SyNAP

Synaptics Confidential. Disclosed Only Under NDA - Limited Distribution.

PN: 511-000xxx-01 Rev 3.0.0

Contents

List of Figures

List of Tables

1. Introduction

The purpose of SyNAP is to support the execution of neural networks by taking advantage of the available hardware accelerators. The execution of a *Neural Network*, commonly called an *inference* or a *prediction*, consists of taking one or more inputs and applying a neural network to them to generate one or more outputs. Each input and output is represented with an n-dimensional array of data, called a *Tensor*. Execution takes place inside the Network Processing Unit (*NPU*) accelerator or in the *GPU* or directly in the *CPU*. In order to do this, the network has to be converted from its original representation (e.g. Tensorflow Lite) to the internal SyNAP representation, optimized for the target hardware.

This conversion can occur at two different moments:

- at runtime, when the network is going to be executed, by using a just-in-time compiler and optimizer. We call this *Online Model Conversion*.
- ahead of time, by applying offline conversion and optimization tools which generate a precompiled representation of the network specific for the target hardware. We call this *Offline Model Conversion*.

1.1. Online Model Conversion

Online model conversion allows to execute a model directly without any intermediate steps. This is the most flexible method as all the required conversions and optimizations are done on the fly at runtime, just before the model is executed. The price to be paid for this is that model compilation takes some time (typically a few seconds) when the model is first executed. Another important limitation is that online execution is not available in a secure media path, that is to process data in secure streams.

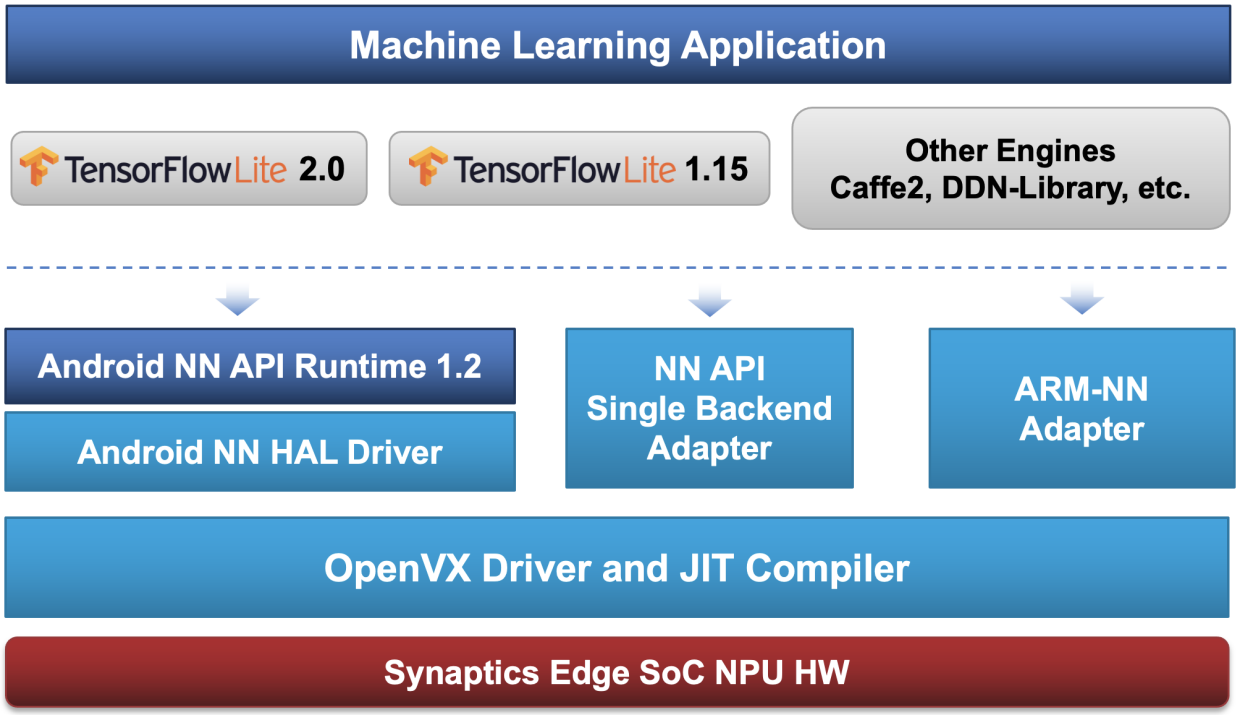

Figure 1. Online model conversion and execution

1.2. Offline Model Conversion

In this mode the network has to be converted from its original representation (e.g. Tensorflow Lite) to the internal SyNAP representation, optimized for the target hardware. Doing the optimization offline allows to perform the highest level of optimizations possible without the tradeoffs of the just-in-time compiler.

In most cases the model conversion can be done with a one-line command using SyNAP toolkit. SyNAP toolkit also supports more advanced operations, such as network quantization and preprocessing. Optionally an offline model can also be signed and encrypted to support Synaptics SyKURE™ secure inference technology.

Note: a compiled model is target-specific and will fail to execute on a different hardware

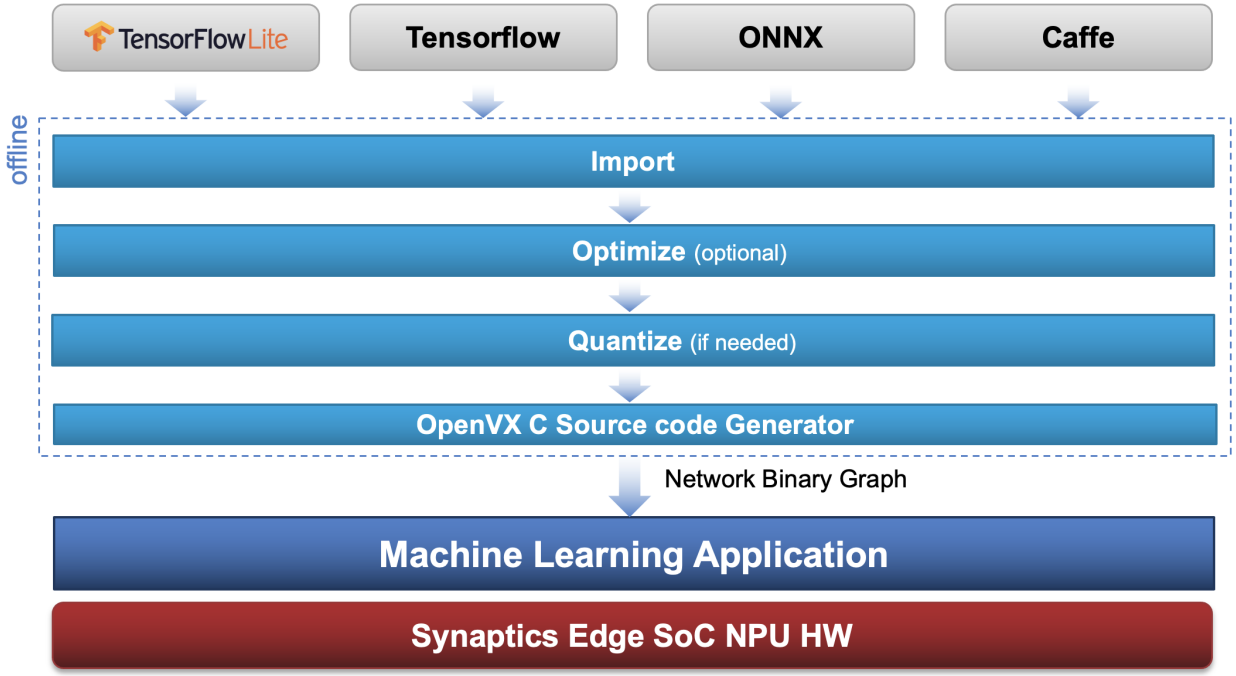

Figure 2. Offline model conversion and execution

1.3. NPU Hardware Capabilities

The NPU itsef actually consists of multiple units. These units can execute in parallel when possible to reduce inference times as much as possible. Three types of units are available:

- **Convolutional Core**: these units are highly optimized to execute convolutions. Cannot be used to implement any other layer.
- **Tensor Processor**: optimized to execute highly parallel operations, such as tensor-add. The *Lite* version is similar but supports a reduced operation set.
- **Parellel Processing Unit**: very flexible unit that can be programmed to process tensors with customized kernels. It supports SIMD execution but it is less specialized (so less efficient) than the Neural Network Engine and Parellel Processing Units.

In addition the NPU also contains an internal static RAM which is used to provide fast access to the data and/or weights thus reducing the need to access the slower external memory during processing.

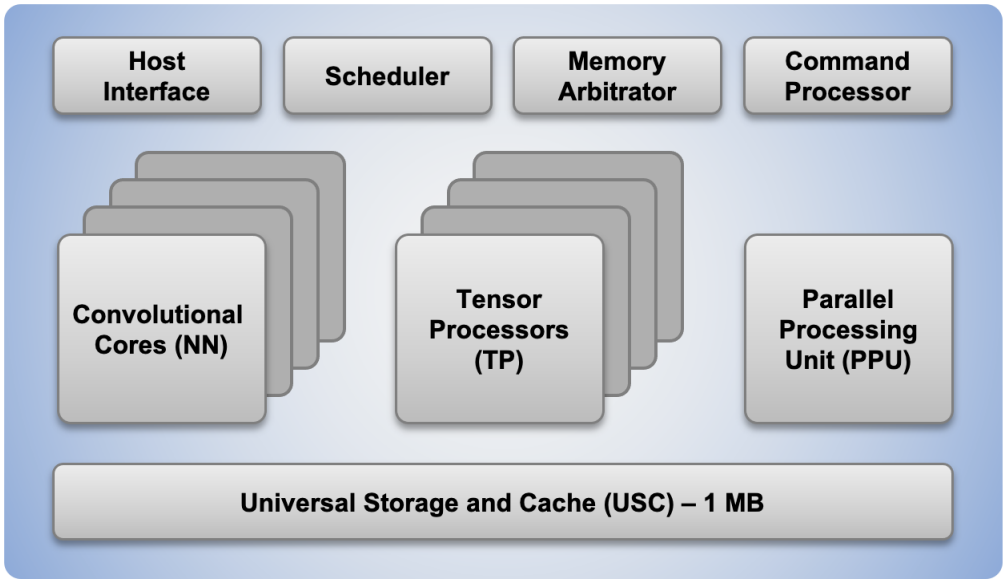

Figure 3. NPU Architecture

The NPU of each SoC differ in the number of units it contains:

SoC	Neural Network Core	Tensor Processor	Parallel Processing Unit
VS640, SL1640	4	2 + 4 Lite	1
VS680, SL1680	22	8	1

2. Getting Started

The simplest way to start experimenting with *Synp* is to use the sample precompiled models and applications that come preinstalled on the board.

Important: On Android the sample models can be found in `/vendor/firmware/models/` while on Yocto Linux they are in `/usr/share/synap/models/`. In this document we will refer to this directory as `$MODELS`.

The models are organized in broad categires according to the type of data they take in input and the information they generate in output. Inside each category, models are organized per topic (for example “imagenet”) and for each topic a set of models and sample input data is provided.

For each category a corresponding command line test application is provided.

Table 1. Model Classification

Category	Input	Output	Test App
image_classification	image	probabilities (one per class)	synap_cli_ic
object_detection	image	detections (bound.box+class+probability)	synap_cli_od
image_processing	image	image	synap_cli_ip

In addition to the specific applications listed above `synap_cli` can be used to execute models of all categories. The purpose of this application is not to provide high-level outputs but to measure inference timings. This is the only sample application that can be used with models requiring secure inputs or outputs.

2.1. `synap_cli_ic` Application

This command line application allows to easily execute *image_classification* models.

It takes in input:

- the converted synap model (*.synap* extension)
- one or more images (*jpeg* or *png* format)

It generates in output:

- the top5 most probable classes for each input image provided

Note: The jpeg/png input image(s) are resized in SW to the size of the network input tensor. This is not included in the classification time displayed.

Example:

```
$ cd $MODELS/image_classification/imagenet/model/mobilenet_v2_1.0_224_quant
$ synap_cli_ic -m model.synap ../../sample/goldfish_224x224.jpg
Loading network: model.synap
Input image: ../../sample/goldfish_224x224.jpg
Classification time: 3.00 ms
Class  Confidence  Description
    1       18.99  goldfish, Carassius auratus
  112        9.30  conch
  927        8.70  trifle
   29        8.21  axolotl, mud puppy, Ambystoma mexicanum
  122        7.71  American lobster, Northern lobster, Maine lobster, Homarus americanus
```

2.2. synap_cli_od Application

This command line application allows to easily execute *object_detection* models.

It takes in input:

- the converted synap model (.*synap* extension)
- optionally the confidence threshold for detected objects
- one or more images (*jpeg* or *png* format)

It generates in output:

- the list of object detected for each input image provided and for each of them the following information:
  - bounding box
  - class index
  - confidence

Note: The jpeg/png input image(s) are resized in SW to the size of the network input tensor.

Example:

```
$ cd $MODELS/object_detection/people/model/mobilenet224_full1/
$ synap_cli_od -m model.synap ../../sample/sample001_640x480.jpg
Input image: ../../sample/sample001_640x480.jpg (w = 640, h = 480, c = 3)
Detection time: 26.94 ms
#   Score  Class  Position  Size     Description
0   0.95       0   94,193    62,143  person
```

Important: The output of object detection models is not standardized, many different formats exist. The output format used has to be specified when the model is converted, see *Model Conversion Tutorial*. If this information is missing or the format is unknown `synap_cli_od` doesn't know how to interpret the result and so it fails with an error message: *"Failed to initialize detector"*.

2.3. synap_cli_ip Application

This command line application allows to execute *image_processing* models. The most common case is the execution of super-resolution models that take in input a low-resolution image and generate in output a higher resolution image.

It takes in input:

- the converted synap model (.*synap* extension)
- optionally the region of interest in the image (if supported by the model)
- one or more raw images with one of the following extensions: *nv12*, *nv21*, *rgb*, *bgr*, *bgra*, *gray* or *bin*

It generates in output:

- a file containing the processed image in for each input file. The output file is called `outimage<i>_<W>x<H>.<ext>`, where <i> is the index of the corresponding input file, <W> and <H> are the dimension of the image, and <ext> depends on the type of the output image, for example `nv12` or `rgb`. By output files are created in the current directory, this can be changed with the `--out-dir` option.

Note: The input image(s) are automatically resized to the size of the network input tensor. This is not supported for `nv12`: if the network takes in input an `nv12` image, the file provided in input must have the same format and the *WxH* dimensions of the image must correspond to the dimensions of the input tensor of the network.

Note: Any `png` and `jpeg` image can be converted to `nv12` and rescaled to the required size using the `image_to_raw` command available in the *SyNAP* `toolkit` (for more info see *Installing Docker*). In the same way the generated raw `nv12` or `rgb` images can be converted to `png` or `jpeg` format using the `image_from_raw` command.

Example:

```
$ cd $MODELS/image_processing/super_resolution/model/sr_qdeo_y_uv_1920x1080_3840x2160
$ synap_cli_ip -m model.synap ../../sample/ref_1920x1080.nv12
Input buffer: input_0 size: 1036800
Input buffer: input_1 size: 2073600
Output buffer: output_13 size: 4147200
Output buffer: output_14 size: 8294400

Input image: ../../sample/ref_1920x1080.nv12
Inference time: 30.91 ms
Writing output to file: outimage0_3840x2160.nv12
```

2.4. `synap_cli_ic2` Application

This application executes two models in sequence, the input image is fed to the first model and its output is then fed to the second one which is used to perform classification as in `synap_cli_ic`. It provides an easy way to experiment with 2-stage inference, where for example the the first model is a *preprocessing* model for downscaling and/or format conversion (see Section 4.3) and the second is an *image_classification* model.

It takes in input:

- the converted synap *preprocessing* model (.*synap* extension)
- the converted synap *classification* model (.*synap* extension)
- one or more images (*jpeg* or *png* format)

It generates in output:

- the top5 most probable classes for each input image provided

Note: The shape of the output tensor of the first model must match that of the input of the second model.

As an example we can use a preprocessing model to convert and rescale a `NV12` image to `RGB` so that it can be processed by the standard `mobilenet_v2_1.0_224_quant` model:

```
$ pp=$MODELS/image_processing/preprocess/model/convert_nv12@1920x1080_rgb@224x224
$ cd $MODELS/image_classification/imagenet/model/mobilenet_v2_1.0_224_quant
$ synap_cli_ic2 -m $pp/model.synap -m2 model.synap ../../sample/goldfish_1920x1080.nv12

Inference time: 4.34 ms
Class  Confidence  Description
    1       19.48  goldfish, Carassius auratus
  122       10.68  American lobster, Northern lobster, Maine lobster, Homarus americanus
  927        9.69  trifle
  124        9.69  crayfish, crawfish, crawdad, crawdaddy
  314        9.10  cockroach, roach
```

The classification output is very close to what we get in Section 2.1, the minor difference is due to the difference in the image rescaled from NV12. The bigger overall inference time is due to the processing required to perform rescale and conversion of the input 1920x1080 image.

2.5. synap_cli Application

This command line application can be used to run models of all categories. The purporse of `synap_cli` is not to show inference results but to benchmark the network execution times. So it provides additional options that allow to run inference mutiple time in order to collect statistics.

An additional feature is that `synap_cli` can automatically generate input images with random content. This makes it easy to test any model even without having a suitable input file available.

Example:

```
$ cd $MODELS/image_classification/imagenet/model/mobilenet_v2_1.0_224_quant
$ synap_cli -m model.synap -r 50 random
Flush/invalidate: yes
Loop period (ms): 0
Network inputs: 1
Network outputs: 1
Input buffer: input_0 size: 150528 : random
Output buffer: output_66 size: 1001

Predict #0: 2.68 ms
Predict #1: 1.81 ms
Predict #2: 1.79 ms
Predict #3: 1.79 ms
.....
Inference timings (ms):  load: 55.91  init: 3.84  min: 1.78  median: 1.82  max: 2.68  stddev:
↪0.13  mean: 1.85
```

Note: Specifying a `random` input is the only way to execute models requiring secure inputs.

2.6. synap_init Application

The purpose of this application is not to execute a model but just to initialize and lock the NPU. It can be used to simulate a process locking the NPU for his exclusive usage.

Example to lock NPU access:

```
$ synap_init -i --lock
```

The lock is released when the program exits or is terminated.

Note: This prevents any process from accessing the NPU via both NNAPI and direct SyNAP API. Please refer to the next section to disable NPU access only for NNAPI.

Note: While the NPU is locked it is still possible to create a Network from another process, but any attempts to do inference will fail. When this occours, the appropriate error message is added to the system log:

```
$ synap_cli_ic
Loading network: /vendor/firmware/models/image_classification/imagenet/model/mobilenet_v2_1.0_
↪224_quant/model.synap
Inference failed
$ dmesg | grep NPU
[ 1211.651] SyNAP: cannot execute model because the NPU is reserved by another user
```

2.7. Troubleshooting

SyNAP libraries and command line applications generate logging messages to help troubleshooting in case something goes wrong. On Android these messages appear in logcat, while on linux they are sent directly to the console.

There are 4 logging levels:

- 0: verbose
- 1: info
- 2: warning
- 3: error

The default level is 3, so that only error logs are generated. It is possible to select a different level by setting the SYNAP_NB_LOG_LEVEL environment variable before starting the application, for example to enable logs up to `info`:

```
export SYNAP_NB_LOG_LEVEL=1
logcat -c; synap_cli_ic; logcat -d | grep SyNAP
Input image: /vendor/firmware/models/image_classification/imagenet/sample/space_shuttle_
↪224x224.jpg
Classification time: 3.16 ms
Class  Confidence  Description
  812       19.48  space shuttle
  ...
1-08 15:10:57.185 830 830 I SyNAP : get_network_attrs():70: Parsing network metadata
1-08 15:10:57.185 830 830 I SyNAP : load_model():252: Network inputs: 1
1-08 15:10:57.185 830 830 I SyNAP : load_model():253: Network outputs: 1
1-08 15:10:57.191 830 830 I SyNAP : resume_cpu_access():65: Resuming cpu access on dmabuf: 5
1-08 15:10:57.193 830 830 I SyNAP : set_buffer():208: Buffer set for tensor: input_0
1-08 15:10:57.193 830 830 I SyNAP : resume_cpu_access():65: Resuming cpu access on dmabuf: 6
1-08 15:10:57.193 830 830 I SyNAP : set_buffer():208: Buffer set for tensor: output_66
1-08 15:10:57.193 830 830 I SyNAP : do_predict():83: Start inference
1-08 15:10:57.193 830 830 I SyNAP : suspend_cpu_access():54: Suspending cpu access on dmabuf:
↪5
1-08 15:10:57.195 830 830 I SyNAP : do_predict():95: Inference time: 2.33 ms
1-08 15:10:57.195 830 830 I SyNAP : resume_cpu_access():65: Resuming cpu access on dmabuf: 6
1-08 15:10:57.196 830 830 I SyNAP : unregister_buffer():144: Detaching buffer from input
↪tensor input_0
1-08 15:10:57.196 830 830 I SyNAP : set_buffer():177: Unset buffer for: input_0
1-08 15:10:57.196 830 830 I SyNAP : unregister_buffer():150: Detaching buffer from output
↪tensor output_66
1-08 15:10:57.196 830 830 I SyNAP : set_buffer():177: Unset buffer for: output_66
```

3. Using Online Inference

3.1. Online Inference With NNAPI

When a model is loaded and executed via NNAPI it is automatically converted to the internal representation suitable for execution on the NPU. This conversion doesn't take place when the model is loaded but when the first inference is executed. This is because the size of the input(s) is needed in order to perform the conversion and with some models this information is available only at inference time. If the input size is specified in the model, then the provided input(s) must match this size. In any case it is not possible to change the size of the input(s) after the first inference.

The model compilation has been heavily optimized, but even so it can take several milliseconds up to a few seconds for typical models, so it is suggested to execute an inference once just after the model has been loaded and prepared. One of the techniques used to speedup model compilation is caching, that is some results of the computations performed to compile a model are cashed in a file so that they don't have to be executed again the next time the same model is compiled.

