synaptics

SyNAP

Synaptics Confidential. Disclosed Only Under NDA - Limited Distribution.

PN: 511-000xxx-01 Rev 2.5.0

Contents

1. Introduction . . . 3
1.1. Online Model Conversion . . . 3
1.2. Offline Model Conversion . . . 4
2. Getting Started . . . 5
2.1. `synap_cli_ic` Application . . . 5
2.2. `synap_cli_od` Application . . . 6
2.3. `synap_cli_ip` Application . . . 6
2.4. `synap_cli` Application . . . 7
2.5. `synap_init` Application . . . 7
2.6. Troubleshooting . . . 8
3. Using NNAPI . . . 9
3.1. Online Model Conversion . . . 9
3.2. Benchmarking Online Model . . . 9
3.3. NNAPI Compilation Caching . . . 10
3.4. Disabling NPU Usage From NNAPI . . . 11
4. Reference Models Timings . . . 12
5. Statistics And Usage . . . 15
5.1. `inference_count` . . . 15
5.2. `inference_time` . . . 15
5.3. `networks` . . . 15
5.4. `network_profile` . . . 16
5.5. Clearing Statistics . . . 16
5.6. Using `/sysfs` Information . . . 17
6. Working With Models . . . 18
6.1. Model Conversion . . . 18
6.2. Host Environment . . . 19
6.3. Using Docker . . . 19
6.4. Conversion Metafile . . . 19
6.5. Model Conversion Tutorial . . . 23
6.6. Model Profiling . . . 25
7. Framework API . . . 26
7.1. Basic Usage . . . 26
7.1.1. Network Class . . . 26
7.1.2. Using A Network . . . 27
7.2. Advanced Topics . . . 28
7.2.1. Tensors . . . 28
7.2.2. Buffers . . . 31
7.2.3. Allocators . . . 34
7.3. Advanced Examples . . . 35
7.3.1. Setting Buffers . . . 35
7.3.2. Settings Default Buffer Properties . . . 35
7.3.3. Buffer Sharing . . . 35
7.3.4. Recycling Buffers . . . 36
7.3.5. Using BufferCache . . . 36
7.3.6. Copying And Moving . . . 37
7.4. NPU Locking . . . 37
7.4.1. NPU Locking . . . 37
7.4.2. NNAPI Locking . . . 38
7.4.3. Description . . . 38
7.4.4. Sample Usage . . . 39
7.5. Preprocessing And Postprocessing . . . 43
7.5.1. InputData Class . . . 43
7.5.2. Preprocessor Class . . . 44

7.5.3. ImagePostprocessor Class 44
7.5.4. Classifier Class 45
7.5.5. Detector Class 47
7.6. Building Sample Code 48

8. Neural Network Processing Unit Operator Support 50
8.1. Basic Operations 51
8.2. Activation Operations 53
8.3. Elementwise Operations 57
8.4. Normalization Operations 59
8.5. Reshape Operations 61
8.6. RNN Operations 65
8.7. Pooling Operations 67
8.8. Miscellaneous Operations 68

9. Direct Access In Android Applications 72

List of Figures

Figure 1. Online model conversion and execution 3
Figure 2. Offline model conversion and execution 4
Figure 3. Network class 26
Figure 4. Running inference 27
Figure 5. Tensor class 29
Figure 6. Buffer class 32
Figure 7. Npu class 38
Figure 8. Locking the NPU 39
Figure 9. Locking and inference 40
Figure 10. Locking NNAPI 41
Figure 11. Automatic lock release 42
Figure 12. InputData class 44
Figure 13. ImagePostprocessor class 44
Figure 14. Classifier class 45
Figure 15. Detector class 47

List of Tables

Table 1. Model Classification 5
Table 2. Synaptics SuperResolution Models on Y+UV Channels 12
Table 3. Inference timings on VS680 13
Table 4. Inference timings on VS640 14
Table 5. Legend 50

1. Introduction

The purpose of SyNAP is to support the execution of neural networks by taking advantage of the available hardware accelerator. 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 is done inside the Network Processing Unit (*NPU*) hardware accelerator. 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 occour 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 using Android NNAPI, 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 NNAPI is not usable in a secure media path, that is to process data in secure streams.

