** – The shape of the given tensor.

- **data** – The new tensor data to be placed within the tensor provided.

NNError **nn_tensor_view**(*NNTensor* *tensor, *NNTensorType* type, int32_t n_dims, const int32_t shape[NN_ARRAY_PARAM(*n_dims*)], *NNTensor* *parent, int32_t offset)

Maps the tensor using the memory from the parent tensor.

Since 2.0

Parameters

- **tensor** – Pointer to the tensor object where the view will be stored.
- **type** – The data type that the tensor is storing.
- **n_dims** – The number of dimensions in the provided tensor.
- **shape** – The shape of the given tensor.
- **parent** – Pointer to the tensor object that holds the original tensor.
- **offset** – TO BE DETERMINED.

NNError **nn_tensor_alloc**(*NNTensor* *tensor, *NNTensorType* type, int32_t n_dims, const int32_t shape[NN_ARRAY_PARAM(*n_dims*)])

Allocates the internal memory for the tensor.

Since 2.0

const int32_t *__nn_tensor_shape__(const *NNTensor* *tensor)

Returns the shape of the given tensor object.

Since 2.0

const int32_t *__nn_tensor_strides__(const *NNTensor* *tensor)

Returns the strides of the given tensor object.

Since 2.0

int32_t __nn_tensor_dims__(const *NNTensor* *tensor)

Returns the number of dimensions of the given tensor object.

Since 2.0

const void *__nn_tensor_mapro__(*NNTensor* *tensor)

Maps the tensor's memory and returns the client accessible pointer.

This is the read-only version which causes the engine to download buffers to the CPU memory space if required but will not flush back to the device on unmap.

If the tensor is already mapped read-only or read-write a pointer is returned and the reference count increased, if it was already mapped write-only NULL is returned.

Since 2.0

void *__nn_tensor_maprw__(*NNTensor* *tensor)
Maps the tensor's memory and returns the client accessible pointer.

This is the read-write version which causes the engine to download buffers to the CPU memory space if required and will also flush back to the device on unmap.

If the tensor is already mapped read-only it needs to be unmapped before calling maprw otherwise NULL is returned. A tensor already mapped as rw will simply increase the reference count. A write-only mapped tensor will also return NULL.

Since 2.0

void *__nn_tensor_mapwo__(*NNTensor* *tensor)
Maps the tensor's memory and returns the client accessible pointer.

This is the write-only version which will not cause a download of the buffers to the CPU memory space on map but will upload to the device on unmap.

If the tensor is already mapped write-only or read-write a pointer is returned and the reference count increased. If it was previously mapped as read-only NULL is returned.

Since 2.0

int __nn_tensor_mapped__(const *NNTensor* *tensor)
Returns the tensor's mapping count, 0 means the tensor is unmapped.

Since 2.0

void __nn_tensor_unmap__(*NNTensor* *tensor)
Releases the tensor mapping, if the reference count reaches 0 it will be fully unmapped and will force the flush to the device, if required.

Since 2.0

NNTensorType __nn_tensor_type__(const *NNTensor* *tensor)
Returns the type of a given tensor object.

Since 2.0

NNError **nn_tensor_set_type**(*NNTensor* *tensor, *NNTensorType* type)
Sets the type of a given tensor object.

Since 2.4

size_t **nn_tensor_element_size**(const *NNTensor* *tensor)
Returns the element size of a given tensor object.

Since 2.0

int32_t **nn_tensor_volume**(const *NNTensor* *tensor)
Calculates the total tensor volume (product of dimensions).

Since 2.0

int32_t **nn_tensor_size**(const *NNTensor* *tensor)
Calculates the total byte size of the tensor (volume * element_size).

Since 2.0

char **nn_tensor_axis**(const *NNTensor* *tensor)
Returns the natural data axis of the tensor.

Since 2.4

const int32_t ***nn_tensor_zeros**(const *NNTensor* *tensor, size_t *n_zeros)
Returns the zero-points for the tensor and optionally the number of zero-points.

Since 2.4

void **nn_tensor_set_zeros**(*NNTensor* *tensor, size_t n_zeros, const int32_t *zeros, int own)
Sets the quantization zero-points for the tensor.

If n_zeros>1 it should match the channel dimension (axis) of the tensor.

If own=1 then the tensor will take ownership of the buffer and free it when the tensor is released. Otherwise the buffer must outlive the tensor.

Since 2.4

void **nn_tensor_set_axis**(*NNTensor* *tensor, int32_t axis)
Configures the channel axis of the tensor.

This refers to the "C" in orderings such as NHWC and NCHW.

Since 2.0

const float ***nn_tensor_scales**(const *NNTensor* *tensor, size_t *n_scales)
Returns the scales array for the tensor and optionally the number of scales.

Since 2.4

void **nn_tensor_set_scales**(*NNTensor* *tensor, size_t n_scales, const float *scales, int own)
Sets the quantization scales for the tensor.

If n_scales>1 it should match the channel dimension (axis) of the tensor.

If own=1 then the tensor will take ownership of the buffer and free it when the tensor is released. Otherwise the buffer must outlive the tensor.

Since 2.4

NNQuantizationType **nn_tensor_quantization_type**(*NNTensor* *tensor)
Returns the quantization type for the tensor.

Since 2.4.32

Note: This API was missing before version 2.4.32 and instead the quantization format is inferred as affine when scales and zeros are provided and per-tensor vs. per-channel is inferred based on scales/zeros being 1 or greater than 1.

Parameters

- **tensor** – the tensor object used to query quantization type.

Returns *NNQuantizationType_None* for tensors which do not provide quantization parameters.

Returns *NNQuantizationType_Affine_PerTensor* for tensors which provide quantization parameters which map globally to the tensor.

Returns *NNQuantizationType_Affine_PerChannel* for tensors which provide quantization parameters which map to each channel "C" of the tensor.

Returns *NNQuantizationType_DFP* for tensors which provide DFP parameters. Currently unsupported.

int **nn_tensor_offset**(const *NNTensor* *tensor, int32_t n_dims, const int32_t shape[NN_ARRAY_PARAM(*n_dims*)])
Returns the offset of a given tensor.

This function can be used to calculate the index across numerous dimensions.

Since 2.0

Note: Avoid using this function as part of inner loops as it requires a multiply and add for each dimenions. Instead it can be used in an outer loop to get the starting index then increment this index in the inner loop, possibly using the tensor strides.

Parameters

- **tensor** – the tensor object used in the operation
- **n_dims** – the number of dimensions provided in the shape
- **shape** – the multi-dimensional index used to calculate the linear index

Returns the element index into the tensor based on the muliple dimenional indices provided.

int **nn_tensor_offsetv**(const *NNTensor* *tensor, int32_t n_dims, ...)

Returns the offset of a given tensor using variable length dimensions.

This works the same as *nn_tensor_offset()* but uses variable arguments. The user **must** provide n_dims number of parameters after the n_dims parameter.

Since 2.0

Parameters

- **tensor** – the tensor object used in the operation
- **n_dims** – the number of dimensions to use when calculating the index
- ... – variable number of shape elements which **must** be of type int32_t

Returns the element index into the tensor based on the muliple dimenional indices provided.

int **nn_tensor_compare**(*NNTensor* *left, *NNTensor* *right, double tolerance)

Element-wise comparison of two tensors within a given tolerance, returning total number of errors relative to the left tensor.

If the two tensors are incompatible the volume of the left tensor is returned (all elements invalid).

Deprecated:
2.3

Since 2.0

NNError **nn_tensor_reshape**(*NNTensor* *tensor, int32_t n_dims, const int32_t shape[NN_ARRAY_PARAM(*n_dims*)])

Reshapes the given tensor to the provided new shape.

Since 2.0

Parameters

- **tensor** – the tensor object used in the operation
- **n_dims** – the number of dimensions which the tensor will contain after the operation completes successfully. It must also match the number of elements in `shape`.
- **shape** – the new shape for the tensor. The array must be at least `n_dims` elements in size.