On Android the cache file is saved by default to `/data/vendor/synap/nnhal.cache` and will contain up to 10000 entries which corresponds to a good setting for NNAPI utilization on an average system. Cache path and size can be changed by setting the properties `vendor.SYNAP_CACHE_PATH` and `vendor.SYNAP_CACHE_CAPACITY`. Setting the capacity to 0 will disable the cache. An additional possibility to speedup model compilation is to use the NNAPI cache, see : *NNAPI Compilation Caching*.

On yocto linux there is no NNAPI cache, but we still have smaller per-process cache files named `synap-cache.<PROGRAM-NAME>` in the `/tmp/` directory.

3.2. Benchmarking Models With NNAPI

It is possible to benchmark the execution of a model with online conversion using the standard Android NNAPI tool `android_arm_benchmark_model` from https://www.tensorflow.org/lite/performance/measurement

A custom version of this tool optimized for SyNAP platforms called `benchmark_model` is already preinstalled on the board in `/vendor/bin`.

Benchmarking a model is quite simple:

1. Download the tflite model to be benchmarked, for example:

   https://storage.googleapis.com/download.tensorflow.org/models/mobilenet_v1_2018_08_02/mobilenet_v1_0.25_224_quant.tgz

2. Copy the model to the board, for example in the `/data/local/tmp` directory:

```
$ adb push mobilenet_v1_0.25_224_quant.tflite /data/local/tmp
```

3. Benchmark the model execution on the NPU with NNAPI (android only):

```
$ adb shell benchmark_model --graph=/data/local/tmp/mobilenet_v1_0.25_224_quant.tflite --
↪use_nnapi=true --nnapi_accelerator_name=synap-npu

INFO: STARTING!
INFO: Tensorflow Version : 2.15.0
INFO: Log parameter values verbosely: [0]
INFO: Graph: [/data/local/tmp/mobilenet_v1_0.25_224_quant.tflite]
INFO: Use NNAPI: [1]
INFO: NNAPI accelerator name: [synap-npu]
INFO: NNAPI accelerators available: [synap-npu,nnapi-reference]
INFO: Loaded model /data/local/tmp/mobilenet_v1_0.25_224_quant.tflite
```

(continues on next page)

(continued from previous page)

```
INFO: Initialized TensorFlow Lite runtime.
INFO: Created TensorFlow Lite delegate for NNAPI.
INFO: NNAPI delegate created.
WARNING: NNAPI SL driver did not implement SL_ANeuralNetworksDiagnostic_
↪registerCallbacks!
VERBOSE: Replacing 31 out of 31 node(s) with delegate (TfLiteNnapiDelegate) node,
↪yielding 1 partitions for the whole graph.
INFO: Explicitly applied NNAPI delegate, and the model graph will be completely executed
↪by the delegate.
INFO: The input model file size (MB): 0.497264
INFO: Initialized session in 66.002ms.
INFO: Running benchmark for at least 1 iterations and at least 0.5 seconds but terminate
↪if exceeding 150 seconds.
INFO: count=1 curr=637079

INFO: Running benchmark for at least 50 iterations and at least 1 seconds but terminate
↪if exceeding 150 seconds.
INFO: count=520 first=2531 curr=2793 min=1171 max=9925 avg=1885.74 std=870

INFO: Inference timings in us: Init: 66002, First inference: 637079, Warmup (avg):
↪637079, Inference (avg): 1885.74
INFO: Note: as the benchmark tool itself affects memory footprint, the following is only
↪APPROXIMATE to the actual memory footprint of the model at runtime. Take the
↪information at your discretion.
INFO: Memory footprint delta from the start of the tool (MB): init=7.40234 overall=7.
↪83203
```

Important: NNAPI is the standard way to perform online inference on the NPU in Android, but it isn't the most efficient or the most flexible one. The suggested way to perform online inference on Synaptics platforms is via the `timvx` delegate. For more information see section Section 3.6.

If for any reason some of the layers in the model cannot be executed on the NPU, they will automatically fall back to CPU execution. This can occur for example in case of specific layer types, options or data types not supported by NNAPI or SyNAP. In this case the network graph will be partitioned in multiple delegate kernels as indicated in the output messages from `benchmark_model`, for example:

```
$ adb shell benchmark_model ...
...
INFO: Initialized TensorFlow Lite runtime.
INFO: Created TensorFlow Lite delegate for NNAPI.
Explicitly applied NNAPI delegate, and the model graph will be partially executed by the
↪delegate w/ 2 delegate kernels.
...
```

Executing part of the network on the CPU will increase inference times, sometimes considerably. To better understand which are the problematic layers and where the time is spent it can be useful to run `benchmark_model` with the option `--enable_op_profiling=true`. This option generates a detailed report of the layers executed on the CPU and the time spent executing them. For example in the execution here below the network contains a `RESIZE_NEAREST_NEIGHBOR` layer which falls back to CPU execution:

```
$ adb shell benchmark_model ... --enable_op_profiling=true
...
Operator-wise Profiling Info for Regular Benchmark Runs:
============================== Run Order ==============================
            [node type]  [first]  [avg ms]      [%]    [cdf%]  [mem KB] [times called] [Name]
```

(continues on next page)

(continued from previous page)

```
    TfLiteNnapiDelegate     3.826     4.011  62.037%   62.037%     0.000          1       []:64
RESIZE_NEAREST_NEIGHBOR     0.052     0.058   0.899%   62.936%     0.000          1       []:38
    TfLiteNnapiDelegate     2.244     2.396  37.064%  100.000%     0.000          1       []:65
```

Execution of the model (or part of it) on the NPU can also be confirmed by looking at the SyNAP `inference_count` file in `sysfs` (see section Section 5.1).

For an even more in-depth analysis, it is possible to obtain detailed layer-by-layer inference timing by setting the profiling property before running `benchmark_model`:

```
$ adb shell setprop vendor.NNAPI_SYNAP_PROFILE 1
$ adb shell benchmark_model --graph=/data/local/tmp/mobilenet_v1_0.25_224_quant.tflite --use_
↪nnapi=true --nnapi_accelerator_name=synap-npu
```

On android, the profiling information will be availabe in `/sys/class/misc/synap/device/misc/synap/statistics/network_profile` while `benchmark_model` is running. On yocto linux, the same information is in `/sys/class/misc/synap/statistics/network_profile`.

For more information see section Section 5.4

Note: When `vendor.NNAPI_SYNAP_PROFILE` is enabled, the network is executed step-by-step, so the overall inference time becomes meaningless and should be ignored.

3.3. NNAPI Compilation Caching

NNAPI compilation caching provides even greater speedup than the default SyNAP cache by caching entire compiled models, but it requires some support from the application (see https://source.android.com/devices/neural-networks/compilation-caching) and requires more disk space.

NNAPI caching support must be enabled by setting the corresponding android property:

```
$ adb shell setprop vendor.npu.cache.model 1
```

As explained in the official android documentation, for NNAPI compilation cache to work the user has to provide a directory when to store the cached model and a unique key for each model. The unique key is normally determined by computing some hash on the entire model.

This can be tested using `benchmark_model`:

```
$ adb shell benchmark_model --graph=/data/local/tmp/mobilenet_v1_0.25_224_quant.tflite --use_
↪nnapi=true --nnapi_accelerator_name=synap-npu --delegate_serialize_dir=/data/local/tmp/
↪nnapiacache --delegate_serialize_token='`md5sum -b /data/local/tmp/mobilenet_v1_0.25_224_
↪quant.tflite`'
```

During the first execution of the above command, NNAPI will compile the model and add it to the cache:

```
INFO: Initialized TensorFlow Lite runtime.
INFO: Created TensorFlow Lite delegate for NNAPI.
NNAPI delegate created.
ERROR: File /data/local/tmp/nnapiacache/a67461dd306cfd2ff0761cb21dedffe2_6183748634035649777.
↪bin couldn't be opened for reading: No such file or directory
INFO: Replacing 31 node(s) with delegate (TfLiteNnapiDelegate) node, yielding 1 partitions.
...
Inference timings in us: Init: 34075, First inference: 1599062, Warmup (avg): 1.59906e+06,
↪Inference (avg): 1380.86
```

In all the following executions NNAPI will load the compiled model directly from the cache, so the first inference will be faster:

```
INFO: Initialized TensorFlow Lite runtime.
INFO: Created TensorFlow Lite delegate for NNAPI.
NNAPI delegate created.
INFO: Replacing 31 node(s) with delegate (TfLiteNnapiDelegate) node, yielding 1 partitions.
...
Inference timings in us: Init: 21330, First inference: 90853, Warmup (avg): 1734.13,
↪Inference (avg): 1374.59
```

3.4. Disabling NPU Usage From NNAPI

It is possible to make the NPU inaccessible from NNAPI by setting the property `vendor.NNAPI_SYNAP_DISABLE` to 1. In this case any attempt to run a model via NNAPI will always fall back to CPU.

NNAPI execution with NPU enabled:

```
$ adb shell setprop vendor.NNAPI_SYNAP_DISABLE 0
$ adb shell 'echo > /sys/class/misc/synap/device/misc/synap/statistics/inference_count'
$ adb shell benchmark_model --graph=/data/local/tmp/mobilenet_v1_0.25_224_quant.tflite --use_
↪nnapi=true --nnapi_accelerator_name=synap-npu
Inference timings in us: Init: 24699, First inference: 1474732, Warmup (avg): 1.47473e+06,
↪Inference (avg): 1674.03
$ adb shell cat /sys/class/misc/synap/device/misc/synap/statistics/inference_count
1004
```

NNAPI execution with NPU disabled:

```
$ adb shell setprop vendor.NNAPI_SYNAP_DISABLE 1
$ adb shell 'echo > /sys/class/misc/synap/device/misc/synap/statistics/inference_count'
$ adb shell benchmark_model --graph=/data/local/tmp/mobilenet_v1_0.25_224_quant.tflite --use_
↪nnapi=true --nnapi_accelerator_name=synap-npu
Inference timings in us: Init: 7205, First inference: 15693, Warmup (avg): 14598.5, Inference
↪(avg): 14640.3
$ adb shell cat /sys/class/misc/synap/device/misc/synap/statistics/inference_count
0
```

Note: It will still be possible to perform online inference on the NPU using the *timvx* tflite delegate.

3.5. Online Inference With *TimVx* Delegate

NNAPI is not the only way to perform online inference on the NPU. It is possible to run a model without using NNAPI, by loading it with the standard Tensorflow Lite API and then using the *timvx* tflite delegate. This delegate has been optimized to call directly the SyNAP API, so it can most often provide better performance and less limitations than the standard NNAPI.

Another advantage of the `timvx` delegate is that it is also available on yocto linux platforms which don't support NNAPI. The only limitation of this approach is that being a delegate for the standard Tensorflow runtime, it doesn't support the execution of other model formats such as ONNX.

timvx tflite delegate internal workflow is similar to that of NNAPI: when a tflite model is loaded it is automatically converted to the internal representation suitable for execution on the NPU. This conversion doesn't take place when the model is loaded but when the first inference is executed.

3.6. Benchmarking Models With *TimVx* Delegate

Synaptics `benchmark_model` tool provide built-in support for both the standard `nnapi` delegate, and the optimized `timvx` delegate.

Benchmarking a model with `timvx` delegate is as simple as using NNAPI:

1. Download the tflite model to be benchmarked, for example:

   https://storage.googleapis.com/download.tensorflow.org/models/mobilenet_v1_2018_08_02/mobilenet_v1_0.25_224_quant.tgz

2. Copy the model to the board, for example in the `/data/local/tmp` directory:

```
$ adb push mobilenet_v1_0.25_224_quant.tflite /data/local/tmp
```

3. Benchmark the model execution on the NPU with `timvx` delegate (both android and linux):

```
$ adb shell benchmark_model --graph=/data/local/tmp/mobilenet_v1_0.25_224_quant.tflite --
↪external_delegate_path=libvx_delegate.so

INFO: STARTING!
INFO: Tensorflow Version : 2.15.0
INFO: Log parameter values verbosely: [0]
INFO: Graph: [/data/local/tmp/mobilenet_v1_0.25_224_quant.tflite]
INFO: External delegate path: [/vendor/lib64/libvx_delegate.so]
INFO: Loaded model /data/local/tmp/mobilenet_v1_0.25_224_quant.tflite
INFO: Initialized TensorFlow Lite runtime.
INFO: Vx delegate: allowed_cache_mode set to 0.
INFO: Vx delegate: device num set to 0.
INFO: Vx delegate: allowed_builtin_code set to 0.
INFO: Vx delegate: error_during_init set to 0.
INFO: Vx delegate: error_during_prepare set to 0.
INFO: Vx delegate: error_during_invoke set to 0.
INFO: EXTERNAL delegate created.
VERBOSE: Replacing 31 out of 31 node(s) with delegate (Vx Delegate) node, yielding 1
↪partitions for the whole graph.
INFO: Explicitly applied EXTERNAL delegate, and the model graph will be completely
↪executed by the delegate.
INFO: The input model file size (MB): 0.497264
INFO: Initialized session in 25.573ms.
INFO: Running benchmark for at least 1 iterations and at least 0.5 seconds but terminate
↪if exceeding 150 seconds.
type 54 str SoftmaxAxis0
INFO: count=277 first=201009 curr=863 min=811 max=201009 avg=1760.78 std=11997

INFO: Running benchmark for at least 50 iterations and at least 1 seconds but terminate
↪if exceeding 150 seconds.
INFO: count=876 first=1272 curr=1730 min=810 max=6334 avg=1096.48 std=476

INFO: Inference timings in us: Init: 25573, First inference: 201009, Warmup (avg): 1760.
↪78, Inference (avg): 1096.48
INFO: Note: as the benchmark tool itself affects memory footprint, the following is only
↪APPROXIMATE to the actual memory footprint of the model at runtime. Take the
↪information at your discretion.
INFO: Memory footprint delta from the start of the tool (MB): init=15.4688 overall=43.
↪2852
```

Comparing the timings with those in section Section 3.2 we can notice that even for this simple model, `timvx` delegate provides better performances than NNAPI (average inference time 1096 us vs 1885).

4. Reference Models

4.1. Timings

The tables in this section contain inference timings for a set of representative models. The quantized models have been imported and compiled offline using SyNAP toolkit. The floating point models are benchmarked for comparison purpose with the corresponding quantized models.

The *mobilenet_v1*, *mobilenet_v2*, *posenet* and *inception* models are open-source models available in *tflite* format from *TensorFlow Hosted Models* page: `https://www.tensorflow.org/lite/guide/hosted_models`

yolov5 models are available from `https://github.com/ultralytics/yolov5`, while *yolov5_face* comes from `https://github.com/deepcam-cn/yolov5-face`.

Other models come from AI-Benchmark APK: `https://ai-benchmark.com/ranking_IoT.html`.

Some of the models are Synaptics proprietary, including test models, object detection (*mobilenet224*), super-resolution and format conversion models.

The model *test_64_128x128_5_132_132* has been designed to take maximum advantage of the computational capabilities of the NPU. It has 64 5x5 convolutions with a [1, 128, 128, 132] input and output. Its execution requires 913'519'411'200 operations (0.913 TOPs). Inference time shows that in the right conditions VS640 and SL1640 achieve above 1.6 TOP/s while VS680 and SL1680 able to achieve above 7.9 TOP/s. For 16-bits inference the maximum TOP/s can be achieved with *test_64_64x64_5_132_132*. With this model we achieve 0.45 TOP/s on VS640/SL1640 and above 1.7 TOP/s on VS680/SL1680. For actual models used in practice it's very difficut to get close to this level of performance and it's hard to predict the inference time of a model from the number of operation it contains. The only reliable way is to execute the model on the platform and measure.

Remarks:

- In the following tables all timing values are expressed in milliseconds
- The columns *Online CPU* and *Online NPU* represent the inference time obtained by running the original *tflite* model directly on the board (*online* conversion)
- Online CPU tests have been done with 4 threads (`--num_threads=4`) on both *vs680* and *vs640*
- Online CPU tests of *floating point models* on *vs640* have been done in *fp16* mode (`--allow_fp16=true`)
- Online NPU tests on have been done with the *timvx* delegate (`--external_delegate_path=libvx_delegate.so`).
- The *Offline Infer* column represents the inference time obtained by using a model converted offline using SyNAP toolkit (median time over 10 consecutive inferences)
- The *Online* timings represent the minimum time measured (for both init and inference). We took minimim instead of average because this is measure less sensitive to outliers due to the test process being temporarily suspended by the CPU scheduler
- Online timings, in particular for init and CPU inference, can be influenced by other processes running on the board and the total amount of free memory available. We ran all tests on Android AOSP/64bits with 4GB of memory on VS680 and 2GB on VS640. Running on Android GMS or 32-bits OS or with less memory can result in longer init and inference times
- Timings for SL1640 and SL1680 corresponds to those of VS640 and VS680, respectively
- Offline tests have been done with non-contiguous memory allocation and no cache flush
- Models marked with * come precompiled and preinstalled on the platform

Table 2. Inference timings on VS680, 64-bits OS, 4GB memory

Model	Online CPU Infer	Online GPU Infer	Online NPU Init	Online NPU Infer	Offline NPU Init	Offline NPU Infer	
inception_v4_299_quant	440.54		13502	17.80	100.79	19.59	
mobilenet_v1_0.25_224_quant	3.37		166	0.81	2.61	0.77	*
mobilenet_v2_1.0_224_quant	18.60		854	1.85	6.13	1.79	*
convert_nv12@1920x1080_rgb@1920x1080					17.52	30.81	*
convert_nv12@1920x1080_rgb@224x224					14.25	1.27	*
convert_nv12@1920x1080_rgb@640x360					13.55	5.14	*
sr_fast_y_uv_1280x720_3840x2160			317	32.88	18.09	11.46	*
sr_fast_y_uv_1920x1080_3840x2160			776	50.56	20.40	17.50	*
sr_qdeo_y_uv_1280x720_3840x2160			149	32.49	21.84	20.59	*
sr_qdeo_y_uv_1920x1080_3840x2160			224	38.41	24.11	25.84	*
posenet_mobilenet_075_float	34.44	61.78					*
posenet_mobilenet_075_quant	28.60		382	6.01	1.84	2.32	
yolov8s-pose					14.61	30.79	*
mobilenet224_full80					646.23	25.17	*
yolov5m-640x480			6606	113.88	54.11	118.82	
yolov5s-640x480			2672	72.27	22.17	75.83	
yolov5s_face_640x480_onnx_mq					13.00	32.55	*
mobilenet224_full1					536.81	14.23	*
deeplab_v3_plus_quant	231.76		4090	60.73	7.68	59.81	
dped_quant	335.63		1014	8.93	4.74	8.82	
inception_v3_float	370.95	436.74					
inception_v3_quant	267.95		7210	9.47	59.55	10.22	
mobilenet_v2_b4_quant	68.89		875	12.50	11.53	13.63	
mobilenet_v2_float	20.84	35.40					
mobilenet_v2_quant	18.72		886	2.02	9.27	1.98	
mobilenet_v3_quant	69.05		1089	9.76	13.15	10.15	
pynet_quant	1294.46		3175	18.56	24.45	19.30	
srgan_quant	1513.86		3517	54.58	14.72	56.95	
unet_quant	442.51		487	9.34	7.73	14.80	
vgg_quant	1641.18		2319	29.77	10.74	30.07	
test_64_128x128_5_132_132					50.07	119.34	

Table 3. Inference timings on VS640, 64-bits OS, 2GB memory

Model	Online CPU Infer	Online GPU Infer	Online NPU Init	Online NPU Infer	Offline NPU Init	Offline NPU Infer	
inception_v4_299_quant	255.94		21481	54.07	127.13	53.82	
mobilenet_v1_0.25_224_quant	2.80		244	1.00	4.99	0.93	*
mobilenet_v2_1.0_224_quant	12.31		1203	2.40	14.21	2.31	*
convert_nv12@1920x1080_rgb@1920x1080					17.48	34.49	*
convert_nv12@1920x1080_rgb@224x224					15.14	1.25	*
convert_nv12@1920x1080_rgb@640x360					14.70	5.29	*
sr_fast_y_uv_1280x720_3840x2160			274	53.03	17.87	17.01	*
sr_fast_y_uv_1920x1080_3840x2160			524	86.39	20.35	25.90	*
sr_qdeo_y_uv_1280x720_3840x2160					20.33	26.16	*
sr_qdeo_y_uv_1920x1080_3840x2160					22.03	33.56	*
posenet_mobilenet_075_float	27.96	90.06					*
posenet_mobilenet_075_quant	18.70		565	9.76	2.48	4.13	
yolov8s-pose					20.66	54.59	*
mobilenet224_full80					718.96	52.98	*
yolov5m-640x480			10657	175.90	60.64	178.00	
yolov5s-640x480			4264	101.73	24.90	103.36	
yolov5s_face_640x480_onnx_mq					27.31	59.63	*
mobilenet224_full1					595.52	36.53	*
deeplab_v3_plus_quant	158.81		3679	82.47	8.31	70.85	
dped_quant	134.25		1694	25.84	6.71	25.72	
inception_v3_float	229.07	706.98					
inception_v3_quant	146.25		11130	30.21	80.70	29.82	
mobilenet_v2_b4_quant	47.85		1373	18.95	13.60	18.39	
mobilenet_v2_float	18.27	52.41					
mobilenet_v2_quant	12.44		1282	2.57	13.81	2.44	
mobilenet_v3_quant	47.99		1593	12.57	16.38	11.91	
pynet_quant	447.61		4803	57.11	31.04	56.30	
srgan_quant	829.17		5232	121.92	15.97	121.75	
unet_quant	159.01		745	18.58	9.93	24.20	
vgg_quant	572.70		3258	103.66	10.65	102.65	
test_64_128x128_5_132_132					63.95	563.81	

4.2. Super Resolution

Synaptics provides two proprietary families of super resolution models: *fast* and *qdeo*, the former provides better inference time, the latter better upscaling quality. They can be tested using `synap_cli_ip` application, see Section 2.3.

These models are preinstalled in `$MODELS/image_processing/super_resolution` .