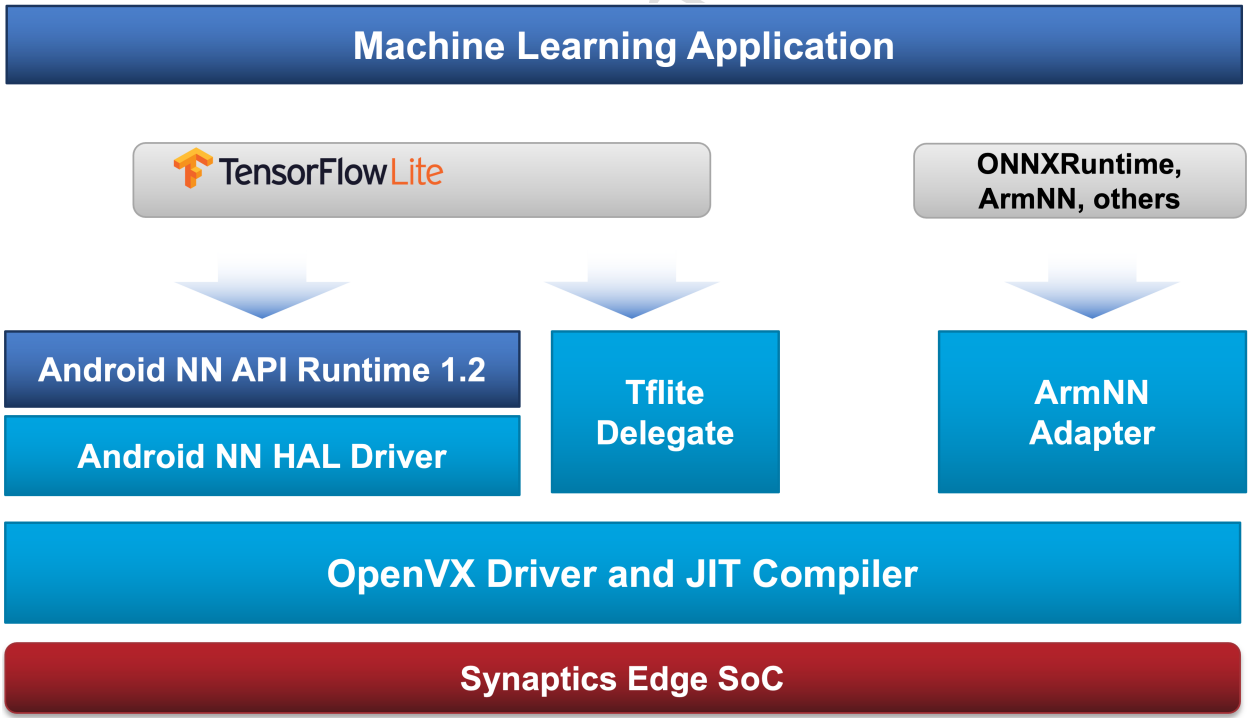

Figure 1. Online model conversion and execution

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 one-line command using Synap toolikt. Synap toolikt also supports more advanced operations, such as network quantization and simulation. 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