Returns *NN_SUCCESS* if the reshape is able to be performed

Returns *NN_ERROR_SHAPE_MISMATCH* if the new shape cannot be represented given the previous shape of the tensor.

NNError **nn_tensor_shuffle**(*NNTensor* *output, *NNTensor* *input, int32_t n_dims, const int32_t order[NN_ARRAY_PARAM(*n_dims*)])

Shuffles (transpose) the tensor moving the current dimensions into the ordering defined in the order parameter.

For example a traditional matrix transpose is done using order[] = { 1, 0 } in other words, the 0 dimension of the output references the 1 dimension of the input and the 1 dimension of the output references the 0 dimension of the input.

Another example would be shuffling an NCHW tensor to NHWC using order[] = { 0, 2, 3, 1 }

Since 2.0

NNError **nn_tensor_fill**(*NNTensor* *tensor, double constant)

Fills the tensor with the provided constant.

The constant is captured as double precision (64-bit floating point) which has 53-bits of precision on whole numbers. This means the constant CANNOT represent all 64-bit integers but it CAN represent all 32-bit and lower integers. If full 64-bit integer support is required nn_tensor_map can be used though it is less efficient with some engines because of the addition memory transfer required.

The double will be cast appropriately to the target tensor's type before filling the tensor.

Since 2.0

NNError **nn_tensor_randomize**(*NNTensor* *tensor)
Randomizes the data within the tensor.

Deprecated:
2.3

Since 2.0

NNError **nn_tensor_copy**(*NNTensor* *dest, *NNTensor* *source)
Copies the contents of source tensor into destination tensor.

This operation only copies the data and does not affect the destination tensor's properties. The destination tensor must have an equal or larger volume. If required data will be converted.

Since 2.0

NNError **nn_tensor_copy_buffer**(*NNTensor* *tensor, const void *buffer, size_t bufsize)
Loads a tensor with data from a user buffer User has to maintain the buffer and ensure compatibility with NHWC tensor Function will return error if there is a size mismatch i.e (bufsize != nn_tensor_size(tensor)) or tensor is invalid.

Since 2.4

NNError **nn_tensor_requantize**(*NNTensor* *dest, *NNTensor* *source)
Requantizes the source tensor into the destination tensor.

The source tensor and destination tensor should be either I8 or U8, and per tensor quantized.

Since 2.4

NNError **nn_tensor_quantize**(*NNTensor* *dest, *NNTensor* *source, int axis)
Quantizes the source tensor into the destination tensor.

The source tensor should be float and the destination integer. If the destination tensor does not have quantization parameters they will be calculated from the source tensor and stored into the destination tensor.

When calculating the quantization parameters if axis is a valid axis* then per-channel quantization will be performed along the axis, otherwise per-tensor quantization will be performed. If the destination tensor has quantization parameters axis is ignored.

Valid Axis: (axis > 0 && axis < n_dims)

Since 2.4

NNError **nn_tensor_quantize_buffer**(*NNTensor* *dest, size_t buffer_length, const float *buffer, int axis)

Quantizes the source buffer into the destination tensor.

The source tensor should be float and the destination integer. If the destination tensor does not have quantization parameters they will be calculated from the source buffer and stored into the destination tensor.

When calculating the quantization parameters if axis is a valid axis* then per-channel quantization will be performed along the axis, otherwise per-tensor quantization will be performed. If the destination tensor has quantization parameters axis is ignored.

Valid Axis: (axis > 0 && axis < n_dims)

Since 2.4

NNError **nn_tensor_dequantize**(*NNTensor* *dest, *NNTensor* *source)

De-quantizes the source tensor into the destination tensor.

The source tensor should be integer and the destination float. The source tensor must have quantization parameters otherwise the operation will simply cast the integer data to float.

Since 2.4

NNError **nn_tensor_dequantize_buffer**(*NNTensor* *source, size_t buffer_length, float *buffer)

De-quantizes the source tensor into the destination buffer.

The source tensor should be integer and the destination float. The source tensor must have quantization parameters otherwise the operation will simply cast the integer data to float.

The buffer must be at least buffer_length*sizeof(float) size in bytes.

Since 2.4

NNError **nn_tensor_concat**(*NNTensor* *output, int32_t n_inputs, *NNTensor* *inputs[NN_ARRAY_PARAM(*n_inputs*)], int32_t axis)

nn_tensor_concat concatenates all of the given input tensors into the given output tensor.

pointer to the output tensor list of pointers to the input tensors the number of inputs the axis along which to concatenate the inputs

Since 2.0

NNError **nn_tensor_slice**(*NNTensor* *output, *NNTensor* *input, int32_t n_axes, const int32_t axes[NN_ARRAY_PARAM(*n_axes*)], const int32_t head[NN_ARRAY_PARAM(*n_axes*)], const int32_t tail[NN_ARRAY_PARAM(*n_axes*)])

nn_tensor_slice copies a slice of the tensor into output.

The axes, head, and tail must be of length n_axes or NULL. Calling slice with axes==NULL will ignore head/tail and is effectively *nn_tensor_copy*.

When head is NULL all axes are assumed to start at 0. When tail is NULL all axes are assumed to end at (len(axis) - head) for the given axis.

Since 2.0

NNError **nn_tensor_padding**(*NNTensor* *tensor, const char *padtype, const int32_t *window, const int32_t *stride, const int32_t *dilation, int32_t *padded_shape, int32_t *paddings)

nn_tensor_padding calculates the paddings for the given tensor, padtype, window, stride, and dilation given n_dims being queried from the tensor's *nn_tensor_dims()*.

The paddings pointer must point to an array of 2 * n_dims elements into which the function will write the head/tail padding tuples for each of the n_dims provided dimensions. The padded_shape parameter must point to an array of n_dims elemens which will receive the output (padded) shape.

The padtype can be "VALID" or "SAME". When padtype is "SAME" padded_shape will equal the shape of the input tensor and the paddings will be provided to achieve this shape. When padtype is "VALID" then paddings will be all zeros and the padded_shape will provide the target output shape given the provided parameters.

Since 2.3

NNError **nn_tensor_pad**(*NNTensor* *output, *NNTensor* *input, const int32_t head[4], const int32_t tail[4], double constant)

nn_tensor_pad implements a padded Tensor to Tensor copy.

This can be used to achieve the various convolution padding strategies (SAME, FULL). For example SAME conv2d would use the following padded_copy before running the conv2d layer.

output_shape = { input_shape[0], int(ceil(float(input_shape[1]) / strides[1])), int(ceil(float(input_shape[2]) / strides[2])), weights_shape[3] };

pad_height = (output_shape[1] - 1) * strides[1] + weights_shape[0] - input_shape[1];
pad_width = (output_shape[2] - 1) * strides[2] + weights_shape[1] - input_shape[2];

pointer to the output tensor pointer to the input tensor lead-in length of the pad for dimension NHWC lead-out length of the pad for dimension NHWC

Since 2.0

NNError **nn_tensor_load_file**(*NNTensor* *tensor, const char *filename)
Loads an image from file into the provided tensor.

Deprecated:
2.3

Since 2.2

NNError **nn_tensor_load_file_ex**(*NNTensor* *tensor, const char *filename, uint32_t proc)
Loads an image from file into the provided tensor.

Deprecated:
2.3

Since 2.2

NNError **nn_tensor_load_image**(*NNTensor* *tensor, const void *image, size_t image_size)
Loads an image from the provided buffer and decodes it accordingly, the function uses the images headers to find an appropriate decoder.

The function will handle any required casting to the target tensor's format.

Since 2.0

NNError **nn_tensor_load_image_ex**(*NNTensor* *tensor, const void *image, size_t image_size, uint32_t proc)
Loads an image from the provided buffer and decodes it accordingly, the function uses the images headers to find an appropriate decoder.

The function will handle any required casting to the target tensor's format and will apply image standardization (compatible with tensorflow's tf.image.per_image_standardization) if the proc parameter is set to NN_IMAGE_PROC_WHITENING.