Table 4. Synaptics SuperResolution Models on Y+UV Channels

Name	Input Image	Ouput Image	Factor	Notes
sr_fast_y_uv_960x540_3840x2160	960x540	3840x2160	4	
sr_fast_y_uv_1280x720_3840x2160	1280x720	3840x2160	3	
sr_fast_y_uv_1920x1080_3840x2160	1920x1080	3840x2160	2	
sr_qdeo_y_uv_960x540_3840x2160	960x540	3840x2160	4	
sr_qdeo_y_uv_1280x720_3840x2160	1280x720	3840x2160	3	
sr_qdeo_y_uv_1920x1080_3840x2160	1920x1080	3840x2160	2	
sr_qdeo_y_uv_640x360_1920x1080	640x360	1920x1080	3	

4.3. Format Conversion

Conversion models can be used to convert an image from `NV12` format to `RGB`. A set of models is provided for the most commonly used resolutions. These models have been generated by taking advantage of the preprocessing feature of the `SyNAP` toolkit (see *Preprocessing*) and can be used to convert an image so that is can be fed to a processing model with `RGB` input.

These models are preinstalled in `$MODELS/image_processing/preprocess` and can be tested using `synap_cli_ic2` application, see Section 2.4.

Table 5. Synaptics Conversion Models NV12 to RGB 224x224

Name	Input Image (NV12)	Ouput Image (RGB)	Notes
convert_nv12@426x240_rgb@224x224	426x240	224x224	
convert_nv12@640x360_rgb@224x224	640x360	224x224	
convert_nv12@854x480_rgb@224x224	854x480	224x224	
convert_nv12@1280x720_rgb@224x224	1280x720	224x224	
convert_nv12@1920x1080_rgb@224x224	1920x1080	224x224	
convert_nv12@2560x1440_rgb@224x224	2560x1440	224x224	
convert_nv12@3840x2160_rgb@224x224	3840x2160	224x224	
convert_nv12@7680x4320_rgb@224x224	7680x4320	224x224	

Table 6. Synaptics Conversion Models NV12 to RGB 640x360

Name	Input Image (NV12)	Ouput Image (RGB)	Notes
convert_nv12@426x240_rgb@640x360	426x240	640x360	
convert_nv12@640x360_rgb@640x360	640x360	640x360	
convert_nv12@854x480_rgb@640x360	854x480	640x360	
convert_nv12@1280x720_rgb@640x360	1280x720	640x360	
convert_nv12@1920x1080_rgb@640x360	1920x1080	640x360	
convert_nv12@2560x1440_rgb@640x360	2560x1440	640x360	
convert_nv12@3840x2160_rgb@640x360	3840x2160	640x360	
convert_nv12@7680x4320_rgb@640x360	7680x4320	640x360	

Table 7. Synaptics Conversion Models NV12 to RGB 1920x1080

Name	Input Image (NV12)	Ouput Image (RGB)	Notes
convert_nv12@426x240_rgb@1920x1080	426x240	1920x1080	
convert_nv12@640x360_rgb@1920x1080	640x360	1920x1080	
convert_nv12@854x480_rgb@1920x1080	854x480	1920x1080	
convert_nv12@1280x720_rgb@1920x1080	1280x720	1920x1080	
convert_nv12@1920x1080_rgb@1920x1080	1920x1080	1920x1080	
convert_nv12@2560x1440_rgb@1920x1080	2560x1440	1920x1080	
convert_nv12@3840x2160_rgb@1920x1080	3840x2160	1920x1080	
convert_nv12@7680x4320_rgb@1920x1080	7680x4320	1920x1080	

5. Statistics And Usage

SyNAP provides usage information and statistics. This is done via the standard linux `/sysfs` interface. Basically `/sysfs` allows to provide information about system devices and resources using a *pseudo file-system* where each piece of information is seen as a file that can be read/written by the user using standard tools.

On Android statistics are available in `/sys/class/misc/synap/device/misc/synap/statistics/`:

```
$ SYNAP_STAT_DIR=/sys/class/misc/synap/device/misc/synap/statistics
```

On Yocto Linux they are in `/sys/class/misc/synap/statistics/`:

```
$ SYNAP_STAT_DIR=/sys/class/misc/synap/statistics
```

```
$ ls $SYNAP_STAT_DIR
inference_count  inference_time  network_profile networks
```

Important: The content of the statistics files is only available from **root** user.

5.1. inference_count

This file contains the total number of inferences performed since system startup. Example:

```
# cat $SYNAP_STAT_DIR/inference_count
1538
```

5.2. inference_time

This file contains the total time spent doing inferences since system startup. It is a 64-bits integer expressed in microseconds. Example:

```
# cat $SYNAP_STAT_DIR/inference_time
32233264
```

5.3. networks

This file contains detailed information for each network currently loaded, with a line per network. Each line contains the following information:

- **pid**: process that created the network
- **nid**: unique network id
- **inference_count**: number of inferences for this network
- **inference_time**: total inference time for this network in us
- **inference_last**: last inference time for this network in us
- **iobuf_count**: number of I/O buffers currently registered to the network
- **iobuf_size**: total size of I/O buffers currently registered to the network
- **layers**: number of layers in the network

Example:

```
# cat $SYNAP_STAT_DIR/networks
pid: 3628, nid: 38, inference_count: 22, inference_time: 40048, inference_last: 1843, iobuf_
↪count: 2, iobuf_size: 151529, layers: 34
pid: 3155, nid: 4, inference_count: 3, inference_time: 5922, inference_last: 1843, iobuf_
↪count: 2, iobuf_size: 451630, layers: 12
```

5.4. network_profile

This file contains detailed information for each network currently loaded, with a line per network. The information in each line is the same as in the `networks` file. In addition if a model has been compiled offline with profiling enabled (see section *Model Profiling*) or executed online with profiling enabled (see section *Benchmarking Models With NNAPI*) the corresponding line will be followed by detailed layer-by-layer information:

- **lyr**: index of the layer (or group of layers)
- **cycle**: number of execution cycles
- **time_us**: execution time in us
- **byte_rd**: number of bytes read
- **byte_wr**: number of bytes written
- **ot**: operation type (NN: Neural Network core, SH: Shader, TP: TensorProcessor)
- **name**: operation name

Example:

```
# cat $SYNAP_STAT_DIR/network_profile
pid: 21756, nid: 1, inference_count: 78, inference_time: 272430, inference_last: 3108, iobuf_
↪count: 2, iobuf_size: 151529, layers: 34
| lyr |    cycle | time_us | byte_rd | byte_wr | ot | name
|   0 |   153811 |     202 |  151344 |       0 | TP | TensorTranspose
|   1 |   181903 |     461 |    6912 |       0 | NN | ConvolutionReluPoolingLayer2
|   2 |     9321 |      52 |    1392 |       0 | NN | ConvolutionReluPoolingLayer2
|   3 |    17430 |      51 |    1904 |       0 | NN | ConvolutionReluPoolingLayer2
|   4 |    19878 |      51 |    1904 |       0 | NN | ConvolutionReluPoolingLayer2
...
|  28 |    16248 |      51 |    7472 |       0 | NN | ConvolutionReluPoolingLayer2
|  29 |   125706 |     408 |  120720 |       0 | TP | FullyConnectedReluLayer
|  30 |   137129 |     196 |    2848 |    1024 | SH | Softmax2Layer
|  31 |        0 |       0 |       0 |       0 | -- | ConvolutionReluPoolingLayer2
|  32 |        0 |       0 |       0 |       0 | -- | ConvolutionReluPoolingLayer2
|  33 |      671 |      51 |    1008 |       0 | NN | ConvolutionReluPoolingLayer2
```

5.5. Clearing Statistics

Statistics can be cleared by writing to either the `inference_count` or `inference_time` file.

Example:

```
# cat $SYNAP_STAT_DIR/inference_time
32233264
# echo > $SYNAP_STAT_DIR/inference_time
# cat $SYNAP_STAT_DIR/inference_time
```

(continues on next page)

(continued from previous page)

```
0
# cat $SYNAP_STAT_DIR/inference_count
0
```

5.6. Using /sysfs Information

The information available from /sysfs can be easily used from scripts or tools. For example in order to get the average NPU utilization in a 5 seconds period:

```
us=5000000;
echo > $SYNAP_STAT_DIR/inference_time;
usleep $us;
npu_usage=$((`cat $SYNAP_STAT_DIR/inference_time`*100/us));
echo "Average NPU usage: $npu_usage%"
```

6. Working With Models

6.1. Model Conversion

The SyNAP toolkit allows to convert a model from its original format to an internal representation optimized for the target hardware. The conversion tool and utilities can run on Linux, MacOS or Windows hosts inside a *Docker* container. Only *Docker* and the `toolkit` image are required, no additional dependencies have to be installed.

The following network model formats are supported:

- Tensorflow Lite (`.tflite` extension)
- Tensorflow (`.pb` extension)
- ONNX (`.onnx` extension)
- Caffe (`.prototxt` extension)

Note: Only standard Caffe 1.0 models are supported. Custom variants such as *Caffe-SSD* or *Caffe-LSTM* models or legacy (pre-1.0) models require specific parsers which are currently not available in SyNAP toolkit. *Caffe2* models are not supported as well.

Note: Pytorch models are supported indirectly by exporting the model in ONNX format.

Warning: Future versions of the toolkit will only support `.tflite` and `.onnx` formats.

6.2. Installing Docker

A few installation hints here below, please note that these are not a replacement for the official *Docker* documentation, for more details please refer to `https://docs.docker.com/get-docker/` .

6.2.1. Linux/Ubuntu

```
apt-get install docker.io
```

To be able to run docker without super user also run the two commands here below once after docker installation (for more info refer to https://docs.docker.com/engine/install/linux-postinstall/)

```
# Create the docker group if it doesn't already exist
sudo groupadd docker
# Add the current user "$USER" to the docker group
sudo usermod -aG docker $USER
```

6.2.2. MacOS - Docker

The easiest way to install Docker on MacOS is via the `brew` package manager. If you don't have it installed yet please follow the official *brew* website: https://brew.sh/ After brew is installed you can install Docker.

Important: On macOS the Docker GUI is not free for use for commercial application, a valid alternative is *Colima*.

```
brew install docker
```

See the note in the linux installation above to run docker without super user.

6.2.3. MacOS - Colima

Colima is a free container runtimes on macOS that can be used a replacement for Docker. (https://github.com/abiosoft/colima). It doesn't have a GUI but nevertheless is easy to install and configure.

```
brew install colima
mkdir -p ~/.docker/cli-plugins
brew install docker-Buildx
ln -sfn $(brew --prefix)/opt/docker-buildx/bin/docker-buildx ~/.docker/cli-plugins/docker-
↪buildx
colima start --vm-type vz --mount-type virtiofs --cpu 4 --memory 8 --disk 80
```

After the above commands, you can use *Colima* to work with Docker containers, the settings are stored in a config file `~/.colima/default/colima.yaml` and can be modified by editing the file if needed. Colima has to be started after each restart of the Mac:

```
colima start
```

6.2.4. Windows

The suggested way to run Docker on Windows is to install it inside a Linux Virtual Machine using *WSL2* available from Windows 10.

Important: Running Docker directy in Windows is incompatible with the presence of a VM. For this reason using a Linux VM in WSL2 is usually the best option.

WSL2 installation steps:

1. Run *Windows PowerShell* App as Administrator and execute the following command to install WSL2:

   ```
   > wsl --install
   ```

   When completed restart the computer.

2. Run *Windows PowerShell* App as before and install *Ubuntu-22.04*:

   ```
   > wsl --install -d Ubuntu-22.04
   ```

3. Run *Windows Terminal* App and select the *Ubuntu-22.04* distribution. From there install Docker and the *SyNAP* toolkit following the instructions in Section 6.2.1 above

For more information on WSL2 installation and setup please refer to the official Microsoft documentation: https://learn.microsoft.com/en-us/windows/wsl/install and https://learn.microsoft.com/en-us/windows/wsl/setup/environment

6.3. Installing SyNAP Tools

Before installing the SyNAP toolkit, please be sure that you have a working Docker installation. The simplest way to do this is to run the `hello-world` image:

```
$ docker run hello-world

Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
...
...
...
Hello from Docker!
This message shows that your installation appears to be working correctly.
...
...
```

If the above command doesn't produce the expected output please check the instructions in the previous section or refer to the official Docker documentation for your platform. If all is well you can proceed with the installation of the toolkit.

The SyNAP toolkit is distributed as a Docker image, to install it just download the image from the SyNAP github repository:

```
docker pull ghcr.io/syna-synap/toolkit:3.0.0
```

This image contains not only the conversion tool itself but also all the required dependencies and additional support utilities.

You can find the latest version of the toolkit in: https://github.com/syna-synap/toolkit/pkgs/container/toolkit

6.4. Running SyNAP Tools

Once Docker and the *SyNAP toolkit* image are installed, the model conversion tool can be executed directly inside a docker container. The source and converted models can be accessed on the host filesystem by mounting the corresponding directories when running the container. For this reason it is important to run the container using the same user/group that owns the files to be converted. To avoid manually specifying these options at each execution it's suggested to create a simple alias and add it to the user's startup file (e.g. `.bashrc` or `.zshrc`):

- `alias synap='docker run -i --rm -u $(id -u):$(id -g) -v $HOME:$HOME -w $(pwd) ghcr.io/syna-synap/toolkit:3.0.0'`

The options have the following meaning:

- **`-i`:** run the container interactively (required for commands that read data from *stdin*, such as `image_od`)
- **`--rm`:** remove the container when it exits (stopped containers are not needed anymore)
- **`-u $(id -u):$(id -g)`:** run the container as the current user (so files will have the correct access rights)
- **`-v $HOME:$HOME`:** mount the user's home directory so that its entire content is visible inside the container. If some models or data are located outside the home directory, additional directories can be mounted by repeating the `-v` option, for example add: `-v /mnt/data:/mnt/data`. It's important to specify the same path outside and inside the container so absolute paths work as expected.
- **`-w $(pwd)`:** set the working directory of container to the current directory, so that relative paths specified in the command line are resolved correctly

With the above alias, the desired *SyNAP* tool command line is just passed as a parameter, for example:

```
$ synap help

SyNAP Toolkit

Docker alias:
    alias synap='docker run -i --rm -u $(id -u):$(id -g) -v $HOME:$HOME -w $(pwd) \
                 ghcr.io/syna-synap/toolkit:3.0.0'
    Use multiple -v options if needed to mount additional directories eg: -v /mnt/dat:/mnt/dat

Usage:
    synap COMMAND ARGS
    Run 'synap COMMAND --help' for more information on a command.

Commands:
    convert           Convert and compile model
    help              Show help
    image_from_raw    Convert image file to raw format
    image_to_raw      Generate image file from raw format
    image_od          Superimpose object-detection boxes to an image
    version           Show version
```

Important: as already noted there is no need to be `root` to run docker. In case you get a *Permission Denied* error when executing the above command, please refer to Section 6.2.1

The toolkit provides a number of tools to convert and manipulate models and images.

Model conversion can be performed using the `convert` command. It takes in input:

- a network model
- the target HW for which to convert the model (e.g. VS680 or VS640)
- the name of the directory where to generate the converted model
- an optional yaml metafile that can be used to specify customized conversion options (mandatory for .pb models)

In output it generates three files:

- **model.synap** the converted network model
- **model_info.txt** additional information about the generated model for user reference, including:
  - input/output tensors attributes
  - subgraph splitting
  - layer table
  - operation table
  - memory usage
- **quantization_info.txt** additional quantization information (only if the model is quantized using the toolkit)

An additional `cache` directory is also generated to speedup future compilations of the same model.

Example:

```
$ synap convert --model mobilenet_v1_quant.tflite --target VS680 --out-dir mnv1
$ ls mnv1
model_info.txt  model.synap  cache
```

In the case of `Caffe` models the weights are not in the `.prototxt` file but stored in a separate file, generally with `.caffemodel` extension. This file has to be provided in input to the converter tool as well. Example:

```
$ synap convert --model mnist.prototxt --weights mnist.caffemodel --target VS680 --out-dir out
```

Important: The model file and the output directory specified must be inside or below a directory mounted inside the Docker container (see `-v` option in the `synap` alias above).

6.5. Conversion Metafile

When converting a model it is possible to provide a yaml metafile to customize the generated model, for example it is possible to specify:

- the data representation in memory (nhwc or nchw)
- model quantization options
- output dequantization
- input preprocessing options
- delegate to be used for inference (npu, gpu, cpu)

Example:

```
$ synap convert --model mobilenet_v1_quant.tflite --meta mobilenet.yaml \
  --target VS680 --out-dir mnv1
```

This metafile is mandatory when converting a Tensorflow `.pb` model. It can be completely omitted when converting a quantized `.tflite` model.

The best way to understand the content of a metafile is probably to first look at an example, here below the one for a typical *mobilenet_v1* model, followed by a detailed description of each field. Most of the fields are optional, mandatory fields are explicitly marked.

```yaml
delegate: npu

data_layout: nhwc

security:
    secure: true
    file: ../security.yaml

inputs:
  - name: input
    shape: [1, 224, 224, 3]
    means: [128, 128, 128]
    scale: 128
    format: rgb
    security: any
    preprocess:
        type: nv21
        size: [1920, 1080]
        crop: true

outputs:
  - name: MobilenetV1/Predictions/Reshape_1
    dequantize: false
    format: confidence_array

quantization:
    data_type: uint8
    scheme: default
    mode: standard
    algorithm: standard
    options:
    dataset:
      - ../../sample/*_224x224.jpg
```

- `delegate`

  Select the delegate to use for inference. Available delegates are:

  `default` (default, automatically select delegate according to the target HW)

  `npu`

  `gpu`

  `cpu`

  If not specified the default delegate for the target hardware is used. It is also possible to specify the delegate on a layer-by-layer basis. See section Section 6.8.

- `data_layout`

  The data layout in memory, allowed values are: `default`, `nchw` and `nhwc`.

  For Tensorflow and Tensorflow Lite models the default is `nhwc`. Forcing the converted model to be `nchw` might provide some performance advantage when the input data is already in this format since no additional data reorganization is needed.

  For Caffe and ONNX models the default is `nchw`. In this case it is not possible to force to `nhwc`.

- `input_format`

  Format of the input tensors. This is an optional string that will be attached as an attribute to all the network input tensors for which a “format” field has not been specified.

- `output_format`

  Format of the ouput tensors. This is an optional string that will be attached as an attribute to all the network ouput tensors for which a “format” field has not been specified.

- `security`

  This section contains security configuration for the model. If this section is not present, security is disabled. Security is only supported with the `npu` delegate.

  - `secure`

    If true enable security for the model. For secure models it is also possible to specify the security policy for each input and output. A secure model is encrypted and signed at conversion time so that its structure and weights will not be accessible and its authenticity can be verified. This is done by a set of keys and certificates files whose path is contained in a security file.

  - **`file`** Path to the security file. This is a `yaml` file with the following fields:

    ```
    encryption_key: <path-to-encryption-key-file>
    signature_key: <path-to-signature-key-file>
    model_certificate: <path-to-model-certificate-file>
    vendor_certificate: <path-to-vendor-certificate-file>
    ```

    Both relative and absolute paths can be used. Relative paths are considered relative to the location of the security file itself. The same fields can also be specified directly in the model metafile in place of the ‘file’ field. For detailed information on the security policies and how to generate and authenticate a secure model please refer to SyNAP_SyKURE.pdf

- **`inputs`** (pb)

  Must contain one entry for each input of the network. Each entry has the following fields:

- `name` (pb)

  Name of the input in the network graph. For `tflite` and `onnx` models this field is not required but can still be used to specify a different input layer than the default input of the network. This feature allows to convert just a subset of a network without having to manually edit the source model. For `.pb` models or when `name` is not specified the inputs must be in the same order as they appear in the model. When this field is specified the `shape` field is mandatory.

- `shape` (pb)

  Shape of the input tensor. This is a list of dimensions, the order is given by the layout of the input tensor in the model (even if a different layout is selected for the compiled model). The first dimension must represent by convention the number of samples *N* (also known as "batch size") and is ignored in the generated model which always works with a batch-size of 1. When this field is specified the `name` field is mandatory.

- `means`

  Used to normalize the range of input values. A list of mean values, one for each channel in the corresponding input. If a single value is specified instead of a list, it will be used for all the channels. If not specified a mean of `0` is assumed.

  The *i-th* channel of each input is normalized as: `norm = (in - means[i]) / scale`

  Normalization is necessary to bring the input values in the range used when the model has been trained. SyNAP does this computation in three occasions:

  * to normalize data from *image* quantization files when the network is quantized (note that this doesn't apply to *numpy* quantization files, in this case it is assumed that the numpy files have already been normalized)
  * to normalize input data at inference time in the NPU when the network is compiled with pre-processing enabled (see the `preprocess` option here below)
  * to normalize input data in SW when the network is compiled *without* preprocessing and input data is assigned using the `Tensor assign()` method in the SyNAP library

  Note: when converting an 8-bits pre-quantized model and no `means` and `scale` are specified they are automatically inferred from the quantization information under the assumption that the input is an 8-bits image. This allows to convert a pre-quantized model without having to explicitly specify the preprocessing information. In this case an unspecified mean and scale is not equivalent to specifying a scale of 1 and a mean of 0. To avoid any ambiguity it's suggested to always specify both means and scale explicitly.

- `scale`

  Used to normalize the range of input values. The scale is a single value for all the channels in the corresponding input. If not specified a scale of `1` is assumed. More details on normalization in the description of the `means` field here above.

- `format`

  Information about the type and organization of the data in the tensor. The content and meaning of this string is custom-defined, however SyNAP Toolkit and SyNAP `Preprocessor` recognize by convention an initial format type optionally followed by one or more named attributes:

  `<format-type> [<key>=value]...`

  Recognised types are:

  `rgb` (default): 8-bits RGB or RGBA or grayscale image

  `bgr`: 8-bits BGR image or BGRA or grayscale image

  Recognised attributes are:

`keep_proportions=1` (default): preserve aspect-ratio when resizing an image using `Preprocessor` or during quantization. `keep_proportions=0`: don't preserve aspect-ratio when resizing an image using `Preprocessor` or during quantization

Any additional attribute if present is ignored by SyNAP.

- `preprocess`

  Input preprocessing options for this input tensor. It can contain the following fields:

  * `type`: format of the input data (e.g. `rgb`, `nv12`) see the table below
  * `size`: size of the input image as a list [H, W]
  * `crop`: enable runtime cropping of the input image

  The meaning of each field is explained in detail in the preprocessing section here below. Preprocessing is only supported with the `npu` delegate.