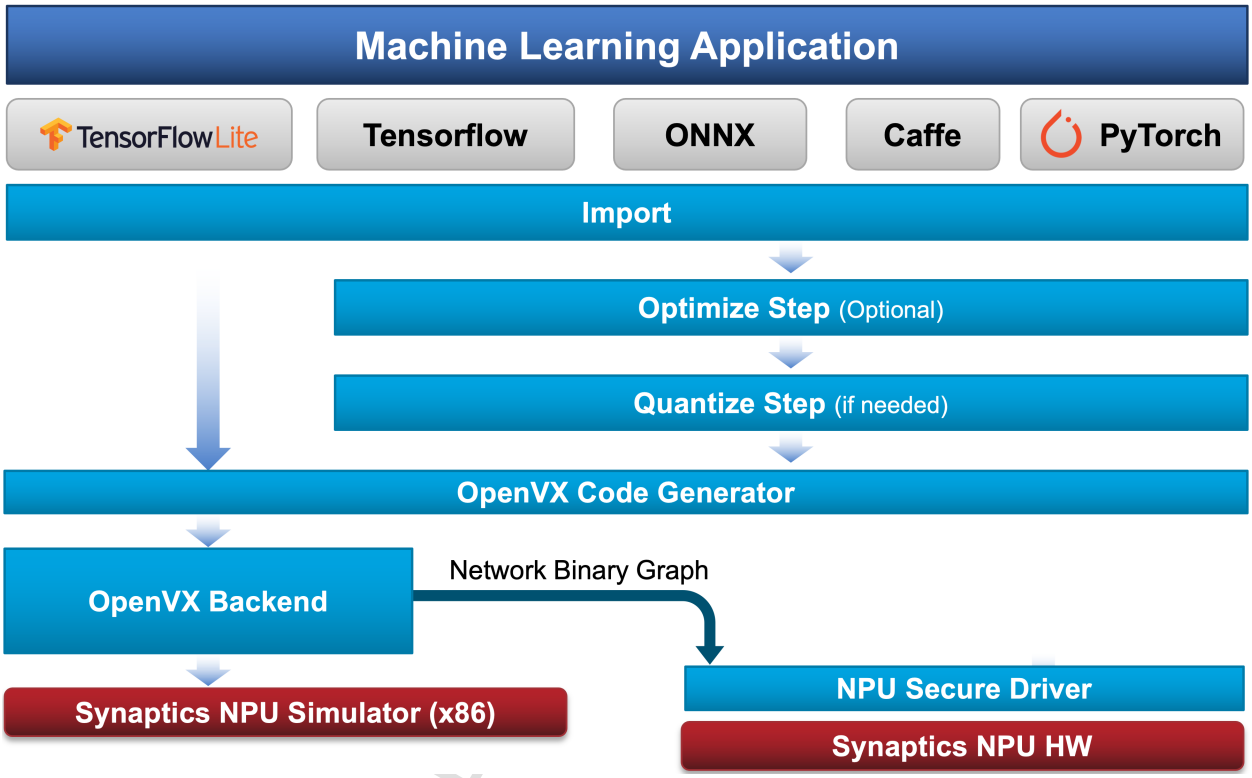

Figure 2. Offline model conversion and execution

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.

On Android the sample models can be found in the directory `/vendor/firmware/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 available in `/vendor/bin/`

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 network binary model (*.nb* extension)
- the converted synap network meta information (*.json* 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 HxW dimensions of the input image provided in input must correspond to the one expected by the network. For example mobilenet_v1 and mobilenet_v2 models expect an image of 224x224 pixels

Example:

```
$ cd /vendor/firmware/models/image_classification/imagenet/model/mobilenet_v2_1.0_224_quant
$ synap_cli_ic --nb model.nb --meta model.json ../../sample/goldfish_224x224.jpg
Loading network: model.nb
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 network binary model (*.nb* extension)
- the converted synap network meta information (*.json* 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

Example:

```
$ cd /vendor/firmware/models/object_detection/people/model/mobilenet224_full1/
$ synap_cli_od --nb model.nb --meta model.json ../../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
```

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 network binary model (*.nb* extension)
- the converted synap network meta information (*.json* extension)
- one or more images in *nv21* (that is *YUV420-semiplanar*) or *bin* (raw binary) format

It generates in output:

- a file containing the processed image in *nv21* format for each input file. The file is called with the same base name as the input file, but with extension *.out.nv21* By default it is created in the current directory, this can be changed with the `--out-dir` option.

Note: The HxW dimensions of the image provided in input must correspond to the one expected by the network. For super-resolution *jpeg* or *png* input files are not supported. These formats can be converted to *nv21* and rescaled to the required size using the `rgb_to_nv21.py` script. This script is located in the SyNAP *toolkit* directory and can be run directly on the host or inside the *synap* docker (for more info see *Using Docker*).

Note: The generated *.out.nv21* images can be converted to *png* format using the `nv21_to_rgb.py` script.

Example:

```
$ cd /vendor/firmware/models/image_processing/super_resolution/model/sr_qdeo_y_uv_1920x1080_
↪3840x2160
$ synap_cli_ip --nb model.nb --meta model.json ../../sample/ref_1920x1080.nv21
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.nv21
Inference time: 30.91 ms
Writing output to file: ref_1920x1080.out.nv21
```

2.4. synap_cli Application

This command line application can be used to run models of all categories. The purpose 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 /vendor/firmware/models/image_classification/imagenet/model/mobilenet_v2_1.0_224_quant
$ synap_cli --nb model.nb --meta model.json -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.

2.5. 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.nb
Inference failed
$ dmesg | grep NPU
[ 1211.651] synap:[do_synap_start_network():977] cannot execute model because the NPU is
↪reserved by another user
```

2.6. 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
  ...
5-17 03:06:39.334 212 212 I Synap : get_network_attrs():70: Parsing network metadata
5-17 03:06:39.335 212 212 I Synap : load_model():226: Network inputs: 1
5-17 03:06:39.335 212 212 I Synap : load_model():227: Network outputs: 1
5-17 03:06:39.340 212 212 I Synap : resume_cpu_access():65: Resuming cpu access on dmabuf:5
5-17 03:06:39.342 212 212 I Synap : set_buffer():189: Buffer set for tensor: input_0
5-17 03:06:39.342 212 212 I Synap : resume_cpu_access():65: Resuming cpu access on dmabuf:6
5-17 03:06:39.342 212 212 I Synap : set_buffer():189: Buffer set for tensor: output_66
5-17 03:06:39.342 212 212 I Synap : do_predict():78: Start inference
5-17 03:06:39.343 212 212 I Synap : suspend_cpu_access():54: Suspending cpu access on dmabuf:5
5-17 03:06:39.345 212 212 I Synap : do_predict():90: Inference time: 2.52 ms
5-17 03:06:39.345 212 212 I Synap : resume_cpu_access():65: Resuming cpu access on dmabuf:6
5-17 03:06:39.345 212 212 I Synap : unregister_buffer():139: Detaching buffer from input
↪tensor input_0
5-17 03:06:39.345 212 212 I Synap : set_buffer():158: Unset buffer for: input_0
5-17 03:06:39.345 212 212 I Synap : unregister_buffer():145: Detaching buffer from output
↪tensor output_66
5-17 03:06:39.345 212 212 I Synap : set_buffer():158: Unset buffer for: output_66
```

3. Using NNAPI

3.1. Online Model Conversion

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 takes 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 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 compilaton is caching, that is some the results of the computations perfomed to compile a model are cashed in a file so that they don't have to be executed again the next time the 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*.

3.2. Benchmarking Online Model

It is possible to benchmark the execution of a model with online conversion using the standard Android NNAPI interface and tools. The native tensorflow benchmark binary is already preinstalled on the board in `/vendor/bin/android_arm_benchmark_model`, so 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 baord, 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:

```
$ adb shell android_arm_benchmark_model --graph=/data/local/tmp/mobilenet_v1_0.25_224_
↪quant.tflite --use_nnapi=true --nnapi_accelerator_name=synap-npu

STARTING!
Log parameter values verbosely: [0]
Graph: [/data/local/tmp/mobilenet_v1_0.25_224_quant.tflite]
Use NNAPI: [1]
NNAPI accelerator name: [synap-npu]
NNAPI accelerators available: [synap-npu,nnapi-reference]
Loaded model /data/local/tmp/mobilenet_v1_0.25_224_quant.tflite
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/ 1 delegate kernels.
The input model file size (MB): 0.497264
```

(continues on next page)

(continued from previous page)

```
Initialized session in 28.37ms.
Running benchmark for at least 1 iterations and at least 0.5 seconds but terminate if
↪exceeding 150 seconds.
count=1 curr=1731574