When called with proc==0 it is the same as *nn_tensor_load_image()*.

NN_IMAGE_PROC_UNSIGNED_NORM NN_IMAGE_PROC_WHITENING_NORM NN_IMAGE_PROC_SIGNED_NORM

Since 2.1

B.3.1 Tensor Type

Tensors can have various internal data types as defined by @ref nn_tensor_type which must resolve to one of the following values.

enum **NNTensorType**
Enumeration of the data types supported by NNTensors in DeepViewRT.

Values:

enumerator **NNTensorType_RAW**
Raw byte-stream tensor, useful for encoded tensors such as PNG images.

The size of this tensor would be in bytes.

enumerator **NNTensorType_STR**
String tensor data, a single dimension would hold one null-terminated string of variable length.

A standard C char* array.

enumerator **NNTensorType_I8**
Signed 8-bit integer tensor data internally int8_t.

enumerator **NNTensorType_U8**
Unsigned 8-bit integer tensor data internally uint8_t.

enumerator **NNTensorType_I16**
Signed 16-bit integer tensor data internally int16_t.

enumerator **NNTensorType_U16**
Unsigned 16-bit integer tensor data internally uint16_t.

enumerator **NNTensorType_I32**
Signed 16-bit integer tensor data internally int32_t.

enumerator **NNTensorType_U32**
Unsigned 16-bit integer tensor data internally uint32_t.

enumerator **NNTensorType_I64**
Signed 16-bit integer tensor data internally int64_t.

enumerator **NNTensorType_U64**
Unsigned 16-bit integer tensor data internally uint64_t.

enumerator **NNTensorType_F16**
Half precision (16-bit) floating point tensor data.

enumerator **NNTensorType_F32**
Single precision (32-bit) floating point tensor data.

enumerator **NNTensorType_F64**
Double precision (64-bit) floating point tensor data.

B.3.2 Quantization Type

Tensors can support quantization using one of the following methods which may be queried using `nn_tensor_quantization_type()`.

enum **NNQuantizationType**
Enumeration of all quantization type provided by DeepViewRT.

Values:

enumerator **NNQuantizationType_None**
No quantization for tensor.

enumerator **NNQuantizationType_Affine_PerTensor**
Affine quantization with parameters applied globally across the tensor.

The scale term is queried from *nn_tensor_scales()* while the zero term is queried from *nn_tensor_zeros()*.

Quantization: $f(x) = \frac{x}{scale} + zero$

Dequantization: $f(x) = (x - zero) * scale$

enumerator **NNQuantizationType_Affine_PerChannel**
Affine quantization with separate parameters applied to each channel.

Also known as per-axis where the axis is always the channel "C" axis in a NCHW, NHWC, and so-on shaped tensor.

Same equation as NNQuantization_Affine_PerTensor but applied per-channel. The scale and zero_point are vectors of channel length.

enumerator **NNQuantizationType_DFP**
Quantized using Dynamic Fixed Point.

B.4 Context

struct **NNContext**

DeepViewRT models can be loaded with an *NNContext* and numerous contexts can be loaded at once.

The context manages the runtime portion of the model including the tensors required to hold intermediate buffers.

A context itself requires NN_CONTEXT_SIZEOF bytes though it will also allocate on the heap additional tensor handles required to support models on *nn_context_model_load()* and these will then be released on a call to *nn_context_model_unload()*.

When a context is created an *NNEngine* plugin may optionally be provided which will take over the management of tensors through the engine plugin and attempting to run models and operators on the compute device enabled by this engine plugin. If an engine is not provided DeepViewRT will use the default implementation which is optimized for CPU and MCU devices.

Public Types

NNError() nn_user_ops (NNContext *context, const char *opname, size_t index)

Callback function for custom user ops.

Since 2.4

Public Functions

size_t **nn_context_sizeof**()

Returns the actual size of the context structure.

This size will be smaller than NN_CONTEXT_SIZEOF which contains additional padding for future extension. Since *nn_context_sizeof()* is called dynamically at runtime it can return the true and unpadded size.

Since 2.0

NNContext ***nn_context_init**(*NNEngine* *engine, size_t memory_size, void *memory, size_t cache_size, void *cache)

Initializes an *NNContext* and allocates required memories.

If any of the pointers are NULL malloc will be called automatically to create the memory using the provided sizes. For memory_size and cache_size if these are 0 then they will not be initialized.

Since 2.0

NNContext ***nn_context_init_ex**(void *context_memory, *NNEngine* *engine, size_t memory_size, void *memory, size_t cache_size, void *cache)

Initializes an *NNContext* into the provided memory which *MUST* be at least NN_CONTEXT_SIZEOF bytes.

If any of the pointers are NULL malloc will be called automatically to create the memory using the provided sizes. For memory_size and cache_size if these are 0 then they will not be initialized.

Since 2.0

void **nn_context_release**(*NNContext* *context)

Release the memory being used by the given context object.

Since 2.0

NNError **nn_context_user_ops_register**(*NNContext* *context, nn_user_ops *callback)

Since 2.4

nn_user_ops ***nn_context_user_ops**(*NNContext* *context)

Since 2.4

NNTensor ***nn_context_cache**(*NNContext* *context)

Since 2.2

NNTensor ***nn_context_mempool**(*NNContext* *context)

Since 2.2

NNEngine ***nn_context_engine**(*NNContext* *context)

Returns the engine used by the given context object.

Since 2.0

const *NNModel* ***nn_context_model**(*NNContext* *context)

Returns the currently loaded model blob for the context.

Since 2.0

NNError **nn_context_model_load**(*NNContext* *context, size_t memory_size, const void *memory)

Loads the model provided by the input into the context.

pointer to the context object pointer to the memory that contains the model the size of the memory that is used by the model

Since 2.0

void **nn_context_model_unload**(*NNContext* *context)

Frees the memory used by the model within the given context object.

Since 2.0

NNTensor ***nn_context_tensor**(*NNContext* *context, const char *name)

Returns the tensor with the given name within the model provided by the given context object.

Since 2.0

NNTensor ***nn_context_tensor_index**(*NNContext* *context, size_t index)

Returns the tensor at the given index with the model provided by the given context object.

Since 2.0

NNError **nn_context_run**(*NNContext* *context)

Runs the model within the given context object.

Since 2.0

NNError **nn_context_step**(*NNContext* *context, size_t index)

Runs layer with index from model within the given context object.

If index is invalid NN_ERROR_INVALID_LAYER is returned, this can be used to determine when at the end of the model.

Since 2.3

B.5 Model

struct **NNModel**

DeepViewRT Models "RTM" are reprensted in memory through the *NNModel* type which is meant to point to a static model blob.

This can point directly to the memory of the RTM either loaded into memory, accessed through a memmap or pointed directly to the flash location. In other words if the RTM is saved into flash which is connected to the memory space then the model does not need to be copied into RAM before being loaded.

Models are loaded into an *NNContext* which handles the dynamic data structures required for operation of the model.

Public Functions

int **nn_model_validate**(const *NNModel* *memory, size_t size)

Attempts to validate model, this is automatically called by nn_model_load and nn_model_mmap.

The function returns 0 on success, otherwise it will return an error code which can be turned into a string by calling *nn_model_validate_error()* with the return value from *nn_model_validate()*.

Since 2.0

const char ***nn_model_validate_error**(int err)

Returns the string associated with a given error returned from *nn_model_validate()*.

Since 2.0

const char ***nn_model_name**(const *NNModel* *model)

Returns the name of the given model object.

Names are optional and if the model does not contain a name then NULL will be returned.

Since 2.0

const char ***nn_model_uuid**(const *NNModel* *model)

Currently returns NULL (UPDATE WHEN FUNCTION IS UPDATED)

Since 2.0

uint32_t **nn_model_serial**(const *NNModel* *model)

Currently returns 0.

Since 2.0

int **nn_model_label_count**(const *NNModel* *model)
Returns the number of labels within a given model object.