- `security`

  Security policy for this input tensor. This field is only considered for secure models and can have the following values:

  `any` (default): the input can be either in secure or non-secure memory

  `secure`: the input must be in secure memory

  `non-secure`: the input must be in non-secure memory

- **`outputs`** (pb)

  Must contain one entry for each input of the network. Each entry has the following fields:

  - `name` (pb)

    Name of the output in the network graph. For `tflite` and `onnx` models this field is not required but can still be used to specify a different output layer than the default output of the network. This feature allows to convert just a subset of a network without having to manually edit the source model. For `.pb` and `.onnx` models or when `name` is not specified the outputs must be in the same order as they appear in the model.

  - `dequantize`

    The output of the network is internally dequantized and converted to `float`. This is more efficient then performing the conversion in software.

  - `format`

    Information about the type and organization of the data in the tensor. The content and meaning of this string is custom-defined, however SyNAP `Classifier` and `Detector` postprocessors recognize by convention an initial format type optionally followed by one or more named attributes:

    `<format-type> [<key>=value]...`

    All fields are separated by one or more spaces. No spaces allowed between the key and the value. Example:

    `confidence_array class_index_base=0`

    See the `Classifier` and `Detector` classes for a description of the specific attributes supported.

  - `security`

    Security policy for this output tensor. This field is only considered for secure models and can have the following values:

`secure-if-input-secure` (default): the output buffer must be in secure memory if at least one input is in secure memory

`any`: the output can be either in secure or non-secure memory

- **`quantization`** [q]

  Quantization options are required when quantizing a model during conversion, they are not needed when importing a model which is already quantized. Quantization is only supported with the `npu` delegate.

  - `data_type`

    Data type used to quantize the network. The same data type is used for both activation data and weights. Available data types are:

    `uint8` (default)

    `int8`

    `int16`

    `float16`

    Quantizing to 8 bits provides the best performance in terms of inference speed. Quantizing to `int16` can provide higher inference accuracy at the price of higher inference times. Interesting tradeoffs between speed and accuracy can be achieved using *mixed quantization*, that is specifying the data type on a layer-by-layer basis. See section Section 6.7.5.

  - `scheme`

    Select the quantization scheme. Available schemes are:

    `default` (default)

    `asymmetric_affine`

    `dynamic_fixed_point`

    `perchannel_symmetric_affine`

    Scheme `asymmetric_affine` is only supported for data types `int8` and `uint8`. Scheme `dynamic_fixed_point` is only supported for data types `int8` and `int16`. Scheme `perchannel_symmetric_affine` is only supported for data type `int8`. If the scheme is not specfied or set to `default`, if will be automatically selected according to the data type: `asymmetric_affine` will be used for `uint8`, `dynamic_fixed_point` for signed types `int8` and `int16`.

  - `mode`

    Select the quantization mode. Available modes are:

    `standard` (default)

    `full`

    The `standard` mode should be used most of the times. In this mode only the layer-types for which this makes sense are quantized. Other layer types where quantization is not helpful are left unchanged (e.g. layers which just change the layout of the data). The `full` mode forces the quantization of all layers. This can in some cases reduce the inference accuracy so should be used only when needed. One case where this is useful is for example when the standard quantization doesn't quantize the initial layer so that the input remains in float16 which would require data type conversion in software.

  - `algorithm`

    Select the quantization algorithm. Available algorithms are:

`standard` (default)

`kl_divergence`

`moving_average`

- `options`

  Special options for fine tuning the quantization in specific cases. Normally not needed.

- `dataset` [q]

  Quantization dataset(s), that it the set of input files to be used to quantize the model. In case of multi-input networks, it is necessary to specify one dataset per input. Each dataset will consist of the sample files to be applied to the corresponding input during quantization.

  A sample file can be provided in one of two forms:

  1. as an image file (`.jpg` or `.png`)
  2. as a NumPy file (`.npy`)

  Image files are suitable when the network inputs are images, that is 4-dimensional tensors (NCHW or NHWC). In this case the `means` and `scale` values specified for the corresponding input are applied to each input image before it is used to quantize the model. Furthermore each image is resized to fit the input tensor.

  NumPy files can be used for all kind of network inputs. A NumPy file shall contain an array of data with the same shape as the corresponding network input. In this case it is not possible to specify a `means` and `scale` for the input, any preprocessing if needed has to be done when the NumPy file is generated.

  To avoid having to manually list the files in the quantization dataset for each input, the quantization dataset is instead specified with a list of *glob expressions*, one glob expression for each input. This makes it very easy to specify as quantization dataset for one input the entire content of a directory, or a subset of it. For example all the *jpeg* files in directory *samples* can be indicated with:

  ```
  samples/*.jpg
  ```

  Both relative and absolute paths can be used. Relative paths are considered relative to the location of the metafile itself. It is not possible to specify a mix of image and `.npy` files for the same input. For more information on the glob specification syntax, please refer to the python documentation: https://docs.python.org/3/library/glob.html

  If the special keyword `random` is specified, a random data file will be automatically generated for this input. This option is useful for preliminary timing tests, but not for actual quantization.

  If this field is not specified, quantization is disabled.

Note: The fields marked [pb] are mandatory when converting `.pb` models. The fields marked [q] are mandatory when quantizing models.

Note: The metafile also supports limited variable expansion: `${ENV:name}` anywhere in the metafile is replaced with the content of the environment variable *name* (or with the empty string if the variable doesn't exist). `${FILE:name}` in a format string is replaced with the content of the corresponding file (the file path is relative to that of the conversion metafile itself). This feature should be used sparingly as it makes the metafile not self-contained.

6.6. Preprocessing

The size, layout, format and range of the data to be provided in the input tensor(s) of a network is defined when the network model is created and trained. For example a typical *mobilenet-v1 .tflite* model will expect an input image of size 224x224, with NHWC layout and channels organized in RGB order, with each pixel value normalized (rescaled) in the range [-1, 1].

Unfortunately, in real world usage, the image to be processed is rarely available in this exact format. For example the image may come from a camera in 1920x1080 YUV format. This image must then be converted to RGB, resized and normalized to match the expected input. Many libraries exist to perform this kind of conversion, but the problem is that these computations are quite compute-intensive, so even if deeply optimized, doing this on the CPU will often require more time than that required by the inference itself.

Another option is to retrain the network to accept in input the same data format that will be available at runtime. This option, while sometimes a good idea, also presents its own problems. For example it might not always be possible or practical to retrain a network, especially if the task has to be repeated for several input sizes and formats.

To simplify and speedup this task, SyNAP Toolkit allows to automatically insert input preprocessing code when a model is converted. This code is executed directly in the NPU and in some cases can be an order of magnitude faster than the equivalent operation in the CPU. An alternative to adding the preprocessing to the original model is to create a separate "preprocessing model" whose only purpose is to convert the input image to the desired format and size, and then execute the two models in sequence without any additional data copy, see *Buffer Sharing* This can be convenient if the original model is large and the input can come in a variety of possible formats. Preprocessing models for the most common cases already come preinstalled.

The available preprocessing options are designed for images and support 5 kinds of transformations:

- format conversion (e.g YUV to RGB, or RGB to BGR)
- cropping
- resize and downscale (without preserving proportions)
- normalization to the required value range (e.g. normalize [0, 255] to [-1, 1])
- data-type conversion (from uint8 to the data type of the network input layer, eg float16 or int16)

Preprocessing is enabled by specifying the `preprocess` section in the input specification in the *.yaml* file. This section contains the following fields (the fields marked (*) are mandatory). Note that the *mean* and *scale* used to normalize the input values don't appear here because they are the same used to quantize the model (see `means` and `scale` fields in the input specification).

6.6.1. *type*(*)

This field specifies the format of the input data that will be provided to the network. Only image formats are supported at the moment. The SyNAP toolkit will add the required operations to convert the input data to the `format` and layout expected by the network input tensor. If the `format` of the network input tensor is not specified, it is assumed to be `rgb` by default. If this field is set to the empty string or to "`none`", no preprocessing is applied.

Not all conversion are supported: `gray` input can only be used if the input tensor has 1 channel. All the other input formats except `float32` can only be used if the input tensor has 3 channels.

Some input formats generates multiple data inputs for one network tensor. For example if `nv12` is specified the converted network will have two inputs: the first for the `y` channel, the second for the `uv` channels. The preprocessing code will combine the data from these two inputs to feed the single `rgb` or `bgr` input tensor of the network.

The following table contains a summary of all the supported input formats and for each the properties and meaning of each generated input tensor. Note that the layout of the input data is always `NHWC` except for the

`rgb888-planar` and `float32` formats. In all cases *H* and *W* represent the height and width of the input image. If the size of the input image is not explicitly specified these are taken from the `H` and `W` of the network input tensor. In all cases each pixel component is represented with 8 bits.

The `float32` type is a bit special in the sense that in this case the input is not considered to be an 8-bits image but raw 32-bits floating point values which are converted to the actual data type of the tensor. For this reason any tensor shape is allowed and resizing via the `size` field is not supported.

Preprocessing Type	Input#	Shape	Format	Input Description
yuv444	0	NHW1	y8	Y component
	1	NHW1	u8	U component
	2	NHW1	v8	V component
yuv420	0	NHW1	y8	Y component
	1	N(H/2)(W/2)1	u8	U component
	2	N(H/2)(W/2)1	v8	V component
nv12	0	NHW1	y8	Y component
	1	N(H/2)(W/2)2	uv8	UV components interleaved
nv21	0	NHW1	y8	Y component
	1	N(H/2)(W/2)2	vu8	VU components interleaved
gray	0	NHW1	y8	Y component
rgb	0	NHW3	rgb	RGB components interleaved
bgra	0	NHW4	bgra	BGRA components interleaved
rgb888p	0	N3HW	rgb	RGB components planar
rgb888p3	0	NHW1	r8	Red component
	1	NHW1	g8	Green component
	2	NHW1	b8	Blue component
float32	0	any		Floating point data

Note: Specifying a *dummy* preprocessing (for example from `rgb` input to `rgb` tensor) can be a way to implement normalization and data-type conversion using the NPU HW instead of doing the same operations in SW.

6.6.2. `size`

This optional field allows to specify the size of the input image as a list containing the H and W dimensions in this order. Preprocessing will rescale the input image to the size of the corresponding input tensor of the network. The proportions of the input image are not preserved. If this field is not specified the *WxH* dimension of the input image will be the same as the W and H of the network tensor.

6.6.3. `crop`

Enable cropping. If specified, 4 additional scalar input tensors are added to the model. These inputs contain a single 32 bits integer each and are used to specify at runtime the dimension and origin of the cropping rectangle inside the input image. If security is enabled these additional inputs will have security attribute “any” so that it is always possible to specify the cropping coordinates from the user application even if the model and the other input / output tensors are secure. The cropping inputs are added after the original model input in the following order:

- width of the cropping rectangle
- height of the cropping rectangle
- left coordinate of the cropping rectangle
- top coordinate of the cropping rectangle

These inputs should be written using the `Tensor` scalar `assign()` method which accepts a value in pixels and converts it to the internal representation. Preprocessing will rescale the specified cropping rectangle to the size of the corresponding input tensor of the network. The proportions of the input image are not preserved. The area of the image outside the cropping rectangle is ignored. The cropping coordinates must be inside the dimension of the input image, oherwise the content of the resulting image is undefined.

6.7. Model Quantization

In order to efficiently run a model on the NPU HW it has to be *quantized*. Quantization consist of reducing the precision of the weights and activations of the model, so that computations can be done using 8-bits or 16-bits integer values, instead of the much more computationally intensive 32 bits floating point. A common side-effect of quantization is often to reduce the accuracy of the results, so it must be done with care.

There are three ways in which a model can be quantized:

- during training, using quantization-aware training features available in recent training framework such as Tensorflow and Pytorch. These techniques allow to compensate for the reduced precision induced by quantization during the training phase itself, thus providing in priciple better results.
- after training, using the same training framework, to convert a trained floating point model into a quantized one (e.g. convert the model to a quantized `uint8` `.tflite` model. The advantage of both these methods is that they benefit from advances in the quantization techniques in these frameworks and the generated model is still a standard model, so the effect of quantization can be tested and evaluated using standard tools.
- when converting the model using the SyNAP toolkit. This is the most convenient way to quantize models outside any traning framework and to take advantage of specific features of the SyNAP NPU and toolkit (e.g. 16-bits or mixed-type quantization).

In order to quantize a model it is necessary to determine an estimate of the range of the output values of each layer. This can be done by running the model on a set of sample input data and analyzing the resulting activations for each layer. To achieve a good quantization these sample inputs should be as representative as possible of the entire set of expected inputs. For example for a classification network the quantization dataset should contain at least one sample for each class. This would be the bare minimum, better quantization results can be achieved

by providing multiple samples for each class, for example in different conditions of size, color and orientation. In case of multi-input networks, each input must be fed with an appropriate sample at each inference.

6.7.1. Quantization Images Resize

The image files in the quantization dataset don't have to match the size of the input tensor. SyNAP toolkit automatically resizes each image to fit the input tensor. Starting from SyNAP 2.6.0 this transformation is done by preserving the aspect-ratio of the image content. If the image and the tensor have different aspect ratios, gray bands are added to the input image so that the actual content is not distorted. This corresponds to what is normally done at runtime and is important in order to achieve a reliable quantization. The aspect ratio is not preserved if the `format` string of the corresponding input contains the `keep_proportions=0` attribute: in this case the image is simply resized to fill the entire input tensor.

6.7.2. Data Normalizaton

When a model is trained the input data are often normalized in order to bring them to a range more suitable for training. It's quite common to bring them in a range [-1, 1] by subtracting the mean of the data distribution and dividing by the range (or standard deviation). A different mean value can be used for each channel.

In order to perform quantization correctly it is important to apply the same transformation to the input images or input samples used. If this is not done, the model will be quantized using a data distribution that is not the same as that used during training (and during inference) with poor results. This information has to be specified in the `means` and `scale` fields in the conversion metafile and will be applied to all input *image* files in the quantization dataset for the corresponding input using the formula:

```
norm = (in - means[channel]) / scale
```

For *data* (*.npy*`) files this is not done, it is assumed that they are already normalized.

In addition, the same transformation must also be applied at runtime on the input data when doing inference. If the model has been compiled with preprocessing enabled, data normalization is embedded in the model and will take place during inference inside the NPU. Otherwise data has to be normalized in SW. The `Tensor` class provides an `assign()` method that does exactly this, using the same `means` and `scale` fields specified in the conversion metafile (this method is smart enough to skip SW normalization when normalization is embedded in the model).

HW and SW normalization can be used interchangeably, and provide the same result. NPU normalization is generally somewhat faster, but this has to be checked case by case. In case of SW normalization, using the same mean for all the channels or using a mean of 0 and scale of 1 can in some cases improve performance: for example if affine quantization is used the normalization and quantization formula (`qval = (normalized_in + zero_point) * qscale`) can become one the inverse of the other thus resulting in a very efficient direct data copy.

The `Tensor::assign()` method is optimized to handle each case in the most efficient way possible. If needed this could be further improved by the customer by taking advantage of the ARM NEON SIMD instructions.

6.7.3. Quantization And Accuracy

As already noted quantizing a model, even if done correctly, will often result is some sort of accuracy loss when compared to the original floating point model. This effect can be reduced by quantizing the model to 16 bits, but the inference time will be higher. As a rule of thumb quantizing a model to 16 bits will double the inference time compared to the same model quantized to 8 bits.

The quantization errors introduced are not uniform across all the layers, they might be small for some layer and relevant for others. The *Quantization Entropy* is a measure of the error introduced in each layer.

A `quantizaton_entropy.txt` file can be generated by quantizing a model with the `kl_divergence` algorithm. This file will contain the quantization entropy for each weight and activation tensor in the network. It can be used as a guide to understand where errors are introduced in the network. Each entropy value is in the range [0, 1], the closer to 1 the higher the quantization error introduced. The `kl_divergence` algorithm is an iterative algorithm based on https://arxiv.org/pdf/1501.07681v1.pdf and tries to minimize the Kullback-Leibler divergence between the original and quantized outputs. It is slower than the standard algorithm but can produce more accurate results.

The quantization error for problematic layers can be reduced by keeping them in float16 or quantizing them to 16 bits integer using mixed quantization.

6.7.4. Per-Channel Quantization

SyNAP supports per-channel quantization by specifiying the `perchannel_symmetric_affine` quantization scheme. With this scheme weights scales are computed per-channel (each channel has its own scale), while activations will still have a single scale and bias for the entire tensor an in `asymmetric_affine` quantization. When weight values distribution changes a lot from one channel to the other, having a separate scale for each channel can provide a more accurate approximation of the original weights and so an improved inference accuracy

6.7.5. Mixed Quantization

Mixed quantization is a feature of the SyNAP toolkit that allows to choose the data type to be used for each layer when a network is quantized during conversion. This allows to achieve a custom balance between inference speed and accuracy.

Different approaches are possible:

- quantize the entire network to 16 bits and keep just the input in 8 bits. This provides the best accuracy possible and can be convenient when the input is an 8-bits image since it avoids the need to perform the 8-to-16 bits conversion is SW (note that this is not needed if preprocessing is used as it will also take care of the type conversion)
- quantize most of the network in 8 bits and just the *problematic* layers with `int16` or even `float16`. The quantization entropy can provide a guide to select the layers which would get more benefit from 16 bits. Note however that each change in data-type requires a conversion layer before and after it, so it is normally a good idea to avoid changing data-type too many times
- quantize the initial part (*backbone*) of the network in `uint8` and switch to `int16` for the last part (*head*). This is often a good choice when the input of the network is an 8-bits image, as networks should not be too sensitive in general to small noise in the input. Using 16 bits processing in the head allows to compute the final results (e.g. bounding boxes) with much greater precision without adding too much in term of inference time

To see how this is done let's consider the very simple model in Figure 4.

This model has one input and six convolutions. We've already seen how to compile it with uniform quantization, for example using 16 bits integers:

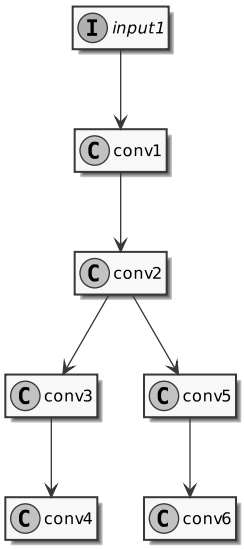

Figure 4. Sample Model

```
quantization:
    data_type: int16
```

Instead of a single type, the `data_type` field can contain an association map between layer-names and layer-types. Layer names are those that appear in the model to be converted, it's easy to see them using free tools such as *Netron*. So, the previous example is equivalent to:

```
quantization:
    data_type:
        input1: int16
        conv1: int16
        conv2: int16
        conv3: int16
        conv4: int16
        conv5: int16
        conv6: int16
```

To perform mixed-type quantization just select the desired type for each layer. The only limitation is that `uint8` and `int8` types can't be both present at the same time. For example we can choose to quantize the input and first convolution to 8 bits, the internal convolutions to 16 bits, and to keep the final convolutions in floating point:

```
quantization:
    data_type:
        input1: uint8
        conv1: uint8
        conv2: int16
        conv3: int16
        conv4: float16
        conv5: int16
        conv6: float16
```

Real models can often have well above one hundred layers, so writing an exhaustive list of all the layers can become confusing and error-prone. To keep the type specification simpler there are a few shortcuts that can be used. First of all, layers can be omitted: layers not explicitly listed will be quantized by default to `uint8`. Furthermore, some special conventions in the layer name specification can help:

- INPUTS : this special name is automatically expanded to the names of all the inputs of the network
- '*@layerId*' : a name preceded by the '@' suffix is interpreted as a *layerID* (see note below)
- *layername...* : a name followed by three dots, is expanded to the names of all the layers that *follows* the layer specified in the model (in execution order). Useful when for example we want to use the same data type for the head of the network or an entire branch.
- `'*'` : expanded to the names of all the layers that haven't been explicitly specified

The type specifications are applied in the order they are declared (except for '*') so it is possible to further override the type of layers already specified.

Note: During the compilation of a model several optimizations are applied and some layers in the original network may be fused together or optimized away completely. For optimized away layers it is of course not possible to specify the data type. For fused layers the issue is that they will not have the same name as the original layers. In this case it is possible to identify them by *layerId*: a *layerId* is a unique identifier assigned to each compiled layer. This is also a convenient way to identify layers in case the original model has layers with ambiguous or empty names. It is possible to see the list of all layerIDs for a compiled model in the generated `quantization_info.yaml` or `quantization_entropy.txt` file.

Lets's see a few examples applied to our sample network.

```
# Quantize input1 as int8, everything else as int16
quantization:
    data_type:
        INPUTS: int8
        '*': int16
```

```
# Quantize as uint8 but use int16 for conv3, conv4, conv5, conv6
quantization:
    data_type:
        '*': uint8
        conv2...: int16
```

```
# Quantize as uint8 but use int16 for conv3, conv4, conv6 but float16 for conv5
quantization:
    data_type:
        '*': uint8
        conv2...: int16
        conv5: float16
```

In the two examples above the specification `'*': uint8` could have been avoided since `uint8` is already the default, but helps in making the intention more explicit.

If we specify the data type for a layer that has been fused, we will get a "*Layer name*" error at conversion time. In this case we have to look for the *layerId* of the corresponding fused layer in `quantization_info.yaml` and use the "@" syntax as explained above. For example if in our sample model `conv5` and `conv6` have been fused, we will get an error if we specify the type for `conv5` alone. Looking in `quantization_info.yaml` we can find the ID of the fused layer, as in: `'@Conv_Conv_5_200_Conv_Conv_6_185:weight':`

We can then use this layer ID in the metafile to specify the data type of the fused layers:

```
# Quantize as uint8 but use int16 for conv3, conv4, conv6 but float16 for fused conv5+conv6
quantization:
    data_type:
        '*': uint8
        conv2...: int16
        '@Conv_Conv_5_200_Conv_Conv_6_185': float16
```

6.8. Heterogeneous Inference

In some cases it can be useful to execute different parts of a network on different hardware. For example consider an object detection network, where the initial part contains a bunch of convolutions and the final part some postprocessing layer such as *TFLite_Detection_PostProcess*. The NPU is heavily optimized for executing convolutions, but doesn't support the postprocessing layer, so the best approach would be to execute the initial part of the network on the NPU and the postprocessing on the CPU.

This can be achieved by specifying the delegate to be used on a per-layer basis, using the same syntax as we've seen for mixed quantization in section Section 6.7.5. For example, considering again the Model in Figure 4, we can specify that all layers should be executed on the NPU, except `conv5` and the layers that follows it which we want to execute on the GPU:

```
# Execute the entire model on the NPU, except conv5 and conv6
delegate:
    '*': npu
    conv5: gpu
    conv5...: gpu
```

Another advantage of distributing processing to different hardware delegates is that when the model is organized in multiple independent branches (so that a branch can be executed without having to wait for the result of another branch), and each is executed on a different HW unit then the branches can be executed in parallel.

In this way the overall inference time can be reduced to the time it takes to execute the slowest branch. Branch parallelization is always done automatically whenever possible.

Note: Branch parallelization should not be confused with in-layer parallelization, which is also always active when possible. In the example above the two branches *(conv3,conv4)* and *(conv5,conv6)* are executed in parallel, the former the NPU and the latter on the GPU. In addition, each convolution layer is parallelized internally by taking advantage of the parallelism available in the NPU and GPU HW.

6.9. Model Conversion Tutorial

Let's see how to convert and run a typical object-detection model.

1. Download the sample *ssd_mobilenet_v1_1_default_1.tflite* object-detection model:

   https://tfhub.dev/tensorflow/lite-model/ssd_mobilenet_v1/1/default/1

2. Create a conversion metafile `ssd_mobilenet.yaml` with the content here below (Important: be careful that newlines and formatting must be respected but they are lost when doing copy-paste from a pdf):

```
outputs:
- name: Squeeze
  dequantize: true
  format: tflite_detection_boxes y_scale=10 x_scale=10 h_scale=5 w_scale=5 anchors=$
↪{ANCHORS}
- name: convert_scores
  dequantize: true
  format: per_class_confidence class_index_base=-1
```

   A few notes on the content of this file:

   "`name: Squeeze`" and "`name: convert_scores`" explicitly specifiy the output tensors where we want model conversion to stop. The last layer (`TFLite_Detection_PostProcess`) is a custom layer not suitable for NPU acceleration, so it is implemented in software in the `Detector` postprocessor class.

   "`dequantize: true`" performs conversion from quantized to float directly in the NPU. This is much faster than doing conversion in software.

   "`tflite_detection_boxes`" and "`convert_scores`" represents the content and data organization in these tensors

   "`y_scale=10`" "`x_scale=10`" "`h_scale=5`" "`w_scale=5`" corresponds to the parameters in the `TFLite_Detection_PostProcess` layer in the network

   "`${ANCHORS}`" is replaced at conversion time with the `anchor` tensor from the `TFLite_Detection_PostProcess` layer. This is needed to be able to compute the bounding boxes during postprocessing.

   "`class_index_base=-1`" this model has been trained with an additional background class as index 0, so we subtract 1 from the class index during postprocessing to conform to the standard *coco* dataset labels.

3. **Convert the model (be sure that the model, meta and output dir are in a directory visible** in the container, see `-v` option in Section 6.4):

```
$ synap convert --model ssd_mobilenet_v1_1_default_1.tflite --meta ssd_mobilenet.
↪yaml --target VS680 --out-dir compiled"
```

4. Push the model to the board:

```
$ adb root
$ adb remount
$ adb shell mkdir /data/local/tmp/test
$ adb push compiled/model.synap /data/local/tmp/test
```

5. Execute the model:

```
$ adb shell
# cd /data/local/tmp/test
# synap_cli_od -m model.synap $MODELS/object_detection/coco/sample/sample001_640x480.jpg"

Input image: /vendor/firmware/.../sample/sample001_640x480.jpg (w = 640, h = 480, c = 3)
Detection time: 5.69 ms
#   Score  Class  Position  Size     Description
0   0.70       2  395,103    69, 34  car
1   0.68       2  156, 96    71, 43  car
2   0.64       1  195, 26   287,445  bicycle
3   0.64       2   96,102    18, 16  car
4   0.61       2   76,100    16, 17  car
5   0.53       2  471, 22   167,145  car
```

6.10. Model Profiling

When developing and optimizing a model it can be useful to understand how the execution time is distributed among the layers of the network. This provides an indication of which layers are executed efficiently and which instead represent bottlenecks.

In order to obtain this information the network has to be executed step by step so that each single timing can be measured. For this to be possible the network must be generated with additional profiling instructions by calling `synap_convert.py` with the `--profiling` option, for example:

```
$ synap convert --model mobilenet_v2_1.0_224_quant.tflite --target VS680 --profiling --out-
↪dir mobilenet_profiling
```

Note: Even if the execution time of each layer doesn't change between *normal* and *profiling* mode, the overall execution time of a network compiled with profiling enabled will be noticeably higher than that of the same network compiled without profiling, due to the fact that NPU execution has to be started and suspended several times to collect the profiling data. For this reason profiling should normally be disabled, and enabled only when needed for debugging purposes.

Note: When a model is converted using SyNAP toolkit, layers can be fused, replaced with equivalent operations and/or optimized away, hence it is generally not possible to find a one-to-one correspondence between the items in the profiling information and the nodes in the original network. For example adjacent convolution, ReLU and Pooling layer are fused together in a single *ConvolutionReluPoolingLayer* layer whenever possible. Despite these optimizations the correspondence is normally not too difficult to find. The layers shown in the profiling correspond to those listed in the *model_info.txt* file generated when the model is converted.

After each execution of a model compiled in profiling mode, the profiling information will be available in *sysfs*, see Section 5.4. Since this information is not persistent but goes away when the network is destroyed, the easiest way to collect it is by using *synap_cli* program. The `--profling <filename>` option allows to save a copy of the *sysfs network_profile* file to a specified file before the network is destroyed:

```
$ adb push mobilenet_profiling $MODELS/image_classification/imagenet/model/
$ adb shell
# cd $MODELS/image_classification/imagenet/model/mobilenet_profiling
# synap_cli -m model.synap --profiling mobilenet_profiling.txt random

# cat mobilenet_profiling.txt
pid: 21756, nid: 1, inference_count: 78, inference_time: 272430, inference_last: 3108, iobuf_
↪count: 2, iobuf_size: 151529, layers: 34
| lyr |   cycle | time_us | byte_rd | byte_wr | type
|   0 |  152005 |     202 |  151344 |       0 | TensorTranspose
|   1 |  181703 |     460 |    6912 |       0 | ConvolutionReluPoolingLayer2
```

(continues on next page)

(continued from previous page)

```
|   2 |    9319 |      51 |    1392 |       0 | ConvolutionReluPoolingLayer2
|   3 |   17426 |      51 |    1904 |       0 | ConvolutionReluPoolingLayer2
|   4 |   19701 |      51 |    1904 |       0 | ConvolutionReluPoolingLayer2
...
|  28 |   16157 |      52 |    7472 |       0 | ConvolutionReluPoolingLayer2
|  29 |  114557 |     410 |  110480 |       0 | FullyConnectedReluLayer
|  30 |  137091 |     201 |    2864 |    1024 | Softmax2Layer
|  31 |       0 |       0 |       0 |       0 | ConvolutionReluPoolingLayer2
|  32 |       0 |       0 |       0 |       0 | ConvolutionReluPoolingLayer2
|  33 |     670 |      52 |    1008 |       0 | ConvolutionReluPoolingLayer2
```

6.11. Compatibility With SyNAP 2.X

SyNAP 3.x is fully backward compatible with SyNAP 2.x.

- It is possible to execute models compiled with SyNAP 3.x toolkit with SyNAP 2.x runtime. The only limitation is that in this case heterogeneous compilation is not available and the entire model will be executed on the NPU. This can be done by specifying the `--out-format nb` option when converting the model. In this case the toolkit will generate in output the legacy `model.nb` and `model.json` files instead of the `model.synap` file:

```
$ synap convert --model mobilenet_v2_1.0_224_quant.tflite --target VS680 --out-format nb
↪--out-dir mobilenet_legacy
```

- It is possible to execute models compiled with SyNAP 2.x toolkit with SyNAP 3.x runtime
- SyNAP 3.x API is an extension of SyNAP 2.x API, so all the existing applications can be used without any modification

7. Framework API

The core functionality of the Synap framework is to execute a precompiled neural network. This is done via the `Network` class. The Network class has been designed to be simple to use in the most common cases while still being flexible enough for most advanced use-cases. The actual inference will take place on different HW units (NPU, GPU, CPU or a combination of them) according to how the model has been compiled.

7.1. Basic Usage

7.1.1. Network Class

The `Network` class is extremely simple, as shown in the picture here below.

There are just two things that can be done with a network:

- load a model, by providing the compiled model in `.synap` format
- execute an inference

A network also has an array of input tensors where to put the data to be processed, and an array of output tensors which will contain the result(s) after each inference.

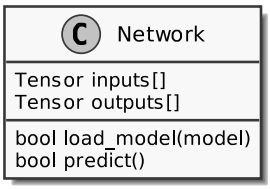

Figure 5. Network class

class `synaptics::synap::`**`Network`**
Load and execute a neural network on the NPU accelerator.

Public Functions

bool **`load_model`**(const std::string &model_file, const std::string &meta_file = "")
Load model.

In case another model was previously loaded it is disposed before loading the one specified.

Parameters

- **`model_file`** – path to .synap model file. Can also be the path to a legacy .nb model file.
- **`meta_file`** – for legacy .nb models must be the path to the model's metadata file (JSON-formatted). In all other cases must be the empty string.

Returns true if success

bool **`load_model`**(const void *model_data, size_t model_size, const char *meta_data = nullptr)
Load model.

In case another model was previously loaded it is disposed before loading the one specified.

Parameters

- **model_data** – model data, as from e.g. fread() of model.synap The caller retains ownership of the model data and can delete them at the end of this method.
- **model_size** – model size in bytes
- **meta_data** – for legacy .nb models must be the model's metadata (JSON-formatted). In all other cases must be nullptr.

Returns true if success

bool **predict()**
Run inference.

Input data to be processed are read from input tensor(s). Inference results are generated in output tensor(s).

Returns true if success, false if inference failed or network not correctly initialized.

Public Members

Tensors **inputs**
Collection of input tensors that can be accessed by index and iterated.

Tensors **outputs**
Collection of output tensors that can be accessed by index and iterated.

7.1.2. Using A Network

The prerequisite in order to execute a neural network is to create a *Network* object and load its model in `.synap` format. This file is generated when the network is converted using the Synap toolkit. This has to be done only once, after a network has been loaded it is ready to be used for inference:

1. put the input data in the Network input tensor(s)
2. call network `predict()` method
3. get the results from the Network input tensor(s)

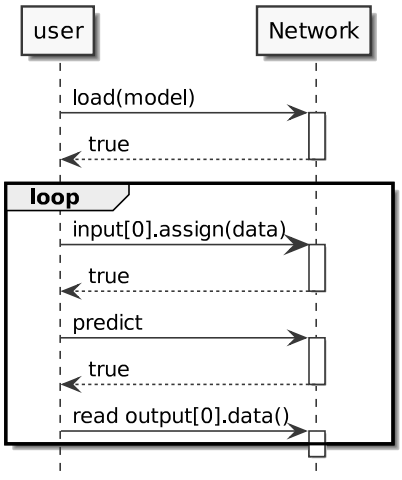

Figure 6. Running inference

Example:

```
Network net;
net.load_model("model.synap");
vector<uint8_t> in_data = custom_read_input_data();
net.inputs[0].assign(in_data.data(), in_data.size());
net.predict();
custom_process_result(net.outputs[0].as_float(), net.outputs[0].item_count());
```

Please note that:

- all memory allocations and alignment for the weights and the input/ouput data are done automatically by the Network object
- all memory is automatically deallocated when the Network object is destroyed
- for simplicity all error checking has been omitted, methods typically return `false` if something goes wrong. No explicit error code is returned since the error can often be too complex to be explained with a simple enum code and normally there is not much the caller code can do do to recover the situation. More detailed information on what went wrong can be found in the logs.
- the routines named *custom...* are just placeholders for user code in the example.
- In the code above there is a data copy when assigning the `in_data` vector to the tensor. The data contained in the `in_data` vector can't be used directly for inference because there is no guarantee that they are correcly aligned and padded as required by the HW. In most cases the cost of this extra copy is negligible, when this is not the case the copy can sometimes be avoided by writing directly inside the tensor data buffer, something like:

  ```
  custom_generate_input_data(net.inputs[0].data(), net.inputs[0].size());
  net.predict();
  ```

  For more details see section Section 7.3.1
- the type of the data in a tensor depends on how the network has been generated, common data types are *float16*, *float32* and quantized *uint8* and *int16*. The `assign()` and `as_float()` methods take care of all the required data conversions.

By using just the simple methods shown in this section it is possible to perform inference with the NPU hardware accelerator. This is almost all that one needs to know in order to use SyNAP in most applications. The following sections explain more details of what's going on behind the scenes: this allows to take full advantage of the available HW for more demanding use-cases.

7.2. Advanced Topics

7.2.1. Tensors

We've seen in the previous section that all accesses to the network input and output data are done via tensor objects, so it's worth looking in detail at what a `Tensor` object can do. Basically a tensor allows to:

- get information and attributes about the contained data
- access data
- access the underlying `Buffer` used to contain data. More on this in the next section.

Let's see a detailed description of the class and the available methods.

class `synaptics::synap::Tensor`
: Synap data tensor.

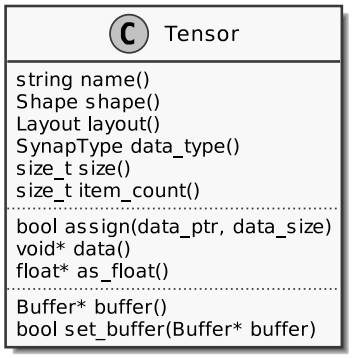

Figure 7. Tensor class

It's not possible to create tensors outside a *Network*, users can only access tensors created by the *Network* itself.

Public Functions

const std::string &**name**() const
: Get the name of the tensor.

  Can be useful in case of networks with multiple inputs or outputs to identify a tensor with a string instead of a positional index.

  Returns tensor name

const Shape &**shape**() const
: Get the Shape of the *Tensor*, that is the number of elements in each dimension.

  The order of the dimensions is specified by the tensor layout.

  Returns tensor shape

const Dimensions **dimensions**() const
: Get the Dimensions of the *Tensor*, that is the number of elements in each dimension.

  The returned values are independent of the tensor layout.

  Returns tensor dimensions (all 0s if the rank of the tensor is not 4)

Layout **layout**() const
: Get the layout of the *Tensor*, that is how data are organized in memory.

  SyNAP supports two layouts: `NCHW` and `NHWC`. The *N* dimension (number of samples) is present for compatibility with standard conventions but must always be one.

  Returns tensor layout

std::string **format**() const
: Get the format of the *Tensor*, that is a description of what the data represents.

  This is a free-format string whose meaning is application dependent, for example "rgb", "bgr".

  Returns tensor format

DataType **data_type**() const
: Get tensor data type.

The integral types are used to represent quantized data. The details of the quantization parameters and quantization scheme are not directly available, an user can use quantized data by converting them to 32-bits *float* using the *as_float()* method below

Returns the type of each item in the tensor.

Security **security**() const
Get tensor security attribute.

Returns security attribute of the tensor (none if the model is not secure).

size_t **size**() const

Returns size *in bytes* of the tensor data

size_t **item_count**() const
Get the number of items in the tensor.

A tensor *size()* is alwas equal to *item_count()* multiplied by the size of the tensor data type.

Returns number of data items in the tensor

bool **is_scalar**() const

Returns true if this is a scalar tensor, that is it contains a single element. (the shape of a scalar tensor has one dimension, equal to 1)

bool **assign**(const uint8_t *data, size_t count)
Normalize and copy data to the tensor data buffer.

The data is normalized and converted to the type and quantization scheme of the tensor. The data count must be equal to the *item_count()* of the tensor.

Parameters

- **data** – pointer to data to be copied
- **count** – number of data items to be copied

Returns true if success

bool **assign**(const int16_t *data, size_t count)
Normalize and copy data to the tensor data buffer.

The data is normalized and converted to the type and quantization scheme of the tensor. The data count must be equal to the *item_count()* of the tensor.

Parameters

- **data** – pointer to data to be copied
- **count** – number of data items to be copied

Returns true if success

bool **assign**(const float *data, size_t count)
Normalize and copy data to the tensor data buffer.

The data is normalized and converted to the type and quantization scheme of the tensor. The data count must be equal to the *item_count()* of the tensor.

Parameters

- **data** – pointer to data to be copied
- **count** – number of data items to be copied

Returns true if success

bool **assign**(const void *data, size_t size)
Copy raw data to the tensor data buffer.

The data is considered as raw data so no normalization or conversion is done whatever the actual data-type of the tensor. The data size must be equal to the *size()* of the tensor.

Parameters

- **data** – pointer to data to be copied
- **size** – size of data to be copied

Returns true if success

bool **assign**(const *Tensor* &src)
Copy the content of a tensor to the tensor data buffer.

No normalization or conversion is done, the data type and size of the two tensors must match

Parameters **src** – source tensor containing the data to be copied.

Returns true if success, false if type or size mismatch

bool **assign**(int32_t value)
Writes a value to the tensor data buffer.

Only works if the tensor is a scalar. The value is converted to the tensor data type: 8, 16 or 32 bits integer. Before writing in the data buffer the value is also rescaled if needed as specified in the tensor format attributes.

Parameters **value** – value to be copied

Returns true if success

template<typename **T**>
T ***data**()
Get a pointer to the beginning of the data inside the tensor data buffer if it can be accessed directly.

This is the case only if T matches the *data_type()* of the tensor and no normalization/quantization is required (or normalization and quantization simplify-out each other). Sample usage:

```
uint8_t* data8 = tensor.data<uint8_t>();
```

Returns pointer to the data inside the data buffer or nullptr.

void ***data**()
Get a pointer to the beginning of the data inside the tensor data buffer, if any.

The method returns a `void` pointer since the actual data type is what returned by the *data_type()* method.

Returns pointer to the raw data inside the data buffer, nullptr if none.

const float ***as_float**() const
Get a pointer to the tensor content converted to float.

The method always returns a `float` pointer. If the actual data type of the tensor is not float, the conversion is performed interally, so the user doesn't have to care about how the data are internally represented.

Please note that this is a pointer to floating point data inside the tensor itself: this means that the returned pointer must *not* be freed, memory will be released automatically when the tensor is destroyed.

Returns pointer to float[*item_count()*] array representing tensor content converted to float (nullptr if tensor has no data)

Buffer ***buffer()**
Get pointer to the tensor's current data *Buffer* if any.

This will be the default buffer of the tensor unless the user has assigned a different buffer using *set_buffer()*

Returns current data buffer, or nullptr if none

bool **set_buffer**(*Buffer* *buffer)
Set the tensor's current data buffer.

The buffer size must be 0 or match the tensor size otherwise it will be rejected (empty buffers are automatically resized to the the tensor size). Normally the provided buffer should live at least as long as the tensor itself. If the buffer object is destroyed before the tensor, it will be automatically unset and the tensor will remain buffer-less.

Parameters **buffer** – buffer to be used for this tensor. The buffer size must match the tensor size (or be 0).

Returns true if success

struct **Private**

Here below a list of all the data types supported in a tensor:

enum class `synaptics::synap::`**DataType**
Values:

enumerator **invalid**

enumerator **byte**

enumerator **int8**

enumerator **uint8**

enumerator **int16**

enumerator **uint16**

enumerator **int32**

enumerator **uint32**

enumerator **float16**

enumerator **float32**

7.2.2. Buffers

The memory used to store a tensor data has to satisfy the following requirements:

- must be correctly aligned
- must be correctly padded
- in some cases must be contiguous
- must be accessible by the NPU HW accelerator and by the CPU or other HW components

Memory allocated with `malloc()` or `new` or `std::vector` doesn't satisfy these requirements so can't be used directly as input or output of a Network. For this reason `Tensor` objects use a special `Buffer` class to handle memory. Each tensor internally contains a default Buffer object to handle the memory used for the data.

The API provided by the `Buffer` is similar when possible to the one provided by `std::vector`. The main notable exeception is that a buffer content can't be indexed since a buffer is just a container for raw memory, without a *data type*. The data type is known by the tensor which is using the buffer. `Buffer` is also taking care of disposing the allocated memory when it is destroyed (*RAII*) to avoid all possible memory leakages. The actual memory allocation is done via an additional `Allocator` object. This allows to allocate memory with different attributes in different memory area. When a buffer object is created it will use the default allocator unless a different allocator is specified. The allocator can be specified directly in the constructor or later using the `set_allocator()` method.

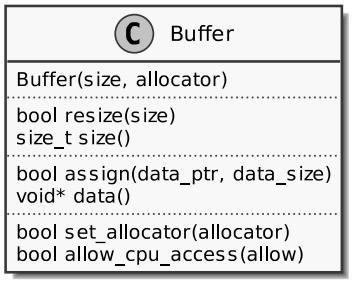

Figure 8. Buffer class

In order for the buffer data to be shared by the CPU and NPU hardware some extra operations have to be done to ensure that the CPU caches and system memory are correcly aligned. All this is done automatically when the buffer content is used in the Network for inference. There are cases when the CPU is not going to read/write the buffer data directly, for example when the data is generated by another HW component (eg. video decoder). In these cases it's possible to have some performance improvements by disabling CPU access to the buffer using the method provided.

Note: it is also possible to create a buffer that refers to an existing memory area instead of using an allocator. In this case the memory area must have been registered with the TrustZone kernel and must be correctly aligned and padded. Furthermore the Buffer object will *not* free the memory area when it is destroyed, since the memory is supposed to be owned by the SW module which allocated it.

class `synaptics::synap::Buffer`
Synap data buffer.

Public Functions

`Buffer`(Allocator *allocator = nullptr)
Create an empty data buffer.

Parameters **`allocator`** – allocator to be used (default is malloc-based)

`Buffer`(size_t size, Allocator *allocator = nullptr)
Create and allocate a data buffer.

Parameters

- **`size`** – buffer size
- **`allocator`** – allocator to be used (default is malloc-based)

`Buffer`(uint32_t mem_id, size_t offset, size_t size)
Create a data buffer to refer to an existing memory area.