Since 2.0

const char ***nn_model_label**(const *NNModel* *model, int index)
Returns the label of the given index within the given model object.

If the model contains no labels or the index is out of range then NULL will be returned.

Since 2.0

const uint8_t ***nn_model_label_icon**(const *NNModel* *model, int index, size_t *size)
Returns an optional icon resource for the provided label index.

Since 2.0

const uint32_t ***nn_model_inputs**(const *NNModel* *model, size_t *n_inputs)
Returns the list of model input indices and optionally the number of inputs.

If the field is missing from the model NULL is returned.

Since 2.4

const uint32_t ***nn_model_outputs**(const *NNModel* *model, size_t *n_outputs)
Returns the list of model output indices and optionally the number of outputs.

If the field is missing from the model 0 is returned.

Since 2.4

size_t **nn_model_layer_count**(const *NNModel* *model)
Returns the number of layers within a given model object.

Since 2.0

const char ***nn_model_layer_name**(const *NNModel* *model, size_t index)
Returns the name of a layer at a given index within the given model object.

Since 2.0

int **nn_model_layer_lookup**(const *NNModel* *model, const char *name)
Returns the index of a given layer with the name provided in the given model object.

Since 2.0

const char ***nn_model_layer_type**(const *NNModel* *model, size_t index)
Returns the type of a layer at the given index within the given model object.

Since 2.0

int16_t **nn_model_layer_type_id**(const *NNModel* *model, size_t index)
Returns the type ID of the layer.

Since 2.4

const char ***nn_model_layer_datatype**(const *NNModel* *model, size_t index)
Returns the datatype of a layer at the given index within the given model object.

Since 2.0

NNTensorType **nn_model_layer_datatype_id**(const *NNModel* *model, size_t index)
Returns the datatype of a layer at the given index within the given model object.

Since 2.0

const int32_t ***nn_model_layer_zeros**(const *NNModel* *model, size_t index, size_t *n_zeros)
Returns the array of quantization zero-points, and optionally the number of zero-points in the array.

The length will either be 0, 1, or equal to the number of channels in an NHWC/NCHW tensor.

The channel axis can be queried using *nn_model_layer_axis()*.

If no quantization parameters are available then n_zeros will be 0. If the tensor is quantized using full tensor quantization n_zeros will be 1. If the tensor is quantized using per-channel quantization n_zeros will be C which will equal the channel dimension of the tensor. For an NHWC tensor it would equal shape[3].

Since 2.4

const float ***nn_model_layer_scales**(const *NNModel* *model, size_t index, size_t *n_scales)
Returns the array of quantization scales, and optionally the number of scales in the array.

The length will either be 0, 1, or equal to the number of channels in an NHWC/NCHW tensor.

The channel axis can be queried using *nn_model_layer_axis()*.

If no quantization parameters are available then n_scales will be 0. If the tensor is quantized using full tensor quantization n_scales will be 1. If the tensor is quantized using per-channel quantization n_scales will be C which will equal the channel dimension of the tensor. For an NHWC tensor it would equal shape[3].

Since 2.4

int **nn_model_layer_axis**(const *NNModel* *model, size_t index)
Returns the natural data axis for the tensor or -1 if one is not set.

Since 2.4

const int32_t ***nn_model_layer_shape**(const *NNModel* *model, size_t index, size_t *n_dims)
Returns the shape of a layer at the given index within the given model object.

Since 2.0

size_t **nn_model_layer_inputs**(const *NNModel* *model, size_t index, const uint32_t **inputs)
Returns the number of inputs to a layer at the given index within the given model object.

Since 2.0

const *NNModelParameter* ***nn_model_layer_parameter**(const *NNModel* *model, size_t layer, const char *key)
Returns an *NNModelParameter* from the model at the layer index defined by layer using the parameter key.

If the layer does not contain this parameter NULL is returned.

Since 2.4

const int32_t ***nn_model_layer_parameter_shape**(const *NNModel* *model, size_t layer, const char *key, size_t *n_dims)
Returns the shape of the model parameter for layer at index <layer>.

nn_model_parameter_shape()

Returns NULL if either the parameter is not found or the shape is missing.

Since 2.4

const float ***nn_model_layer_parameter_data_f32**(const *NNModel* *model, size_t layer, const char *key, size_t *length)
Returns float data for parameter <key> at layer index <layer>.

This is a convenience wrapper around acquiring the parameter followed by acquiring the data.

nn_model_parameter_data_f32()

Returns NULL if either the parameter is not found or the data is missing.

Since 2.4

const int16_t ***nn_model_layer_parameter_data_i16**(const *NNModel* *model, size_t layer, const char *key, size_t *length)

Returns int16 data for parameter <key> at layer index <layer>.

This is a convenience wrapper around acquiring the parameter followed by acquiring the data.

nn_model_parameter_data_i16()

Returns NULL if either the parameter is not found or the data is missing.

Since 2.4

const uint8_t ***nn_model_layer_parameter_data_raw**(const *NNModel* *model, size_t layer, const char *key, size_t *length)

Returns raw data for parameter <key> at layer index <layer>.

This is a convenience wrapper around acquiring the parameter followed by acquiring the data.

nn_model_parameter_data_raw()

Returns NULL if either the parameter is not found or the data is missing.

Since 2.4

const char ***nn_model_layer_parameter_data_str**(const *NNModel* *model, size_t layer, const char *key, size_t index)

Returns string data for parameter <key> at layer index <layer> for string array element <index>.

This is a convenience wrapper around acquiring the parameter followed by acquiring the data.

nn_model_parameter_data_str()

Returns NULL if either the parameter is not found or the data is missing.

Since 2.4

size_t **nn_model_layer_parameter_data_str_len**(const *NNModel* *model, size_t layer, const char *key)

Returns number of string elements in the data_str array for the specified layer and parameter key.

This is a convenience wrapper around acquiring the parameter followed by acquiring the data.

nn_model_parameter_data_str_len()

Returns number of string elements in the array.

Since 2.4

size_t **nn_model_memory_size**(const *NNModel* *model)

Returns the memory size of the given model object.

Since 2.0

size_t **nn_model_cache_minimum_size**(const *NNModel* *model)

Returns the minimum cache size of a given model object.

Since 2.0

size_t **nn_model_cache_optimum_size**(const *NNModel* *model)

Returns the optimum cache size of a given model object.

Since 2.0

size_t **nn_model_resource_count**(const *NNModel* *model)

The number of resources defined in the model.

Since 2.4

Parameters

- **model** – pointer to the RTM model

Returns number of resources defined in the model.

const *NNModelResource* ***nn_model_resource_at**(const *NNModel* *model, size_t index)

Retrieves a reference to the resource at the provided index.

Since 2.4

Parameters

- **model** – pointer to the RTM model
- **index** – resource index

Returns an *NNModelResource* pointer for the provided `index` in the given model.

Returns NULL if either the model or index are invalid.

const *NNModelResource* ***nn_model_resource**(const *NNModel* *model, const char *name)
Retrieves a reference to the resource with the given name.

Since 2.4

Parameters

- **model** – pointer to the RTM model
- **name** – the unique name of the resource

Returns an *NNModelResource* pointer for the provided unique `name`.

Returns NULL if either the `model` or `name` are invalid, NULL, or the `name` is not found.

B.5.1 Model Resource

struct **NNModelResource**
DeepViewRT Models may have resources embedded into them and this datatype is their handle.

Public Functions

const char ***nn_model_resource_name**(const *NNModelResource* *resource)
The unique name of the resource as can be used to retrieve the resource using nn_model_resource().

Since 2.4

Parameters

- **resource** – pointer to a *NNModelResource* retrieved from the model.

Returns A string with the name of the resource.

Returns NULL if the resource or name is NULL.

const char *__nn_model_resource_meta__(const *NNModelResource* *resource)
Returns the meta string for the resource.

Since 2.4

Parameters

- **resource** – pointer to a *NNModelResource* retrieved from the model.