The user must make sure that the provided memory is correctly aligned and padded. The specified memory area will *not* be deallocated when the buffer is destroyed. It is the responsiblity of the caller to release mem_id *after* the *Buffer* has been destroyed.

Parameters

- **mem_id** – id of an existing memory area registered with the TZ kernel.
- **offset** – offset of the actual data inside the memory area
- **size** – size of the actual data

Buffer(uint32_t handle, size_t offset, size_t size, bool is_mem_id)
Create a data buffer to refer to an existing memory area.

The user must make sure that the provided memory is correctly aligned and padded. The specified memory area will *not* be deallocated when the buffer is destroyed. It is the responsiblity of the caller to release mem_id *after* the *Buffer* has been destroyed.

Parameters

- **handle** – fd of an existing dmabuf or mem_id registered with the TZ kernel.
- **offset** – offset of the actual data inside the memory area
- **size** – size of the actual data
- **is_mem_id** – true if the first argument is a mem_id, false if it is a fd

Buffer(const *Buffer* &rhs, size_t offset, size_t size)
Create a data buffer that refers to a part of the memory area of an existing buffer.

The memory of the provided buffer must have already been allocated. To avoid referring to released memory, the existing buffer memory must *not* be deallocated before this buffer is destroyed.

Parameters

- **rhs** – an existing *Buffer*
- **offset** – offset of the desired data inside the *Buffer* memory area
- **size** – size of the desired data

Buffer(*Buffer* &&rhs) noexcept
Move constructor (only possible from buffers not yet in use by a *Network*)

Buffer &**operator=**(*Buffer* &&rhs) noexcept
Move assignment (only possible for buffers not yet in use by a *Network*)

bool **resize**(size_t size)
Resize buffer.

Only possible if an allocator was provided. Any previous content is lost.

Parameters **size** – new buffer size

Returns true if success

bool **assign**(const void *data, size_t size)
Copy data in buffer.

Always successful if the input data size is the same as current buffer size, otherwise the buffer is resized if possible.

Parameters

- **data** – pointer to data to be copied
- **size** – size of data to be copied

Returns true if all right

size_t **size**() const
Actual data size.

const void ***data**() const
Actual data.

bool **allow_cpu_access**(bool allow)
Enable/disable the possibility for the CPU to read/write the buffer data.

By default CPU access to data is enabled. CPU access can be disabled in case the CPU doesn't need to read or write the buffer data and can provide some performance improvements when the data is only generated/used by another HW components.

Note: reading or writing buffer data while CPU access is disabled might cause loss or corruption of the data in the buffer.

Parameters **allow** – false to indicate the CPU will not access buffer data

Returns current setting

bool **set_allocator**(Allocator *allocator)
Change the allocator.

Can only be done if the buffer is empty.

Parameters **allocator** – allocator

Returns true if success

7.2.3. Allocators

Two allocators are provided for use with buffer objects:

- the *standard* allocator: this is the default allocator used by buffers created without explicitly specifying an allocator. The memory is paged (non-contiguous).
- the *cma* allocator: allocates contiguous memory. Contiguous memory is required for some HW components and can provide some small performance improvement if the input/output buffers are very large since less overhead is required to handle memory pages. Should be used with great care since the contiguous memory available in the system is quite limited.

Allocator ***std_allocator**()
return a pointer to the system standard allocator.

Allocator ***cma_allocator**()
return a pointer to the system contiguous allocator.

Important: The calls above return pointers to global objects, so they *must NOT be deleted* after use

---

7.3. Advanced Examples

7.3.1. Accessing Tensor Data

Data in a Tensor is normally written using the `Tensor::assign(const T* data, size_t count)` method. This method will take care of any required data normalization and data type conversion from the type `T` to the internal representation used by the network.

Similarly the output data are normally read using the `Tensor::as_float()` method that provides a pointer to the tensor data converted to floating point values from whatever internal represention is used.

These conversions, even if quite optimized, present however a runtime cost that is proportional to the size of the data. For input data this cost could be avoided by generating them directly in the Tensor data buffer, but this is only possible when the tensor data type corresponds to that of the data available in input and no additional normalization/quantization is required. Tensor provides a type-safe `data<T>()` access method that will return a pointer to the data in the tensor only if the above conditions are satisfied, for example:

```
uint8_t* data_ptr = net.inputs[0].data<uint8_t>();
if (data_ptr) {
   custom_generate_data(data_ptr, net.inputs[0].item_count());
}
```

If the data in the tensor is not `uint8_t` or normalization/[de]quantization is required, the returned value will be `nullptr`. In this case the direct write or read is not possible and `assign()` or `as_float()` is required.

It's always possible to access the data directly by using the raw `data()` access method which bypasses all checks:

```
void* in_data_ptr = net.inputs[0].data();
void* out_data_ptr = net.outputs[0].data();
```

In the same way it's also possible to assign raw data (without any conversion) by using `void*` data pointer:

```
const void* in_raw_data_ptr = ....;
net.inputs[0].assign(in_raw_data_ptr, size);
```

In these cases it's responsibility of the user to know how the data are represented and how to handle them.

7.3.2. Setting Buffers

If the properties of the default tensor buffer are not suitable, the user can explicity create a new buffer and use it instead of the default one. For example suppose we want to use a buffer with contiguous memory:

```
Network net;
net.load_model("model.synap");

// Replace the default buffer with one using contiguous memory
Buffer cma_buffer(net.inputs[0].size(), cma_allocator());
net.inputs[0].set_buffer(&cma_buffer);

// Do inference as usual
custom_generate_input_data(net.inputs[0].data(), net.inputs[0].size());
net.predict();
```

7.3.3. Settings Default Buffer Properties

A simpler alternative to replacing the buffer used in a tensor as seen in the previous section is to directly change the properties of the default tensor buffer. This can only be done at the beginning, before the tensor data is accessed:

```
Network net;
net.load_model("model.synap");

// Use contiguous allocator for default buffer in input[0]
net.inputs[0].buffer()->set_allocator(cma_allocator());

// Do inference as usual
custom_generate_input_data(net.inputs[0].data(), net.inputs[0].size());
net.predict();
```

7.3.4. Buffer Sharing

The same buffer can be shared among multiple networks if they need to process the same input data. This avoids the need of redundant data copies:

```
Network net1;
net1.load_model("nbg1.synap");
Network net2;
net2.load_model("nbg2.synap");

// Use a common input buffer for the two networks (assume same input size)
Buffer in_buffer;
net1.inputs[0].set_buffer(&in_buffer);
net2.inputs[0].set_buffer(&in_buffer);

// Do inference as usual
custom_generate_input_data(in_buffer.data(), in_buffer.size());
net1.predict();
net2.predict();
```

Another interesting case of buffer sharing is when the output of a network must be processed directly by another network. For example the first network can do some preprocessing and the second one the actual inference. In this case setting the output buffer of the first network as the input buffer of the second network allows to completely avoid data copying (the two tensors must have the same size of course). Furthermore, since the CPU has no need to access this intermediate data, it is convenient to disable its access to this buffer, this will avoid the un-necessary overhead of cache flushing and provide an additional improvement in performance.

```
Network net1;
net1.load_model("nbg1.synap");
Network net2;
net2.load_model("nbg2.synap");

// Use net1 output as net2 input. Disable CPU access for better performance.
net1.outputs[0].buffer()->allow_cpu_access(false);
net2.inputs[0].set_buffer(net1.outputs[0].buffer()));

// Do inference as usual
custom_generate_input_data(net1.inputs[0].data(), net1.inputs[0].size());
net1.predict();
net2.predict();
```

One last case is when the output of the first network is smaller than the input of the second network, and we still want to avoid copy. Imagine for example that the output of net1 is an image 640x360 that we want to generate inside the input of net2 which expects an image 640x480. In this case the buffer sharing technique shown above can't work due to the mismatch in size of the two tensors. What we need instead is to share part of the memory used by the two Buffers.

```
Network net2;  // Important: this has to be declared first, so it is destroyed after net1
net2.load_model("nbg2.synap");
Network net1;
net1.load_model("nbg1.synap");

// Initialize the entire destination tensor now that we still have CPU access to it
memset(net2.inputs[0].data(), 0, net2.inputs[0].size());

// Replace net1 output buffer with a new one using (part of) the memory of net2 input buffer
*net1.outputs[0].buffer() = Buffer(*net2.inputs[0].buffer(), 0, net2.outputs[0].size());

// Disable CPU access for better performance
net1.outputs[0].buffer()->allow_cpu_access(false);
net2.inputs[0].buffer()->allow_cpu_access(false);

// Do inference as usual
custom_generate_input_data(net1.inputs[0].data(), net1.inputs[0].size());
net1.predict();
net2.predict();
```

Note: Since net1 input tensor now uses the memory allocated by net2, it is important that net1 is destroyed before net2, otherwise it will be left pointing to unallocated memory. This limitation will be fixed in the next release.

7.3.5. Recycling Buffers

It is possible for the user to explicitly set at any time which buffer to use for each tensor in a network. The cost of this operation is very low compared to the creation of a new buffer so it is possible to change the buffer associated to a tensor at each inference if desired.

Despite this, the cost of *creating* a buffer and setting it to a tensor the first time is quite high since it involves multiple memory allocations and validations. It is possible but deprecated to create a new `Buffer` at each inference, better to create the required buffers in advance and then just use `set_buffer()` to choose which one to use.

As an example consider a case where we want to do inference on the current data while at the same time preparing the next data. The following code shows how this can be done:

```
Network net;
net.load_model("model.synap");

// Create two input buffers
const size_t input_size = net.inputs[0].size();
vector<Buffer> buffers { Buffer(input_size), Buffer(input_size) };

int current = 0;
custom_start_generating_input_data(&buffers[current]);
while(true) {
    custom_wait_for_input_data();

    // Do inference on current data while filling the other buffer
```

(continues on next page)

(continued from previous page)

```
    net.inputs[0].set_buffer(&buffers[current]);
    current = !current;
    custom_start_generating_input_data(&buffers[current]);
    net.predict();
    custom_process_result(net.outputs[0]);
}
```

7.3.6. Using BufferCache

There are situations where the data to be processed comes form other components that provide each time a data block taken from a fixed pool of blocks. Each block can be uniquely identified by an ID or by an address. This is the case for example of a video pipeline providing frames.

Processing in this case should proceed as follows:

1. Get the next block to be processed
2. If this is the first time we see this block, create a new `Buffer` for it and add it to a collection
3. Get the `Buffer` corresponding to this block from the collection
4. Set is as the current buffer for the input tensor
5. Do inference and process the result

The collection is needed to avoid the expensive operation of creating a new `Buffer` each time. This is not complicate to code but steps 2 and 3 are always the same. The `BufferCache` template takes care of all this. The template parameter allows to specify the type to be used to identify the received block, this can be for example a BlockID or directly the address of the memory area.

Note: In this case the buffer memory is not allocated by the `Buffer` object. The user is responsible for ensuring that all data is properly padded and aligned. Futhermore the buffer cache is not taking ownership of the data block, it's responsibility of the user to deallocate them in due time after the `BufferCache` has been deleted.

7.3.7. Copying And Moving

`Network`, `Tensor` and `Buffer` objects internally access to HW resources so can't be copied. For example:

```
Network net1;
net1.load_model("model.synap");
Network net2;
net2 = net1;  // ERROR, copying networks is not allowed
```

However `Network` and `Buffer` objects can be moved since this has no overhead and can be convenient when the point of creation is not the point of use. Example:

```
Network my_create_network(string nb_name, string meta_name) {
    Network net;
    net.load(nb_name, meta_name);
    return net;
}

void main() {
    Network network = my_create_network("model.synap");
    ...
}
```

The same functionality is not available for `Tensor` objects, they can exist only inside their own Network.

7.4. NPU Locking

An application can decide to reserve the NPU for its exclusive usage. This can be useful in case of realtime applications that have strict requirements in terms of latency, for example video or audio stream processing.

Locking the NPU can be done at two levels:

1. reserve NPU access to the current process using `Npu::lock()`
2. reserve NPU for *offline use only* (that is disable NPU access from NNAPI)

7.4.1. NPU Locking

The NPU locking is done **by process**, this means that once the `Npu::lock()` API is called no other process will be able to run inference on the NPU. Other processes will still be able to load networks, but if they try to do offline or online NNAPI inference or to `lock()` the NPU again, they will fail.

The process which has locked the NPU is the only one which has the righs to unlock it. If a process with a different PID tries to unlock() the NPU, the operation will be ignored and have no effect.

Note: There is currently no way for a process to test if the NPU has been locked by some other process. The only possibility is to try to lock() the NPU, if this operation fails it means that the NPU is already locked by another process or unavailable due to some failure.

Note: If the process owning the NPU lock terminates or is terminated for any reason, the lock is automatically released.

7.4.2. NNAPI Locking

A process can reserve the NPU for offline use only so that nobody will be able to run *online* inference on the NPU via NNAPI. Other processes will still be able to run *offline* inference on the NPU. SyNAP has no dedicated API for this, NNAPI can be disabled by setting the property `vendor.NNAPI_SYNAP_DISABLE` to `1` using the standard Android API `__system_property_set()` or `android::base::SetProperty()`. Sample code in: https://android.googlesource.com/platform/system/core/+/master/toolbox/setprop.cpp

See also: *Disabling NPU Usage From NNAPI*

Note: It will still be possible to perform online inference on the NPU using the *timvx* tflite delegate.

7.4.3. Description

The `Npu` class controls the locking and unlocking of the NPU. Normally only one object of this class needs to be created when the applicaton start and destroyed when the application is going to terminate.

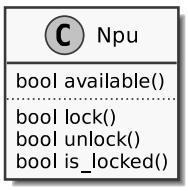

Figure 9. Npu class

class `synaptics::synap::`**`Npu`**
Reserve NPU usage.

Public Functions

bool **available**() const

Returns true if NPU successfully initialized.

bool **lock**()
Lock exclusive right to perform inference for the current process.

All other processes attemping to execute inference will fail, including those using NNAPI. The lock will stay active until *unlock()* is called or the *Npu* object is deleted.

Returns true if NPU successfully locked, false if NPU unavailable or locked by another process. Calling this method on an *Npu* object that is already locked has no effect, just returns true

bool **unlock**()
Release exclusive right to perform inference.

Returns true if success. Calling this method on an *Npu* object that is not locked has no effect, just returns true

bool **is_locked**() const

Note: the only way to test if the NPU is locked by someone else is to try to *lock()* it.

Returns true if we currently own the NPU lock.

struct **Private**
Npu private implementation.

Note: The Npu class uses the *RAII* technique, this means that when an object of this class is destroyed and it was locking the NPU, the NPU is automatically unlocked. This helps ensure that when a program terminates the NPU is in all cases unlocked.

7.4.4. Sample Usage

The following diagrams show some example use of the NPU locking API.