Returns A string with the meta parameter of the resource.

Returns NULL if the resource or meta are NULL.

const char *__nn_model_resource_mime__(const *NNModelResource* *resource)
Returns the mime type string for the resource.

Since 2.4

Parameters

- **resource** – pointer to a *NNModelResource* retrieved from the model.

Returns A string with the mime parameter of the resource.

Returns NULL if the resource or mime are NULL.

const uint8_t *__nn_model_resource_data__(const *NNModelResource* *resource, size_t *data_size)
Returns the raw binary data for the resource, the size of the data will be saved in `data_size` if non-NULL.

Since 2.4

Parameters

- **resource** – pointer to a *NNModelResource* retrieved from the model.
- **data_size** – optional pointer to a size_t to receive the length in bytes of the data, if provided.

Returns pointer to the start of the data stream of length `data_size`.

Returns NULL if resource has no data associated.

B.5.2 Model Parameter

struct **NNModelParameter**

DeepViewRT Models use parameters to store various configuration information such as layer parameters.

Public Functions

const int32_t ***nn_model_parameter_shape**(const *NNModelParameter* *parameter, size_t *n_dims)

Returns the shape of the parameter data or NULL if no shape was defined.

If n_dims is non-NULL the number of dimensions will be stored there. The shape attribute is not required for parameters but can be used either on its own or as part of defining layout of data attributes.

Since 2.4

const float ***nn_model_parameter_data_f32**(const *NNModelParameter* *parameter, size_t *length)

Returns parameter float data, length of the array is optionally stored into the length parameter if non-NULL.

If parameter does not have this data type, then NULL is returned.

Since 2.4

const int32_t ***nn_model_parameter_data_i32**(const *NNModelParameter* *parameter, size_t *length)

Returns parameter int32_t data, length of the array is optionally stored into the length parameter if non-NULL.

If parameter does not have this data type, then NULL is returned.

Since 2.4

const int16_t ***nn_model_parameter_data_i16**(const *NNModelParameter* *parameter, size_t *length)

Returns parameter int16_t data, length of the array is optionally stored into the length parameter if non-NULL.

If parameter does not have this data type, then NULL is returned.

Since 2.4

const int8_t *`nn_model_parameter_data_i8`(const *NNModelParameter* *parameter, size_t *length)

Returns parameter int8_t data, length of the array is optionally stored into the length parameter if non-NULL.

If parameter does not have this data type, then NULL is returned.

Since 2.4

const uint8_t *`nn_model_parameter_data_raw`(const *NNModelParameter* *parameter, size_t *length)

Returns parameter raw data pointer, length of the array is optionally stored into the length parameter if non-NULL.

If parameter does not have this data type, then NULL is returned.

Since 2.4

const char *`nn_model_parameter_data_str`(const *NNModelParameter* *parameter, size_t index)

Returns parameter string data at desired index.

This data handler is different from the others which return the array as strings are themselves arrays and need special handling. Refer to *nn_model_parameter_data_str_len()* to query the size of the data_str array, which refers to the number of strings in this parameter.

Since 2.4

size_t `nn_model_parameter_data_str_len`(const *NNModelParameter* *parameter)

Returns the number of strings in the parameter's data_str attribute.

Since 2.4

B.6 Engine

struct `NNEngine`

Engine structure provides the means to implement custom tensor and kernel implementations which implement the DeepViewRT inference backend.

As an example the OpenCL backend is provided as a plugin which exposes an *NNEngine* which maps NNTensors to cl_mem objects and kernels as OpenCL kernels.

Public Functions

size_t **nn_engine_sizeof**()
The actual size of the *NNEngine* structure.

This will differ from the size defined by NN_ENGINE_SIZEOF as the later is padded for future API extensions while this function returns the actual size currently required.

Since 2.0

Returns *NNEngine* structure size as reported by sizeof().

NNEngine ***nn_engine_init**(void *memory)
Initializes the *NNEngine* structure using the provided memory or allocating a new buffer is none was provided.

When providing memory it must be at least the size returned by *nn_engine_sizeof()* and for statically initiallized arrays the NN_ENGINE_SIZEOF can be used instead which is padded for future API extensions.

Since 2.0

Note: previous to version 2.4.32 the memory parameter is required otherwise NULL will always be returned and no engine structure is created.

Parameters

- **memory** – Pointer to the start of where a *NNEngine* object should be initialized.

Returns Pointer to the initialized *NNEngine* structure.

Returns NULL if memory was NULL and malloc using nn_engine_size() returns NULL.

void ***nn_engine_native_handle**(*NNEngine* *engine)
Returns handle of the *NNEngine* object.

Since 2.0

Parameters

- **memory** – Pointer to the *NNEngine* structure

void **nn_engine_release**(*NNEngine* *engine)
Releases the memory that was being used by the engine.

Since 2.0

Parameters

- **engine** – Pointer to the engine object.

NNError **nn_engine_load**(*NNEngine* *engine, const char *plugin)
Loads the plugin to provided engine object.

The plugin should point to an engine plugin library either as an absolute or relative path or be found in the standard OS search path for shared libraries.

Since 2.0

Parameters

- **engine** – Pointer to the engine object.
- **plugin** – String of the absolute or relative path to the plugin.

Returns NN_ERROR_INVALID ENGINE given the engine pointer is NULL or the plugin does not have the necessary functions.

Returns NN_ERROR_MISSING_RESOURCE given the plugin dll cannot be found.

Returns The error returned by the plugin's init function given a valid engine and dll.

void **nn_engine_unload**(*NNEngine* *engine)
Unloads the plugin from the given engine object.

Since 2.0

Parameters

- **engine** – Pointer to the engine object.

Returns NN_ERROR_INVALID_ENGINE given the engine pointer is NULL.

Returns NN_ERROR_INTERNAL if the plugin dll could not be closed properly.

Returns The NNError from the plugin's cleanup function.

const char ***nn_engine_name**(*NNEngine* *engine)
Returns the name of the engine object.

Since 2.0

Parameters

- **engine** – Pointer to the engine object.

const char *__nn_engine_version__(*NNEngine* *engine)
Returns the version of the engine object.

Since 2.0

Parameters

- **engine** – Pointer to the engine object.

B.7 Definitions

B.7.1 Versions

NN_TARGET_VERSION
This macro defines the target version when compiling against deepview_rt.h and deepview_ops.h and will cause warnings to be generated when the target version does not include a symbol or if a symbol is marked as depracated.

By default the target version is the latest version of DeepViewRT for the provided deepview_rt.h file.

APPENDIX

C

OPERATIONS REFERENCE

This chapter outlines the supported operations in DeepViewRT. Some of the operations support "fused activations" where an optional activation parameter is provided which causes the implementation to apply the activation function before writting out the results to the output tensor.

enum **NNActivation**
Activation function to apply, if desired.

Values:

enumerator **NNActivation_Linear**
Linear activation results in a no-op.

enumerator **NNActivation_ReLU**
ReLU activation performs $f(x) = max(0, x)$.

enumerator **NNActivation_ReLU6**
ReLU6 activation performs $f(x) = min(max(0, x), 6)$.

enumerator **NNActivation_Sigmoid**
Sigmoid activation performs $f(x) = 1/(1 + e^{-}x)$.

enumerator **NNActivation_SigmoidFast**

enumerator **NNActivation_Tanh**
Tanh activation performs $f(x) = tanh(x)$.

Functions

NNError **nn_add**(*NNTensor* *output, *NNTensor* *input_a, *NNTensor* *input_b)
nn_add implements addition for two tensors, a + b = c.

This function follows standard broadcasting rules for tensors. The output shape of the result is equal to the largest tensor a or b.

Since 2.0

Parameters

- **output** – pointer to the output tensor
- **input_a** – pointer to the first input tensor
- **input_b** – pointer to the second input tensor

Returns NNError

NNError **nn_subtract**(*NNTensor* *output, *NNTensor* *input_a, *NNTensor* *input_b)

nn_subtract implements subtraction for two tensors, a - b = c.