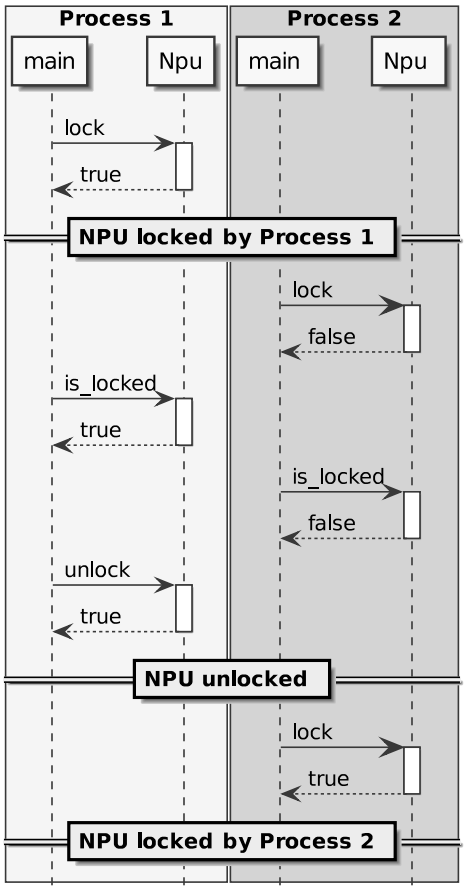

Figure 10. Locking the NPU

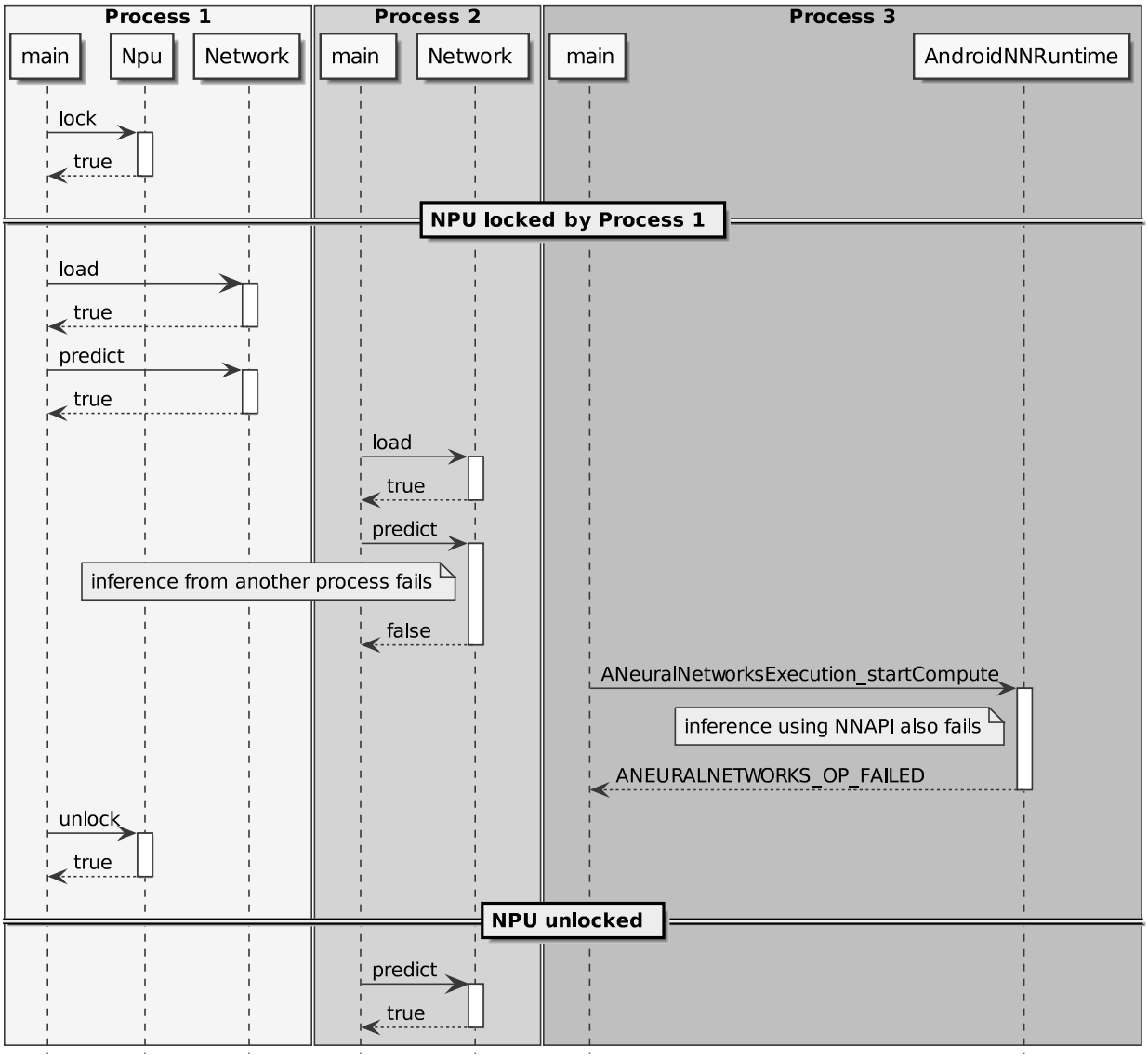

Figure 11. Locking and inference

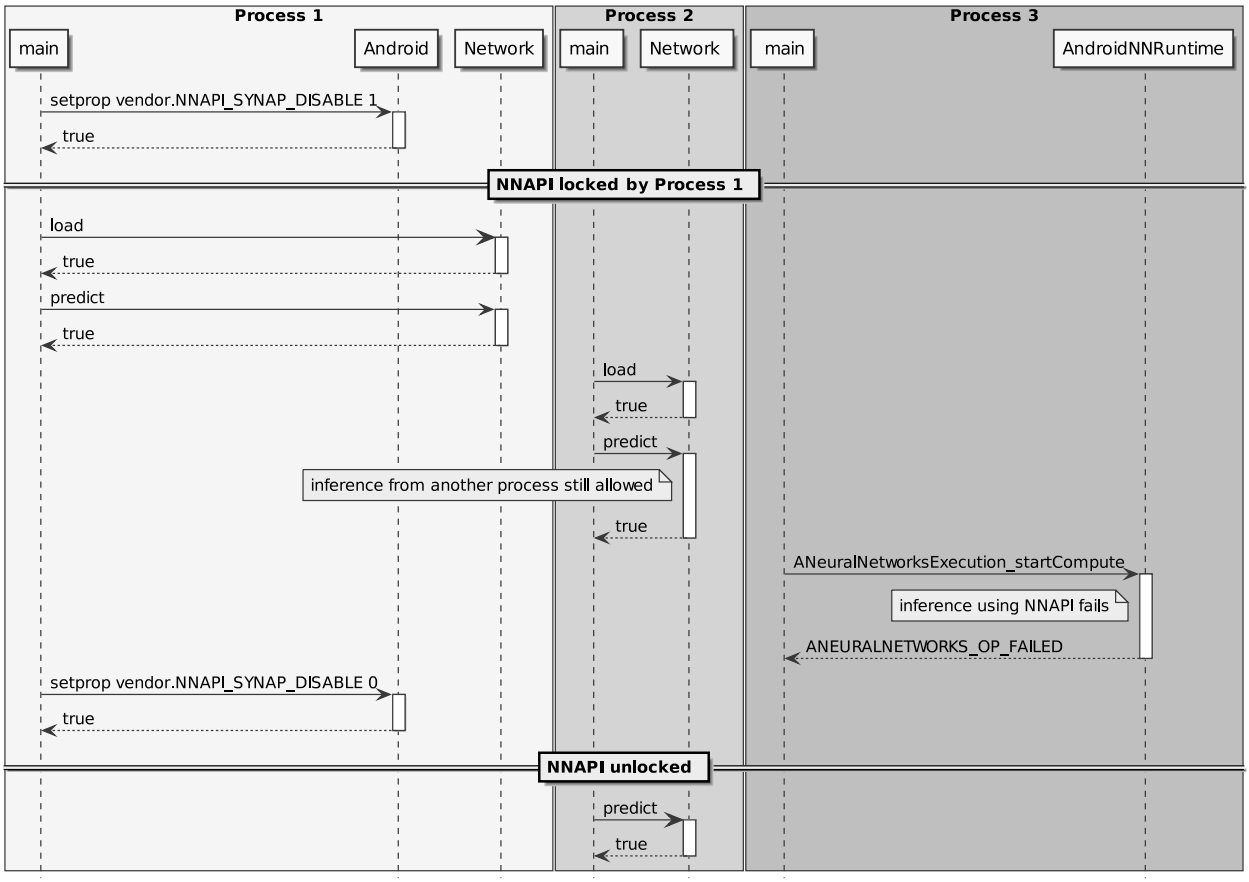

Figure 12. Locking NNAPI

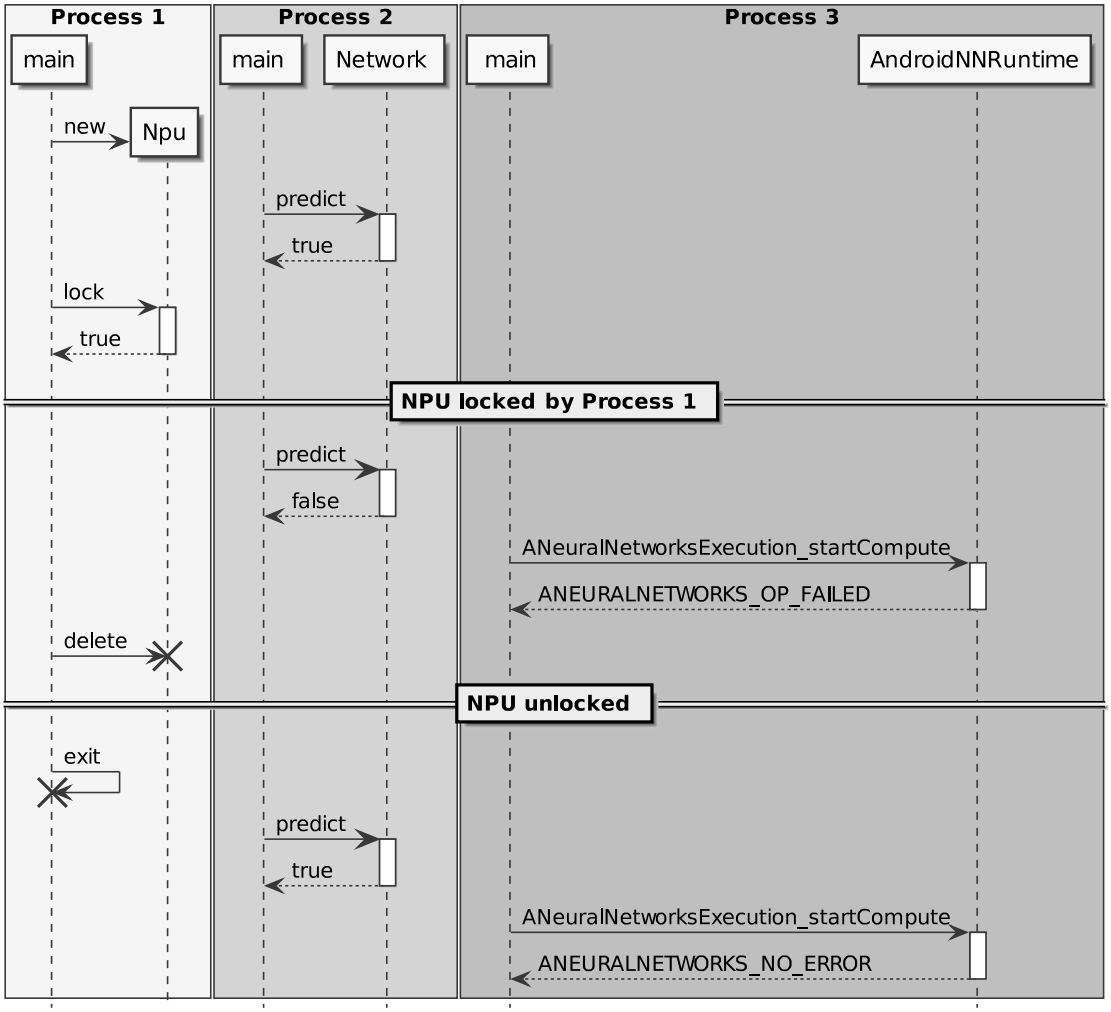

Figure 13. Automatic lock release

7.5. Preprocessing And Postprocessing

When using neural networks the input and output data are rarely used in their raw format. Most often data conversion has to be performed on the input data in order make them match the format expected by the network, this step is called *preprocessing*.

Example of preprocessing in the case of an image are:

- scale and/or crop the input image the the size expected by the network
- convert planar to interleaved or vice-versa
- convert RGB to BGR or vicerversa
- apply mean and scale normalization

These operations can be performed using the NPU at inference time by enabling preprocessing when the model is converted using the SyNAP Toolkit, or they can be performed in SW when the data is assigned to the Network.

Similarly the inference results contained in the network output tensor(s) normally require further processing to make the result usable. This step is called *postprocessing*. In some cases postprocessing can be a non-trivial step both in complexity and computation time.

Example of post-processing are:

- convert quantized data to floating point representation
- analyze the network output to extract the most significant elements
- combine the data from multiple output tensor to obtain a meaningful result

The classes in this section are not part of the SyNAP API, they are intented mainly as utility classes that can help writing SyNAP applications by combining the three usual steps of preprocess-inference-postprocess just explained.

Full source code is provided, so they can be used as a reference implementation for the user to extend.

7.5.1. InputData Class

The main role of the `InputData` class is to wrap the actual input data and complement it with additional information to specify what the data represents and how it is organized. The current implementation is mainly focused on image data.

`InputData` functionality includes:

- reading raw files (binary)
- reading and parsing images (jpeg or png) from file or memory
- getting image attributes, e.g. dimensions and layout.

The input filename is specified directly in the constructor and can't be changed. In alternative to a filename it is also possible to specify a memory address in case the content is already available in memory.

Note: No data conversion is performed, even for jpeg or png images the data is kept in its original form.

Example:

```
Network net;
net.load_model("model.synap");
InputData image("sample_rgb_image.dat");
net.inputs[0].assign(image.data(), image.size());
net.predict();
custom_process_result(net.outputs[0]);
```

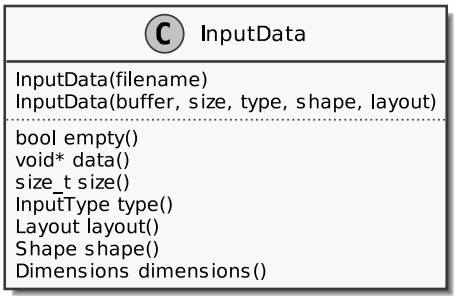

Figure 14. InputData class

7.5.2. Preprocessor Class

This class takes in input an `InputData` object and assigns its content to the input Tensor(s) of a network by performing all the necessary conversions. The conversion(s) required are determined automatically by reading the attributes of the tensor itself.

Supported conversions include:

- image decoding (jpeg, png or nv21 to rgb)
- layout conversion: *nchw* to *nhwc* or vice-versa
- format conversion: *rgb* to *bgr* or *grayscale*
- image cropping (if preprocessing with cropping enabled in the compiled model)
- image rescaling to fit the tensor dimensions

The conversion (if needed) is performed when an `InputData` object is assigned to a `Tensor`.

Cropping is only performed if enabled in the compiled model and the multi-tensor assign API is used: `Preprocessor::assign(Tensors& ts, const InputData& data)`.

Rescaling by default preserves the aspect ratio of the input image. If the destination tensor is taller than the rescaled input image, gray bands are added at the top and bottom. If the destination tensor is wider than then the rescaled input image, gray bands are added at the left and right. It is possible to configure the gray level of the fill using the `fill_color=N` option in the format string of the input tensor, where *N* is an integer between 0 (black) and 255 (white).

The preservation of the aspect-ratio can be disabled by specifying the `keep_proportions=0` option in the format string of the input tensor. In this case the input image is simply resized to match the size of the tensor.

Note: The `Preprocessor` class performs preprocessing using the CPU. If the conversion to be done is known in advance it may be convenient to perform it using the NPU by adding a preprocessing layer when the network is converted, see *Preprocessing*

7.5.3. ImagePostprocessor Class

`ImagePostprocessor` functionality includes:

- reading the content of a set of Tensors
- converting the raw content of the Tensors to a standard representation (currently only `nv21` is supported) The format of the raw content is determined automatically by reading the attributes of the tensors themselves. For example in some super-resolution network, the different component of the output image (*y*, *uv*) are provided in separate outputs. The converted data is made available in a standard vector.

Example:

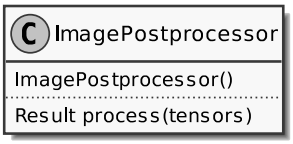

Figure 15. ImagePostprocessor class

```
Preprocessor preprocessor
Network net;
ImagePostprocessor postprocessor;

net.load_model("model.synap");
InputData image("sample_image.jpg");
preprocessor.assign(net.inputs[0], image);
net.predict();
// Convert to nv21
ImagePostprocessor::Result out_image = postprocessor.process(net.outputs);
binary_file_write("out_file.nv21", out_image.data.data(), out_image.data.size());
```

7.5.4. Classifier Class

The `Classifier` class is a postprocessor for the common use case of image classification networks.

There are just two things that can be done with a classifier:

- initialize it
- process network outputs: this will return a list of possible classifications sorted in order of decreasing confidence, each containing the following information:
  - class_index
  - confidence

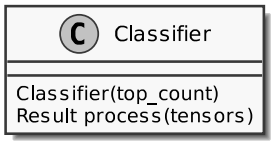

Figure 16. Classifier class

class `synaptics::synap::`**`Classifier`**
: Classification post-processor for *Network* output tensors.

  Determine the top-N classifications of an image.

Public Functions

inline **Classifier**(size_t top_count = 1)
Constructor.

Parameters **top_count** – number of most probable classifications to return

Result **process**(const Tensors &tensors)
Perform classification on network output tensors.

Parameters **tensors** – output tensors of the network tensors[0] is expected to contain a list of confidences, one for each image class

Returns classification results

struct **Result**
Classification result.

Public Members

bool **success** = {}
True if classification successful, false if failed.

std::vector<*Item*> **items**
List of possible classifications for the input, sorted in descending confidence order, that is items[0] is the classification with the highest confidence.

Empty if classification failed.

struct **Item**
Classification item.

Public Members

int32_t **class_index**
Index of the class.

float **confidence**
Confidence of the classification, normally in the range [0, 1].

Example:

```
Preprocessor preprocessor
Network net;
Classifier classifier(5);
net.load_model("model.synap");
InputData image("sample_image.jpg");
preprocessor.assign(net.inputs[0], image);
net.predict();
Classifier::Result top5 = classifier.process(net.outputs);
```

The standard content of the output tensor of a classification network is a list of probabilities, one for each class on which the model has been trained (possibly including an initial element to indicate a “background” or “unrecognized” class). In some cases the final *SoftMax* layer of the model is cut away to improve inference time: in this case the output values can’t be interpreted as probabilities anymore but since *SoftMax* is monotonic this

doesn't change the result of the classification. The postprocessing can be parametrized using the *format* field of the corresponding output in the conversion metafile (see *Conversion Metafile*):

Format Type	Out#	Shape	Description
confidence_array	0	NxC	List of probabilities, one per class

Attribute	Default	Description
class_index_base	0	Class index corresponding to the first element of the output vector

Where:

- N: Number of samples, must be 1
- C: number of recognized classes

7.5.5. Detector Class

The `Detector` class is a postprocessor for the common use case of object detection networks. Here *object* is a generic term that can refer to actual objects or people or anything used to train the network.

There are just two things that can be done with a detector:

- initialize it
- run a detection: this will return a list of detection items, each containing the following information:
  - class_index
  - confidence
  - bounding box
  - landmarks (optional)

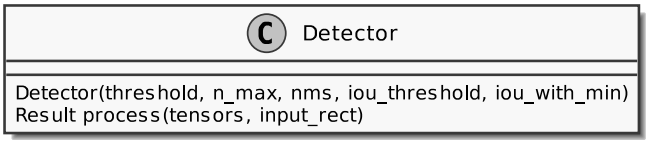

Figure 17. Detector class

class `synaptics::synap::`**`Detector`**
Object-detector.

The output format of object-detection networks is not always the same but depends on the network architecture used. The format type must be specified in the *format* field of the output tensor in the network metafile when the network is compiled. This following formats are currently supported: “retinanet_boxes”, “tflite_detection_boxes”, “yolov5”

Public Functions

`Detector``(`float score_threshold = 0.5, int n_max = 0, bool nms = true, float iou_threshold = .5, bool iou_with_min = false`)`
Constructor.

Parameters

- **`score_threshold`** – detections below this score are discarded
- **`n_max`** – max number of detections (0: all)
- **`nms`** – if true apply non-max-suppression to remove duplicate detections
- **`iou_threshold`** – intersection-over-union threshold (used if nms is true)
- **`iou_with_min`** – use min area instead of union to compute intersection-over-union

bool **`init`**`(`const Tensors &tensors`)`
Initialize detector.

If not called the detector is automatically initialized the 1st time *process()* is called.

Parameters **`tensors`** – output tensors of the network (after the network has been loaded)

Returns true if success

Result **process**(const Tensors &tensors, const Rect &input_rect)
Perform detection on network output tensors.

Parameters

- **tensors** – output tensors of the network
- **input_rect** – coordinates of the (sub)image provided in input (to compute bounding-boxes)

Returns detection results

class **Impl**
Subclassed by DetectorBoxesScores, DetectorTfliteODPostprocessOut, DetectorYoloBase, DetectorYolov5Pyramid

struct **Result**
Object-detector result.

Public Members

bool **success** = {}
True if detection successful, false if detection failed.

std::vector<*Item*> **items**
One entry for each detection.

Empty if nothing detected or detection failed.

struct **Item**
Detection item.

Public Members

int32_t **class_index**
Index of the object class.

float **confidence**
Confidence of the detection in the range [0, 1].

Rect **bounding_box**
Top,left corner plus horizontal and vertical size (in pixels)

std::vector<Landmark> **landmarks**
One entry for each landmark.

Empty if no landmark available.

Example:

```
Preprocessor preprocessor
Network net;
Detector detector;
net.load_model("model.synap");
```

(continues on next page)

(continued from previous page)

```
InputData image("sample_image.jpg");
Rect image_rect;
preprocessor.assign(net.inputs[0], image, &image_rect);
net.predict();
Detector::Result objects = detector.process(net.outputs, image_rect);
```

The rectangle argument passed to the `process()` method is needed so that is can compute bounding boxes and landmarks in coordinates relative to the original image, even if the image has been resized and/or cropped during the assignment to the network input tensor.

Postprocessing consist of the following steps:

- for each possible position in the input grid compute the score of the highest class there
- if this score is too low nothing is detected at that position
- if above the detection threshold then there is something there, so compute the actual bounding box of the object by combining information about the anchors location, the regressed deltas from the network and the actual size of the input image
- once all the detections have been computed, filter them using Non-Min-Suppression algorithm to discard spurious overlapping detections and keep only the one with highest score at each position. The NMS filter applies only for bounding boxes which have an overlap above a minimum threshold. The overlap itself is computed using the *IntersectionOverUnion* formula (ref: https://medium.com/analytics-vidhya/iou-intersection-over-union-705a39e7acef). In order to provide more filtering for boxes of different sizes, the "intersection" area is sometimes replaced by the "minimum" area in the computation. SyNAP Detector implements both formula.

The content of the output tensor(s) from an object detection network is not standardized. Several formats exist for the major families of detection networks, with variants inside each family. The information contained is basically always the same, what changes is the way they are organized. The `Detector` class currently supports the following output formats:

- `retinanet_boxes`
- `tflite_detection_input`
- `tflite_detection`
- `yolov5`
- `yolov8`

The desired label from the above list has to be put in the "format" field of the first output tensor of the network in the conversion metafile (see *Conversion Metafile*) so the `Detector` knows how to interpret the output.

`retinanet_boxes` is the output format used by Synaptics sample detection networks (mobilenet224_full80 for COCO detection and mobilenet224_full1 for people detection).

`tflite_detection_input` is the format of the input tensors of the `TFLite_Detection_PostProcess` layer, used for example in the *ssd_mobilenet_v1_1_default_1.tflite* object-detection model:

https://tfhub.dev/tensorflow/lite-model/ssd_mobilenet_v1/1/default/1

This format is used when the `TFLite_Detection_PostProcess` layer is removed from the network at conversion time and the corresponding postprocessing algorithm is performed in software.

In both cases above the model has two output tensors: the first one is a regression tensor, and contains the bounding box deltas for the highest-score detected object in each position of the input grid. The second one is the classification tensor and for each class contains the score of that class, that is the confidence that this class is contained in the corresponding position of the input grid.

`tflite_detection` is the format of the *output* tensors of the `TFLite_Detection_PostProcess` layer, used for example in the *ssd_mobilenet_v1_1_default_1.tflite* object-detection model.

`yolov5` is the output format used by models derived from the well-kown *yolov5* arcitecture. In this case the model has a single output 3D tensor organized as a list of detections, where each detection contains the following fields:

- bounding box deltas (x, y, w, h)
- overall confidence for this detection
- landmarks deltas (x, y) if supported by the model
- confidence vector, one entry per class

`yolov8` is the output format used by models derived from the *yolov8* arcitecture, the most recent update to the *yolo* family. The organization of the output tensor is very similar to that for yolov5 here above, the only difference is that the *overall confidence* field is missing.

In some cases the final layers in the model can be executed more efficiently in the CPU, so they are cut away when the model is generated or compiled with the SyNAP Toolkit. In this case the network will have one output tensor for each item of the image pyramid (normally 3) and each output will be a 4D or 5D tensor, whose layout depends on where exacly the model has been cut.

SyNAP Detector is able to automatically deduce the layout used, it just requires an indication if the information in the tensor are transposed.

Format Type	Out#	Shape	Description	Notes
retinanet_boxes	0	Nx4	bounding box deltas	
	1	NxC	Per-class probability	
tflite_detection_input tflite_detection_boxes	0	Nx4	bounding box deltas	
	1	NxC	Per-class probability	
tflite_detection	0	NxMx4	Bounding boxes	
	1	NxM	Index of detected class	
	2	NxM	Score of detected class	
	3	1	Actual number of detections	
yolov5	0..P-1	NxTxD	Processing done in the model	
		NxHxWxAxD	One 5D tensor per pyramid element	
		NxHxWx(A*D)	One 4D tensor per pyramid element	
		NxAxHxWxD	One 5D tensor per pyramid element	
		NxAxDxHxW	One 5D tensor per pyramid element	Requires “`transposed=1`”
		Nx(A*D)xHxW	One 4D tensor per pyramid element	Requires “`transposed=1`”
yolov8	0	NxTxD	Processing done in the model	Overall confidence missing

Where:

- N: number of samples, must be 1

- C: number of classes detected
- T: total number of detections
- M: maximum number of detections
- D: detection size (includes: bounding box deltas xywh, confidence, landmarks, per-class confidences)
- A: number of anchors
- H: heigth of the image in the pyramid
- W: width of the image in the pyramid
- P: number of images in the pyramid

Attributes for `retinanet_boxes` and `tflite_detection_input` formats:

Attribute	Default	Description
class_index_base	0	Class index corresponding to the first element of the output vector
transposed	0	Must be 1 if the output tensor uses the transposed format
anchors		Anchor points
x_scale	10	See `x_scale` parameter in the `TFLite_Detection_PostProcess` layer
y_scale	10	See `y_scale` parameter in the `TFLite_Detection_PostProcess` layer
h_scale	5	See `h_scale` parameter in the `TFLite_Detection_PostProcess` layer
w_scale	5	See `w_scale` parameter in the `TFLite_Detection_PostProcess` layer

In this case, the anchor points can be defined using the build-in variable `${ANCHORS}`:

```
anchors=${ANCHORS}
```

This variable is replaced at conversion time with the content of the `anchor` tensor from the `TFLite_Detection_PostProcess` layer (if present in the model).

Attributes for `tflite_detection` format:

Attribute	Default	Description
class_index_base	0	Class index corresponding to the first element of the output vector
h_scale	0	Vertical scale of the detected boxes (normally the H of the input tensor)
w_scale	0	Horizontal scale of the detected boxes (normally the W of the input tensor)

Attributes for `yolov5` and `yolov8` formats:

Attribute	Default	Description
class_index_base	0	Class index corresponding to the first element of the output vector
transposed	0	Must be 1 if the output tensor uses the transposed format
landmarks	0	Number of landmark points
anchors		Anchor points. Not needed if processing done in the model
h_scale	0	Vertical scale of the detected boxes
		(normally the H of the input tensor when processing is done in the model)
w_scale	0	Horizontal scale of the detected boxes
		(normally the W of the input tensor when processing is done in the model)
bb_normalized	0	Must be 1 if the bounding box deltas are normalized (only for *yolov8* `)
		Indicates that bounding boxes are normalized to the range [0, 1]
		while landmarks are in the range h_scale, wscale

For `yolov5` format, the `anchors` attribute must contain one entry for each pyramid element from P0, where each entry is a list of the `x,y` anchor deltas. For example for yolov5s-face, the anchors are defined in https://github.com/deepcam-cn/yolov5-face/blob/master/models/yolov5s.yaml :

```
- [4,5,  8,10,  13,16]  # P3/8
- [23,29,  43,55,  73,105]  # P4/16
- [146,217,  231,300,  335,433]  # P5/32
```

The corresponding outputs in the metafile can be defined as follows:

```
outputs:
  - format: yolov5 landmarks=5 anchors=[[],[],[],[4,5,8,10,13,16],[23,29,43,55,73,105],[146,
↪217,231,300,335,433]]
    dequantize: true
  - dequantize: true
  - dequantize: true
```

7.6. Building Sample Code

The source code of the sample applications (e.g. `synap_cli`, `synap_cli_ic`, etc) is included in the SyNAP release, together with that of the SyNAP libraries. Users based on the *ASTRA* distribution can build SyNAP using the provided Yocto recipe.