This function follows standard broadcasting rules for tensors. The output shape of the result is equal to the shape of input_a.

Since 2.0

Parameters

- **output** – pointer to the output tensor
- **input_a** – pointer to the tensor being subtracted from
- **input_b** – pointer to the tensor to subtract from input_a

Returns NNError

NNError **nn_multiply**(*NNTensor* *output, *NNTensor* *input_a, *NNTensor* *input_b)

nn_multiply implements addition for two tensors, a * b = c.

This function follows standard broadcasting rules for tensors. The output shape of the result is equal to the largest tensor a or b.

Since 2.0

Parameters

- **output** – pointer to the output tensor
- **input_a** – pointer to the first input tensor
- **input_b** – pointer to the second input tensor

Returns NNError

NNError **nn_divide**(*NNTensor* *output, *NNTensor* *input_a, *NNTensor* *input_b)

nn_divide implements addition for two tensors, a / b = c.

This function follows standard broadcasting rules for tensors. The output shape of the result is equal the shape of input_b.

Since 2.0

Parameters

- **output** – pointer to the output tensor
- **input_a** – pointer to the tensor representing the dividend
- **input_b** – pointer to the tensor representing the divisor

Returns NNError

NNError **nn_abs**(*NNTensor* *output, *NNTensor* *input)
nn_abs implements the absolute value function, |input| = output.

Since 2.0

Parameters

- **output** – pointer to the output tensor
- **input** – point to the input tensor

Returns NNError

NNError **nn_sqrt**(*NNTensor* *output, *NNTensor* *input)
nn_sqrt implements the square root function, input ^ 0.5 = output.

Since 2.0

Parameters

- **output** – pointer to the output tensor
- **input** – pointer to the input tensor

Returns NNError

NNError **nn_rsqrt**(*NNTensor* *output, *NNTensor* *input, const float *epsilon)
nn_rsqrt implements the inverse square root function, 1 / (input ^ 0.5) = output.

Since 2.0

Parameters

- **output** – pointer to the output tensor
- **input** – pointer to the input tensor
- **epsilon** – epsilon parameter is applied to the input to avoid divide by zero.

Returns NNError

NNError **nn_log**(*NNTensor* *output, *NNTensor* *input)
Implements the log operation.

Since 2.4

Parameters

- **output** – pointer to the output tensor
- **input** – pointer to the input tensor

Returns NNError

NNError **nn_exp**(*NNTensor* *output, *NNTensor* *input)
Implements the exp operation.

Since 2.4

Parameters

- **output** – pointer to the output tensor
- **input** – pointer to the input tensor

Returns NNError

NNError **nn_sigmoid_fast**(*NNTensor* *output, *NNTensor* *input)
A fast approximation of the logistic sigmoid implemented as 1/(1+|x|).

Since 2.0

Parameters

- **output** – pointer to the output tensor
- **input** – pointer to the input tensor

Returns NNError

NNError **nn_sigmoid**(*NNTensor* *output, *NNTensor* *input)
The logistic sigmoid 1/(1+e^-x) or e^x/(e^x+1).

Since 2.0

Parameters

- **output** – pointer to the output tensor

- **input** – pointer to the input tensor

Returns NNError

NNError **nn_tanh**(*NNTensor* *output, *NNTensor* *input)

nn_tanh implements the hyperbolic tangent function.

Since 2.0

Parameters

- **output** – pointer to the output tensor
- **input** – pointer to the input tensor

Returns NNError

NNError **nn_relu**(*NNTensor* *output, *NNTensor* *input)

nn_relu implements the rectified linear function.

The formula used for a given input tensor is, max(input, 0) = output. The output shape is equal to the shape of the input.

Since 2.0

Parameters

- **output** – pointer to the output tensor
- **input** – pointer to the input tensor

Returns NNError

NNError **nn_relu6**(*NNTensor* *output, *NNTensor* *input)

nn_relu implements the rectified linear 6 function.

The formula used for a given input tensor is, min(max(input, 0), 6) = output. The output shape is equal to the shape of the input.

Since 2.0

Parameters

- **output** – pointer to the output tensor
- **input** – pointer to the input tensor

Returns NNError

NNError **nn_swish**(*NNTensor* *output, *NNTensor* *input)
Hard-Swish activation function.

Since 2.4

Parameters

- **output** –
- **input** –

Returns NNError

NNError **nn_prelu**(*NNTensor* *output, *NNTensor* *input, *NNTensor* *weights)
Implements the parametric rectified linear unit.

A PReLU is a specialization of ReLU where negative values are multiplied against a weight instead of set to zero.

The weights can be a vector but must fit evenly within the output tensor.

$$f(x) = \begin{cases} x, & \text{if } x \geq 0 \\ x * w, & \text{otherwise} \end{cases}$$

Since 2.4

Parameters

- **output** – tensor which will receive the processed results
- **input** – input tensor against which the output will be generated
- **weights** – the weights tensor used for calculating the prelu

Returns NNError

NNError **nn_leaky_relu**(*NNTensor* *output, *NNTensor* *input, float alpha)
nn_leaky_relu implements the LeakyRelu function.

The formula used for a given input tensor is, output = (input < 0)*alpha*input + (input >= 0) * input. The output shape is equal to the shape of the input.

Since 2.4

Parameters

- **output** – pointer to the output tensor
- **input** – pointer to the input tensor

- **alpha** – the alpha value multiplied to the result.

Returns NNError

NNError **nn_matmul**(*NNTensor* *output, *NNTensor* *a, *NNTensor* *b, bool transpose_a, bool transpose_b)

nn_matmul implements matrix multiplication, A*B = C.

If transpose_a is true, then tensor a is transposed before the matrix multiplication. If transpose_b is true, then tensor a is transposed before the matrix multiplication. Output Shapes: transpose_a = false, transpose_b = false, [a[0], b[1]] transpose_a = false, transpose_b = true, [a[0], b[0]] transpose_a = true, transpose_b = false, [a[1], b[1]] transpose_a = true, transpose_b = true, [a[1], b[0]]

Since 2.0

Parameters

- **output** – pointer to the output tensor
- **a** – pointer to the first input tensor
- **b** – pointer to the second input tensor
- **transpose_a** – boolean stating whether to transpose A
- **transpose_b** – boolean stating whether to transpose B

Returns NNError

NNError **nn_matmul_cache**(*NNTensor* *output, *NNTensor* *cache, *NNTensor* *a, *NNTensor* *b, bool transpose_a, bool transpose_b)

Special version of *nn_matmul* that can use cache tensor, useful for MCU.

If the cache tensor is at least as large as the b tensor then it will be cached there.

Since 2.0

Parameters

- **output** – pointer to the output tensor
- **cache** – pointer to the cache tensor
- **a** – pointer to the first input tensor
- **b** – pointer to the second input tensor
- **transpose_a** – boolean stating whether to transpose A
- **transpose_b** – boolean stating whether to transpose B

Returns NNError

NNError **nn_dense**(*NNTensor* *output, *NNTensor* *input, *NNTensor* *weights, *NNTensor* *bias, *NNActivation* activation)

nn_dense implements the dense operation.

The formula used for the dense operation is as follows, activation(input * weights + bias) = output. The output shape is [input[0], weights[1]].

See also:

NNActivation

Since 2.0

Parameters

- **output** – pointer to the output tensor
- **input** – pointer to the input tensor
- **weights** – pointer to the weight tensor
- **bias** – pointer to the bias tensor
- **activation** – the activation to be performed

Returns NNError

NNError **nn_linear**(*NNTensor* *output, *NNTensor* *input, *NNTensor* *weights, *NNTensor* *bias, *NNActivation* activation)

nn_linear implements the linear operation.

The formula used for the linear operation is as follows, activation(input * transpose(weights) + bias) = output. The output shape is [input[0], weights[0]].

See also:

NNActivation

Since 2.0

Parameters

- **output** – pointer to the output tensor
- **input** – pointer to the input tensor
- **weights** – pointer to the weight tensor
- **bias** – pointer to the bias tensor
- **activation** – the activation to be performed

Returns NNError

NNError **nn_conv**(*NNTensor* *output, *NNTensor* *cache, *NNTensor* *input, *NNTensor* *filter, *NNTensor* *bias, const int32_t stride[4], int groups, *NNActivation* activation)

nn_conv implements the 2D convolution layer given a 4-D input and filter.

The output shape will have the shape format NHWC with N = input[0], C = filter[3] and the shape values HW given by the formula, floor((input[i] - filter[i]) / stride[i]) + 1

See also:

NNActivation

Since 2.0

Parameters

- **output** – pointer to the output tensor
- **cache** – pointer to the cache tensor, used to increase efficiency
- **input** – pointer to the input tensor
- **filter** – pointer to the filter tensor
- **bias** – pointer to the bias tensor
- **stride** – array providing the strides
- **groups** – the number of convolution groups
- **activation** – the activation to be performed after the convolution

Returns NNError

NNError **nn_conv_ex**(*NNTensor* *output, *NNTensor* *cache, *NNTensor* *input, *NNTensor* *filter, *NNTensor* *bias, const int32_t stride[4], const int32_t padding[8], const int32_t dilation[4], int groups, *NNActivation* activation)

Extended version of the *nn_conv* operation.

The extensions provide support for custom padding handling as well as filter dilation. It is otherwise the same as *nn_conv* which itself calls into this function with padding [0,0,0,0,0,0,0,0] and dilation [1,1,1,1].

The padding parameter must be an array of 4 pairs (8 elements) of int32_t which represent the head/tail paddings for each of the 4 dimensions used by nn_conv_ex.

The dilation parameter must be an array of 4 elements of int32_t which represent the dilation rate of the filter in each of the 4 dimensions used by nn_conv_ex. Dilation of 1 is the same as the standard convolution with each filter spatial element moving by 1 in their respective dimension.

See also:

NNActivation

Since 2.3

Parameters

- **output** – pointer to the output tensor
- **cache** – pointer to the cache tensor, used to increase efficiency
- **input** – pointer to the input tensor
- **filter** – pointer to the filter tensor
- **bias** – pointer to the bias tensor
- **stride** – array providing the strides
- **padding** – array providing the padding sizes
- **dilation** – array providing the dilation amounts
- **groups** – the number of convolution groups
- **activation** – the activation to be performed after the convolution

Returns NNError

NNError **nn_transpose_conv2d_ex**(*NNTensor* *output, *NNTensor* *cache, *NNTensor* *input, *NNTensor* *filter, *NNTensor* *bias, const int32_t stride[4], const int32_t padding[8], *NNActivation* activation)

nn_transpose_conv2d_ex implements the 2d transpose convolution It is the adjoint operator of the standard conv2d.

See also:

NNActivation

Since 2.4

Parameters

- **output** – pointer to the output tensor
- **cache** – pointer to the cache tensor, used to increase efficiency
- **input** – pointer to the input tensor
- **filter** – pointer to the filter tensor
- **bias** – pointer to the bias tensor
- **stride** – array providing the strides
- **padding** – array providing the padding amounts
- **activation** – the activation to be performed after the convolution

Returns NNError

NNError **nn_maxpool**(*NNTensor* *output, *NNTensor* *input, const int32_t window[4], const int32_t stride[4])

nn_maxpool implements the max pooling layer given a 4-D input.

The output shape will have the shape format NHWC with N = input[0], C = input[3] and the shape values HW given by the formula, floor((input[i] - window[i]) / stride[i]) + 1

Since 2.0

Parameters

- **output** – pointer to the output tensor
- **input** – pointer to the input tensor
- **window** – array providing the window size
- **stride** – array providing the strides

Returns NNError

NNError **nn_maxpool_ex**(*NNTensor* *output, *NNTensor* *input, const int32_t window[4], const int32_t stride[4], const int32_t padding[8], const int32_t dilation[4])

Extended version of *nn_maxpool* which provides fused padding support and window dilation, refer to *nn_conv_ex* for details for padding and dilation parameters.

Since 2.3

Parameters

- **output** – pointer to the output tensor
- **input** – pointer to the input tensor
- **window** – array providing the window size
- **stride** – array providing the strides
- **padding** – array providing the padding amounts
- **dilation** – array providing the dilation amounts

Returns NNError

NNError **nn_avgpool**(*NNTensor* *output, *NNTensor* *input, const int32_t window[4], const int32_t stride[4])

nn_avgpool implements the average pooling layer given a 4-D input.

The output shape will have the shape format NHWC with N = input[0], C = input[3] and the shape values HW given by the formula, floor((input[i] - window[i]) / stride[i]) + 1

Since 2.0

Parameters

- **output** – pointer to the output tensor
- **input** – pointer to the input tensor
- **window** – array providing the window size
- **stride** – array providing the strides

Returns NNError

NNError **nn_avgpool_ex**(*NNTensor* *output, *NNTensor* *input, *NNTensor* *cache, const int32_t window[4], const int32_t stride[4], const int32_t padding[8], const int32_t dilation[4])

Extended version of *nn_avgpool* which provides fused padding support and window dilation, refer to *nn_conv_ex* for details for padding and dilation parameters.

Since 2.0

Parameters

- **output** – pointer to the output tensor
- **input** – pointer to the input tensor
- **window** – array providing the window size
- **stride** – array providing the strides
- **padding** – array providing the padding amounts
- **dilation** – array providing the dilation amounts

Returns NNError

NNError **nn_reduce_sum**(*NNTensor* *output, *NNTensor* *input, int32_t n_axes, const int32_t axes[NN_ARRAY_PARAM(*n_axes*)], bool keep_dims)

nn_reduce_sum reduces the axes in (axes[n_axes]) removing the axes in the output tensor unless keep_dims is true in which case they are kept at 1.

Since 2.0

Parameters

- **output** – pointer to the output tensor
- **input** – pointer to the input tensor
- **axes** – pointer to the axes to be reduced
- **n_axes** – number of axes to be reduced

- **keep_dims** – boolean to determine whether to maintain the reduced dimensions as 1

Returns NNError

NNError **nn_reduce_min**(*NNTensor* *output, *NNTensor* *input, int32_t n_axes, const int32_t axes[NN_ARRAY_PARAM(*n_axes*)], bool keep_dims)

nn_reduce_min reduces the axes in (axes[n_axes]) removing the axes in the output tensor unless keep_dims is true in which case they are kept at 1.

Since 2.0

Parameters

- **output** – pointer to the output tensor
- **input** – pointer to the input tensor
- **axes** – pointer to the axes to be reduced
- **n_axes** – number of axes to be reduced
- **keep_dims** – boolean to determine whether to maintain the reduced dimensions as 1

Returns NNError

NNError **nn_reduce_max**(*NNTensor* *output, *NNTensor* *input, int32_t n_axes, const int32_t axes[NN_ARRAY_PARAM(*n_axes*)], bool keep_dims)

nn_reduce_max reduces the axes in (axes[n_axes]) removing the axes in the output tensor unless keep_dims is true in which case they are kept at 1.

Since 2.0

Parameters

- **output** – pointer to the output tensor
- **input** – pointer to the input tensor
- **axes** – pointer to the axes to be reduced
- **n_axes** – number of axes to be reduced
- **keep_dims** – boolean to determine whether to maintain the reduced dimensions as 1

Returns NNError

NNError **nn_reduce_mean**(*NNTensor* *output, *NNTensor* *input, int32_t n_axes, const int32_t axes[NN_ARRAY_PARAM(*n_axes*)], bool keep_dims)

nn_reduce_mean reduces the axes in (axes[n_axes]) removing the axes in the output tensor unless keep_dims is true in which case they are kept at 1.