For other users building SyNAP code requires the following components installed:

1. VSSDK tree
2. cmake

Build steps:

```
cd synap/src
mkdir build
cd build
cmake -DVSSDK_DIR=/path/to/vssdk-directory -DCMAKE_INSTALL_PREFIX=install ..
make install
```

The above steps will create the binaries for the sample applications in `synap/src/build/install/bin`. The binaries can then be pushed to the board using `adb`:

```
cd synap/src/build/install/bin
adb push synap_cli_ic /vendor/bin
```

Users are free to change the source code provided to adapt it to their specific requirements.

8. Neural Network Processing Unit Operator Support

This section summarizes neural network operators supported by the SyNAP VS6x0/SL16x0 class of NPUs and accompanying software stack. For each operator type, the supported tensor types and execution engines is also documented. Designing networks that maximise the use of operators executed in the NN core will provide the best performances.

Table 8. Legend

Acronym	Description
NN	Neural Network Engine
PPU	Parellel Processing Unit
TP	Tensor Processor
asym-u8	asymmetric affine uint8
asym-i8	asymmetric affine int8
pc-sym-i8	per channel symmetric int8
fp32	floating point 32 bits
fp16	floating point 16 bits
h	half
int16	int16
int32	int32

Note: int16 dynamic fixed point convolution is supported by NN Engine in their multiplication. Other layers follow the tables, if asym-u8 in not available in NN column, int16 is also not available.

8.1. Basic Operations

	Tensor Types			Execution Engine		
Operator	**Input**	**Kernel**	**Output**	**NN**	**TP**	**PPU**
CONV2D	asym-u8	asym-u8	asym-u8	✓		
	asym-i8	pc-sym-i8	asym-i8	✓		✓
	fp32	fp32	fp32			✓
	fp16	fp16	fp16			✓
CONV1D	asym-u8	asym-u8	asym-u8	✓		
	asym-i8	pc-sym-i8	asym-i8	✓		✓
	fp32	fp32	fp32			✓
	fp16	fp16	fp16			✓
DECONVOLUTION	asym-u8	asym-u8	asym-u8	✓		
	asym-i8	pc-sym-i8	asym-i8	✓		✓
	fp32	fp32	fp32			✓
	fp16	fp16	fp16			✓
DECONVOLUTION1D	asym-u8	asym-u8	asym-u8	✓		
	asym-i8	pc-sym-i8	asym-i8	✓		✓
	fp32	fp32	fp32			✓
	fp16	fp16	fp16			✓
GROUPED_CONV2D	asym-u8	asym-u8	asym-u8	✓		
	asym-i8	pc-sym-i8	asym-i8	✓		✓
	fp32	fp32	fp32			✓
	fp16	fp16	fp16			✓

Note: convolutions are executed in the NN emgine only if they satisfy the following conditions: **stride == 1**, **kernel_size <= 15x15**, **dilation size + kernel size <= 15x15** If any of these conditions is not satisfied, the convolution will require support of the TP core and will run considerably slower

	Tensor Types			Execution Engine		
Operator	**Input**	**Kernel**	**Output**	**NN**	**TP**	**PPU**
FULLY_CONNECTED	asym-u8	asym-u8	asym-u8		✓	
	asym-i8	pc-sym-i8	asym-i8		✓	
	fp32	fp32	fp32			✓

8.2. Activation Operations

	Tensor Types		Execution Engine		
Operator	**Input**	**Output**	**NN**	**TP**	**PPU**
ELU	asym-u8	asym-u8		✓	✓
	asym-i8	asym-i8		✓	✓
	fp32	fp32		✓	✓
	fp16	fp16		✓	✓
HARD_SIGMOID	asym-u8	asym-u8		✓	✓
	asym-i8	asym-i8		✓	✓
	fp32	fp32		✓	✓
	fp16	fp16		✓	✓
SWISH	asym-u8	asym-u8		✓	
	asym-i8	asym-i8		✓	
	fp32	fp32			✓
	fp16	fp16		✓	
LEAKY_RELU	asym-u8	asym-u8		✓	
	asym-i8	asym-i8		✓	
	fp32	fp32			✓
	fp16	fp16		✓	
PRELU	asym-u8	asym-u8		✓	
	asym-i8	asym-i8		✓	
	fp32	fp32			✓
	fp16	fp16		✓	
RELU	asym-u8	asym-u8		✓	
	asym-i8	asym-i8		✓	
	fp32	fp32			✓
	fp16	fp16		✓	

	Tensor Types		Execution Engine		
Operator	**Input**	**Output**	**NN**	**TP**	**PPU**
RELUN	asym-u8	asym-u8		✓	
	asym-i8	asym-i8		✓	
	fp32	fp32			✓
	fp16	fp16		✓	
RSQRT	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓
SIGMOID	asym-u8	asym-u8		✓	
	asym-i8	asym-i8		✓	
	fp32	fp32			✓
	fp16	fp16		✓	
SOFTRELU	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓
SQRT	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓
TANH	asym-u8	asym-u8		✓	
	asym-i8	asym-i8		✓	
	fp32	fp32			✓
	fp16	fp16		✓	

	Tensor Types		Execution Engine		
Operator	**Input**	**Output**	**NN**	**TP**	**PPU**
ABS	asym-u8	asym-u8		✓	
	asym-i8	asym-i8		✓	
	fp32	fp32			✓
	fp16	fp16		✓	
CLIP	asym-u8	asym-u8		✓	✓
	asym-i8	asym-i8		✓	✓
	fp32	fp32			✓
	fp16	fp16		✓	✓
EXP	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓
LOG	asym-u8	asym-u8		✓	✓
	asym-i8	asym-i8		✓	✓
	fp32	fp32			✓
	fp16	fp16		✓	✓
NEG	asym-u8	asym-u8		✓	✓
	asym-i8	asym-i8		✓	✓
	fp32	fp32			✓
	fp16	fp16		✓	✓
MISH	asym-u8	asym-u8		✓	✓
	asym-i8	asym-i8		✓	✓
	fp32	fp32			✓
	fp16	fp16		✓	✓

	Tensor Types		Execution Engine		
Operator	**Input**	**Output**	**NN**	**TP**	**PPU**
SOFTMAX	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓
LOG_SOFTMAX	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓
SQUARE	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓
SIN	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓
LINEAR	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓
ERF	asym-u8	asym-u8		✓	✓
	asym-i8	asym-i8		✓	✓
	fp32	fp32			✓
	fp16	fp16		✓	✓
GELU	asym-u8	asym-u8		✓	✓
	asym-i8	asym-i8		✓	✓
	fp32	fp32			✓
	fp16	fp16		✓	✓

8.3. Elementwise Operations

	Tensor Types		Execution Engine		
Operator	**Input**	**Output**	**NN**	**TP**	**PPU**
ADD	asym-u8	asym-u8	✓		
	asym-i8	asym-i8	✓		
	fp32	fp32			✓
	fp16	fp16			✓
SUBTRACT	asym-u8	asym-u8	✓		
	asym-i8	asym-i8	✓		
	fp32	fp32			✓
	fp16	fp16			✓
MULTIPLY	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓
DIVIDE	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓
MAXIMUM	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓
MINIMUM	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓

	Tensor Types		Execution Engine		
Operator	**Input**	**Output**	**NN**	**TP**	**PPU**
POW	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓
FLOORDIV	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓
MATRIXMUL	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓
RELATIONAL_OPS	asym-u8	bool8			✓
	asym-i8	bool8			✓
	fp32	bool8			✓
	fp16	bool8			✓
	bool8	bool8			✓
LOGICAL_OPS	bool8	bool8			✓
LOGICAL_NOT	bool8	bool8			✓
SELECT	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓
	bool8	bool8			✓
ADDN	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓

8.4. Normalization Operations

	Tensor Types		Execution Engine		
Operator	**Input**	**Output**	**NN**	**TP**	**PPU**
BATCH_NORM	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓
LRN2	asym-u8	asym-u8		✓	
	asym-i8	asym-i8		✓	
	fp32	fp32			✓
	fp16	fp16		✓	
L2_NORMALIZE	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓
LAYER_NORM	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓
INSTANCE_NORM	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓
BATCHNORM_SINGLE	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓

	Tensor Types		Execution Engine		
Operator	**Input**	**Output**	**NN**	**TP**	**PPU**
MOMENTS	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓
	fp16	fp16			✓
GROUP_NORM	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓

8.5. Reshape Operations

	Tensor Types		Execution Engine		
Operator	**Input**	**Output**	**NN**	**TP**	**PPU**
EXPAND_BROADCAST	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓
SLICE	asym-u8	asym-u8		✓	
	asym-i8	asym-i8		✓	
	fp32	fp32			✓
	fp16	fp16		✓	
SPLIT	asym-u8	asym-u8		✓	
	asym-i8	asym-i8		✓	
	fp32	fp32			✓
	fp16	fp16		✓	
CONCAT	asym-u8	asym-u8		✓	
	asym-i8	asym-i8		✓	
	fp32	fp32			✓
	fp16	fp16		✓	
STACK	asym-u8	asym-u8		✓	
	asym-i8	asym-i8		✓	
	fp32	fp32			✓
	fp16	fp16		✓	
UNSTACK	asym-u8	asym-u8		✓	
	asym-i8	asym-i8		✓	
	fp32	fp32			✓
	fp16	fp16		✓	

	Tensor Types		Execution Engine		
Operator	**Input**	**Output**	**NN**	**TP**	**PPU**
RESHAPE	asym-u8	asym-u8		✓	
	asym-i8	asym-i8		✓	
	fp32	fp32			✓
	fp16	fp16		✓	
SQUEEZE	asym-u8	asym-u8		✓	
	asym-i8	asym-i8		✓	
	fp32	fp32			✓
	fp16	fp16		✓	
PERMUTE	asym-u8	asym-u8		✓	
	asym-i8	asym-i8		✓	
	fp32	fp32			✓
	fp16	fp16		✓	
REORG	asym-u8	asym-u8		✓	
	asym-i8	asym-i8		✓	
	fp32	fp32			✓
	fp16	fp16		✓	
SPACE2DEPTH	asym-u8	asym-u8		✓	
	asym-i8	asym-i8		✓	
	fp32	fp32			✓
	fp16	fp16		✓	
DEPTH2SPACE	asym-u8	asym-u8		✓	
	asym-i8	asym-i8		✓	
	fp32	fp32			✓
	fp16	fp16		✓	
	bool8	bool8			

	Tensor Types		Execution Engine		
Operator	**Input**	**Output**	**NN**	**TP**	**PPU**
BATCH2SPACE	asym-u8	asym-u8		✓	
	asym-i8	asym-i8		✓	
	fp32	fp32			✓
	fp16	fp16		✓	
SPACE2BATCH	asym-u8	asym-u8		✓	
	asym-i8	asym-i8		✓	
	fp32	fp32			✓
	fp16	fp16		✓	
PAD	asym-u8	asym-u8		✓	
	asym-i8	asym-i8		✓	
	fp32	fp32			✓
	fp16	fp16		✓	
REVERSE	asym-u8	asym-u8		✓	
	asym-i8	asym-i8		✓	
	fp32	fp32			✓
	fp16	fp16		✓	
STRIDED_SLICE	asym-u8	asym-u8		✓	
	asym-i8	asym-i8		✓	
	fp32	fp32			✓
	fp16	fp16		✓	
REDUCE	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓

	Tensor Types		Execution Engine		
Operator	**Input**	**Output**	**NN**	**TP**	**PPU**
ARGMAX	asym-u8	asym-u8 / int16 / int32			✓
	asym-i8	asym-u8 / int16 / int32			✓
	fp32	int32			✓
	fp16	asym-u8 / int16 / int32			✓
ARGMIN	asym-u8	asym-u8 / int16 / int32			✓
	asym-i8	asym-u8 / int16 / int32			✓
	fp32	int32			✓
	fp16	asym-u8 / int16 / int32			✓
SHUFFLECHANNEL	asym-u8	asym-u8		✓	
	asym-i8	asym-i8		✓	
	fp32	fp32			✓
	fp16	fp16		✓	

8.6. RNN Operations

	Tensor Types			Execution Engine		
Operator	**Input**	**Kernel**	**Output**	**NN**	**TP**	**PPU**
LSTMUNIT_OVXLIB	asym-u8	asym-u8	asym-u8		✓	✓
	asym-i8	pc-sym-i8	asym-i8		✓	✓
	fp32	fp32	fp32			✓
	fp16	fp16	fp16		✓	✓
CONV2D_LSTM	asym-u8	asym-u8	asym-u8	✓		
	asym-i8	pc-sym-i8	asym-i8	✓		
	fp32	fp32	fp32			✓
	fp16	fp16	fp16			✓
CONV2D_LSTM_CELL	asym-u8	asym-u8	asym-u8	✓		
	asym-i8	pc-sym-i8	asym-i8	✓		
	fp32	fp32	fp32			✓
	fp16	fp16	fp16			✓
LSTM_OVXLIB	asym-u8	asym-u8	asym-u8		✓	✓
	asym-i8	pc-sym-i8	asym-i8		✓	✓
	fp32	fp32	fp32			✓
	fp16	fp16	fp16		✓	✓

	Tensor Types			Execution Engine		
Operator	**Input**	**Kernel**	**Output**	**NN**	**TP**	**PPU**
GRUCELL_OVXLIB	asym-u8	asym-u8	asym-u8		✓	✓
	asym-i8	pc-sym-i8	asym-i8		✓	✓
	fp32	fp32	fp32			✓
	fp16	fp16	fp16		✓	✓
GRU_OVXLIB	asym-u8	asym-u8	asym-u8		✓	✓
	asym-i8	pc-sym-i8	asym-i8		✓	✓
	fp32	fp32	fp32			✓
	fp16	fp16	fp16		✓	✓
SVDF	asym-u8	asym-u8	asym-u8		✓	✓
	asym-i8	pc-sym-i8	asym-i8		✓	✓
	fp32	fp32	fp32			✓
	fp16	fp16	fp16		✓	✓

8.7. Pooling Operations

	Tensor Types			Execution Engine	
Operator	**Input**	**Output**	**NN**	**TP**	**PPU**
POOL	asym-u8	asym-u8	✓	✓	
	asym-i8	asym-i8	✓	✓	
	fp32	fp32			✓
	fp16	fp16		✓	
ROI_POOL	asym-u8	asym-u8		✓	✓
	asym-i8	asym-i8		✓	✓
	fp32	fp32			✓
	fp16	fp16		✓	✓
POOLWITHARGMAX	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓
UPSAMPLE	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓

8.8. Miscellaneous Operations

	Tensor Types			Execution Engine	
Operator	**Input**	**Output**	**NN**	**TP**	**PPU**
PROPOSAL	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓
VARIABLE	asym-u8	asym-u8		✓	
	asym-i8	asym-i8		✓	
	fp32	fp32			✓
	fp16	fp16		✓	
DROPOUT	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓
RESIZE	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓
DATACONVERT	asym-u8	asym-u8		✓	
	asym-i8	asym-i8		✓	
	fp32	fp32			✓
	fp16	fp16		✓	
FLOOR	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓

	Tensor Types			Execution Engine	
Operator	**Input**	**Output**	**NN**	**TP**	**PPU**
EMBEDDING_LOOKUP	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓
GATHER	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓
GATHER_ND	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓
SCATTER_ND	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓
GATHER_ND_UPDATE	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓
TILE	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓

	Tensor Types			Execution Engine	
Operator	**Input**	**Output**	**NN**	**TP**	**PPU**
ELTWISEMAX	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓
SIGNAL_FRAME	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓
CONCATSHIFT	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓
UPSAMPLESCALE	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp16	fp16			✓
ROUND	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓
CEIL	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓

	Tensor Types			Execution Engine	
Operator	**Input**	**Output**	**NN**	**TP**	**PPU**
SEQUENCE_MASK	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓
REPEAT	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓
ONE_HOT	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓
CAST	asym-u8	asym-u8			✓
	asym-i8	asym-i8			✓
	fp32	fp32			✓
	fp16	fp16			✓

9. Direct Access In Android Applications

In Android, in addition to NN API, SyNAP can be directly accessed by applications. Direct access to SyNAP main benefits are zero-copy input/output and execution of optimized models compiled ahead of time with the SyNAP toolkit.

Access to SyNAP can be performed via custom JNI C++ code using the `synapnb` library. The library can be used as usual, the only constraint is to use the Synap allocator, which can be obtained with `synap_allocator()`.

Another option, is to use custom JNI C code using the `synap_device` library. In this case there are no constraints. The library allows to create new I/O buffers with the function `synap_allocate_io_buffer`. It is also possible to use existing DMABUF handles obtained for instance from gralloc with `synap_create_io_buffer`. The DMABUF can be accessed with standard Linux DMABUF APIs (i.e. mmap/munmap/ioctls).

SyNAP provides a sample JNI library that shows how to use the `synap_device` library in a Java application. The code is located in `java` and can be included in an existing Android application by adding the following lines to the `settings.gradle` of the application:

```
include ':synap'
project(':synap').projectDir = file("[absolute path to synap]/java")
```

The code can then be used as follows:

```
package com.synaptics.synap;

public class InferenceEngine {

    /**
     * Perform inference using the model passed in  data
     *
     * @param model EBG model
     * @param inputs arrays containing model input data, one byte array per network input,
     *                of the size expected by the network
     * @param outputs arrays where to store output of the network, one byte array per network
     *                 output, of the size expected by the network
     */
    public static void infer(byte[] model, byte[][] inputs, byte[][] outputs) {

        Synap synap = Synap.getInstance();

        // load the network
        Network network = synap.createNetwork(model);

        // create input buffers and attach them to the network
        IoBuffer[] inputBuffers = new IoBuffer[inputs.length];
        Attachment[] inputAttachments = new Attachment[inputs.length];

        for (int i = 0; i < inputs.length; i++) {
            // create the input buffer of the desired length
            inputBuffers[i] = synap.createIoBuffer(inputs[i].length);

            // attach the buffer to the network (make sure you keep a reference to the
            // attachment to avoid it is garbage collected and destroyed)
            inputAttachments[i] = network.attachIoBuffer(inputBuffers[i]);

            // set the buffer as the i-th input of the network
            inputAttachments[i].useAsInput(i);

            // copy the input data to the buffer
```

(continues on next page)

(continued from previous page)

```
            inputBuffers[i].copyFromBuffer(inputs[i], 0, 0, inputs[i].length);
        }

        // create the output buffers and attach them to the network
        IoBuffer[] outputBuffers = new IoBuffer[outputs.length];
        Attachment[] outputAttachments = new Attachment[inputs.length];

        for (int i = 0; i < outputs.length; i++) {
            // create the output buffer of the desired length
            outputBuffers[i] = synap.createIoBuffer(outputs[i].length);

            // attach the buffer to the network (make sure you keep a reference to the
            // attachment to avoid it is garbage collected and destroyed)
            outputAttachments[i] = network.attachIoBuffer(outputBuffers[i]);

            // set the buffer as the i-th output of the network
            outputAttachments[i].useAsOutput(i);
        }

        // run the network
        network.run();

        // copy the result data to the output buffers
        for (int i = 0; i < outputs.length; i++) {
            outputBuffers[i].copyToBuffer(outputs[i], 0, 0, outputs[i].length);
        }

        // release resources (it will be done automatically when the objects are garbage
        // collected but this may take some time so it is better to release them explicitly
        // as soon as possible)

        network.release();  // this will automatically release the attachments

        for (int i = 0 ; i < inputs.length; i++) {
            inputBuffers[i].release();
        }

        for (int i = 0 ; i < outputs.length; i++) {
            outputBuffers[i].release();
        }

    }

}
```

Note: To simplify application development by default VSSDK allows untrusted applications (such as application sideloaded or downloaded from Google Play store) to use the SyNAP API. Since the API uses limited hardware resources this can lead to situations in which a 3rd party application interferes with platform processes. To restrict access to SyNAP only to platform applications remove the file `vendor/vsi/sepolicy/synap_device/untrusted_app.te`.

Copyright

Trademarks

Synaptics; the Synaptics logo; add other trademarks here, are trademarks or registered trademarks of Synaptics Incorporated in the United States and/or other countries.

All other trademarks are the properties of their respective owners.

Notice

This document contains information that is proprietary to Synaptics Incorporated ("Synaptics"). The holder of this document shall treat all information contained herein as confidential, shall use the information only for its intended purpose, and shall not duplicate, disclose, or disseminate any of this information in any manner unless Synaptics has otherwise provided express, written permission.

Use of the materials may require a license of intellectual property from a third party or from Synaptics. This document conveys no express or implied licenses to any intellectual property rights belonging to Synaptics or any other party. Synaptics may, from time to time and at its sole option, update the information contained in this document without notice.

INFORMATION CONTAINED IN THIS DOCUMENT IS PROVIDED "AS-IS," AND SYNAPTICS HEREBY DISCLAIMS ALL EXPRESS OR IMPLIED WARRANTIES, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, AND ANY WARRANTIES OF NON-INFRINGEMENT OF ANY INTELLECTUAL PROPERTY RIGHTS. IN NO EVENT SHALL SYNAPTICS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES ARISING OUT OF OR IN CONNECTION WITH THE USE OF THE INFORMATION CONTAINED IN THIS DOCUMENT, HOWEVER CAUSED AND BASED ON ANY THEORY OF LIABILITY, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, AND EVEN IF SYNAPTICS WAS ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. IF A TRIBUNAL OF COMPETENT JURISDICTION DOES NOT PERMIT THE DISCLAIMER OF DIRECT DAMAGES OR ANY OTHER DAMAGES, SYNAPTICS' TOTAL CUMULATIVE LIABILITY TO ANY PARTY SHALL NOT EXCEED ONE HUNDRED U.S. DOLLARS.

Contact Us

Visit our website at www.synaptics.com to locate the Synaptics office nearest you.