Since 2.0

Parameters

- **output** – pointer to the output tensor
- **input** – pointer to the input tensor
- **axes** – pointer to the axes to be reduced
- **n_axes** – number of axes to be reduced
- **keep_dims** – boolean to determine whether to maintain the reduced dimensions as 1

Returns NNError

NNError **nn_reduce_product**(*NNTensor* *output, *NNTensor* *input, int32_t n_axes, const int32_t axes[NN_ARRAY_PARAM(*n_axes*)], bool keep_dims)

nn_reduce_product reduces the axes in (axes[n_axes]) removing the axes in the output tensor unless keep_dims is true in which case they are kept at 1.

Since 2.0

Parameters

- **output** – pointer to the output tensor
- **input** – pointer to the input tensor
- **axes** – pointer to the axes to be reduced
- **n_axes** – number of axes to be reduced
- **keep_dims** – boolean to determine whether to maintain the reduced dimensions as 1

Returns NNError

NNError **nn_softmax**(*NNTensor* *output, *NNTensor* *input)

nn_softmax implements the softmax on the tensor along the 4th dimension.

The output shape is equal to the input shape

Since 2.0

Parameters

- **output** – pointer to the output tensor
- **input** – pointer to the input tensor

Returns NNError

NNError **nn_argmax**(*NNTensor* *input, int *index, void *value, size_t value_size)

Finds the maximum value in the flattened tensor, storing the index into the target of the index pointer maximum scalar value into value.

Both index and value are ignored if NULL. The value pointer must be at least value_size bytes long which in turn must be large enough to hold input's scalar size.

For non-flattened version refer to *nn_reduce_max*.

Since 2.0

Parameters

- **input** – pointer to the input tensor
- **index** – pointer to an integer to receive the index of the maximum value.
- **value** – pointer to a variable of size `value_size` which will receive the maximum value found. If NULL value is not saved and `value_size` can be zero.
- **value_size** – size of the variable pointed to by `value`.

Returns NNError

NNError **nn_batchnorm**(*NNTensor* *output, *NNTensor* *cache, *NNTensor* *input, *NNTensor* *mean, *NNTensor* *variance, *NNTensor* *offset, *NNTensor* *scale, *NNTensor* *epsilon)

nn_batchnorm implements batch normalization on a given input.

The formula used for batch normalization is as follows, scale * (input - mean) / sqrt(variance + epsilon) + offset = output. The shape of the output tensor is equal to that of the input tensor.

Since 2.0

Parameters

- **output** – pointer to the output tensor
- **cache** – pointer to the cache tensor, used to increase efficiency
- **input** – pointer to the input tensor
- **mean** – pointer to the mean tensor

- **variance** – pointer to the variance tensor
- **offset** – pointer to the offset tensor
- **scale** – pointer to the scale tensor
- **epsilon** – pointer to the epsilon tensor

Returns NNError

NNError **nn_resize**(*NNTensor* *output, *NNTensor* *input, int mode, bool align_corners, bool half_pixel_centers)

Resizes the tensor spatial (height/width) dimensions using either nearest neighbour or bilinear interpolation.

Modes:

- 0: nearest neighbour scaling
- 1: bilinear interpolation scaling

Since 2.4.18

Since 2.4.18

Note: Currently bilinear interpolation is not implemented on CPU and will fallback to nearest neighbour.

Note: Currently bilinear interpolation is not implemented on CPU and will fallback to nearest neighbour.

Parameters

- **output** – tensor which will receive the results and should be the target shape into which the resize operation will be performed.
- **input** – input tensor for resize operation.
- **mode** – the interpolation model to be used, modes are noted above.
- **align_corners** – not currently implemented
- **half_pixel_centers** – not currently implemented
- **output** – tensor which will receive the results and should be the target shape into which the resize operation will be performed.
- **input** – input tensor for resize operation.
- **mode** –
- **align_corners** –

- **half_pixel_centers** –

Returns NNError

Returns

NNError **nn_ssd_decode_nms_standard_bbx**(*NNTensor* *score, *NNTensor* *trans, *NNTensor* *anchors, *NNTensor* *cache, float score_threshold, float iou_threshold, int32_t max_output_size_per_class, *NNTensor* *bbx_out_tensor, *NNTensor* *bbx_out_dim_tensor)

Implements the post-processing for TensorFlow Object Detection API SSD.

It supports both version 1 and 2 of the TensorFlow Object Detection API for SSD models.

Anchors are an unlearned constant which defines the starting candidates for bounding boxes. The anchors can generally be extracted from the source .pb or SavedModel files as well as the Keras H5. Otherwise the source code for the model trainer is where the anchors would be defined.

The detectv4.c example provides an example of using this API and how to access the needed tensors from the model.

Since 2.4.28

Note: While this API was originally added in DeepViewRT 2.4.28 it was not exposed by deepview_ops.h until 2.4.32. To use it in versions 2.4.28 and 2.4.30 the function signature needs to be manually "extern" into the user application.

Parameters

- **score** – Tensor holding scores of tensorflow ssd (the first output)
- **trans** – Tensor holding transformation parameters of tensorflow ssd (second output).
- **anchors** – Tensor holding pre-defined anchors for the ssd model
- **cache_tensor** – Tensor holding the optional cache tensor
- **score_threshold** – threshold to filter out noises
- **iou_threshold** – threshold for nms
- **max_output_size_per_class** – maximum detection per class
- **bbx_out_tensor** – Tensor receiving the bounding box output.
- **bbx_out_dim_tensor** – Tensor receiving the number of valid detections for each class.

Returns NNError

NNError **nn_ssd_decode_nms_variance_bbx**(*NNTensor* *prediction, *NNTensor* *anchors, *NNTensor* *cache, float score_threshold, float iou_threshold, int32_t max_output_size_per_class, *NNTensor* *bbx_out_tensor, *NNTensor* *bbx_out_dim_tensor)

Implements the post-processing for the eIQ Portal variant of SSD.

In this approach, the bounding box encoding method is slightly different compared to the standard TensorFlow approach, each encoded representation is further divided by their corresponding variance and the two outputs of `score` and `trans` are concatenated into a single output `prediction`.

Anchors are an unlearned constant which defines the starting candidates for bounding boxes. When using eIQ Portal the exported DeepViewRT Model will have the anchors embedded into the model automatically. They are also saved in the model's checkpoint folder.

The detectv4.c example provides an example of using this API and how to access the needed tensors from the model.

Since 2.4.28

Note: While this API was originally added in DeepViewRT 2.4.28 it was not exposed by deepview_ops.h until 2.4.32. To use it in versions 2.4.28 and 2.4.30 the function signature needs to be manually "extern" into the user application.

Parameters

- **prediction** – Tensor holding the concatenated scores and transforms
- **anchors** – Tensor holding the pre-defined anchors for the ssd model
- **cache_tensor** – Tensor holding the optional cache memory
- **score_threshold** – threshold to filter out noises
- **iou_threshold** – threshold for nms
- **max_output_size_per_class** – maximum detection per class
- **bbx_out_tensor** – Tensor receiving the bounding boxes
- **bbx_out_dim_tensor** – Tensor receiving the number of valid detections for each class.

Returns NNError

NNError **nn_ssd_nms_full**(*NNTensor* *input, *NNTensor* *bbx_tensor, *NNTensor* *score_tensor, *NNTensor* *cache, float score_threshold, float iou_threshold, int32_t max_output_size, *NNTensor* *bbx_out_tensor, *NNTensor* *bbx_out_dim_tensor)

Performs the NMS portion of the SSD box decoding algorithm.

Deprecated:
It is deprecated since 2.4.32 in favour of the more complete implementations applicable to eIQ Portal models using *nn_ssd_decode_nms_variance_bbx()* or for TensorFlow 1.x/2.x Object Detection API models by using *nn_ssd_decode_nms_standard_bbx()*.

Since 2.4.25

Returns NNError

PYTHON MODULE INDEX

d

INDEX

Symbols

A

C

D

E

F

G

L

M

N

P

R

S

T

U

V