Running benchmark for at least 50 iterations and at least 1 seconds but terminate if
↪exceeding 150 seconds.
count=679 first=1600 curr=1389 min=1370 max=1771 avg=1441 std=54

Inference timings in us: Init: 28370, First inference: 1731574, Warmup (avg): 1.
↪73157e+06, Inference (avg): 1441
Note: the benchmark tool itself affects memory footprint, the following is APPROXIMATE.
Peak memory footprint (MB): init=2.75391 overall=3.15625
```

For more information on android_arm_benchmark_model please refer to the official Android documentation page:

https://www.tensorflow.org/lite/performance/measurement

Note: `android_arm_benchmark_model` is meant to measure inference timing and memory usage, not to provide acutal inference results.

Note: if for any reason the model or part of it cannot be executed on the NPU, NNAPI will automatically fall back to CPU execution. In this case the inference time will be longer. Execution of the model (or part of it) on the NPU can be confirmed by looking at the SyNAP `inference_count` file in `sysfs` (see section Section 5.1).

It is possible to obtain detailed layer-by-layer inference timing by setting the profiling property before running `android_arm_benchmark_model`:

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

The profiling information will be availabe in `/sys/class/misc/synap/device/misc/synap/statistics/network_profile` while `android_arm_benchmark_model` is running.

For more information see section Section 5.4

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 eanbled 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 `android_arm_benchmark_model`:

```
$ adb shell android_arm_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 android_arm_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 android_arm_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
```

4. Reference Models 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. They are available in TFlite format from TensorFlow Hosted Model page:

https://www.tensorflow.org/lite/guide/hosted_models

The remaining models are Synaptics proprietary and include some test models, object detection models (*mobilenet224*) and super-resolution models.

Table 2. Synaptics SuperResolution Models on Y+UV Channels

Name	Input Image	Ouput Image	Factor	Notes
sr_fast_y_uv_1280x720_3840x2160	1280x720	3840x2160	3	
sr_fast_y_uv_1920x1080_3840x2160	1920x1080	3840x2160	2	
sr_qdeo_y_uv_1280x720_3840x2160	1280x720	3840x2160	3	
sr_qdeo_y_uv_1920x1080_3840x2160	1920x1080	3840x2160	2	

Remarks:

- All timing values in the following tables are expressed in milliseconds
- The columns *NNAPI CPU* and *NNAPI NPU* represent the inference time obtained by running the original *TFLite* model directly on the board (*online* conversion)
- Online CPU test has been done with 4 threads (`--num_threads=4`)
- The *Offline Infer* column represents the inference time obtained by using a model converted offline using SyNAP toolkit (median time over 10 consecutive inferences)
- Offline tests have been done with non-contiguous memory allocation and no cache flush

Table 3. Inference timings on VS680

Model	NNAPI CPU Infer	NNAPI NPU Init	NNAPI NPU Infer	Offline NPU Init	Offline NPU Infer
inception_v3_quant.tflite	266.94	14361	10.62	25.21	9.37
mobilenet_v2_quant.tflite	27.01	1431	2.65	3.89	1.98
mobilenet_v2_b4_quant.tflite	89.06	1445	14.08	4.56	12.25
unet_quant.tflite	463.14	906	10.38	4.17	13.09
mobilenet_v3_quant.tflite	86.23	1760	10.62	5.32	9.49
pynet_quant.tflite	1095.59	6616	19.76	8.90	17.81
vgg_quant.tflite	1449.34	2754	30.50	3.37	29.69
deeplab_v3_plus_quant.tflite	298.20	4815	62.48	3.56	58.17
dped_quant.tflite	335.63	1292	9.58	2.33	8.82
srgan_quant.tflite	1816.22	4280	59.78	6.40	54.03
mobilenet224_full80.tflite				643.81	26.63
mobilenet224_full1.tflite				509.61	14.17
posenet_mobilenet_075_quant.tflite	39.01	577	6.89	1.84	2.32
mobilenet_v1_0.25_224_quant.tflite	5.82	243	1.30	1.18	0.77
mobilenet_v2_1.0_224_quant.tflite	26.98	1424	2.46	3.89	1.77
inception_v4_299_quant.tflite	480.91	26174	18.93	65.87	18.05
sr_fast_y_uv_1920x1080_3840x2160.tflite				18.31	17.50
sr_fast_y_uv_1280x720_3840x2160.tflite				16.33	11.36
sr_qdeo_y_uv_1920x1080_3840x2160.tflite				21.51	25.74
sr_qdeo_y_uv_1280x720_3840x2160.tflite				19.36	20.46

Table 4. Inference timings on VS640

Model	NNAPI CPU Infer	NNAPI NPU Init	NNAPI NPU Infer	Offline NPU Init	Offline NPU Infer
inception_v3_quant.tflite	910.74	22451	31.00	32.97	29.82
mobilenet_v2_quant.tflite	83.96	2097	3.33	5.35	2.44
mobilenet_v2_b4_quant.tflite	327.35	2206	19.72	5.19	18.39
unet_quant.tflite	1143.65	1474	19.93	4.01	24.20
mobilenet_v3_quant.tflite	295.40	2770	13.62	6.31	11.91
pynet_quant.tflite	3492.44	10494	59.03	11.46	56.30
vgg_quant.tflite	5240.10	2116	103.66	4.28	102.65
deeplab_v3_plus_quant.tflite	1097.19	4748	84.37	3.88	70.85
dped_quant.tflite	1138.08	1086	26.65	2.61	25.72
srgan_quant.tflite	4248.82	4556	127.39	7.12	121.58
mobilenet224_full80.tflite				634.03	58.29
mobilenet224_full1.tflite				553.44	36.53
posenet_mobilenet_075_quant.tflite	127.26	887	10.80	2.48	4.13
mobilenet_v1_0.25_224_quant.tflite	16.80	399	1.82	1.81	0.93
mobilenet_v2_1.0_224_quant.tflite	83.64	2063	3.20	5.70	2.31
inception_v4_299_quant.tflite	1647.18	41666	54.07	104.33	53.82
sr_fast_y_uv_1920x1080_3840x2160.tflite				17.92	25.90
sr_fast_y_uv_1280x720_3840x2160.tflite				15.70	17.01
sr_qdeo_y_uv_1920x1080_3840x2160.tflite				20.67	33.56
sr_qdeo_y_uv_1280x720_3840x2160.tflite				17.70	26.16

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.

Synap information are available in the `/sys/class/misc/synap/device/misc/synap/statistics/`:

```
$ ls /sys/class/misc/synap/device/misc/synap/statistics/
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 /sys/class/misc/synap/device/misc/synap/statistics/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 /sys/class/misc/synap/device/misc/synap/statistics/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 /sys/class/misc/synap/device/misc/synap/statistics/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 *Online Model Conversion*) 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 /sys/class/misc/synap/device/misc/synap/statistics/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 /sys/class/misc/synap/device/misc/synap/statistics/inference_time
32233264
# echo > /sys/class/misc/synap/device/misc/synap/statistics/inference_time
# cat /sys/class/misc/synap/device/misc/synap/statistics/inference_time
0
```

(continues on next page)

(continued from previous page)

```
# cat /sys/class/misc/synap/device/misc/synap/statistics/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 > /sys/class/misc/synap/device/misc/synap/statistics/inference_time;
usleep $us;
npu_usage=$((`cat /sys/class/misc/synap/device/misc/synap/statistics/inference_time`*100/us));
echo "Average NPU usage: $npu_usage%"
```

6. Working With Models

6.1. Model Conversion

In order to perform inference using SyNAP, the network model has to be converted from the original representation to an internal representation optimized for the target hardware. This can be done using a command-line utility: *synap_converter.py*

This tool 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.nb** the converted network model, including weights
- **model.json** meta information about the generated model, needed at inference time
- **model.info.txt** additional information about the generated model for user reference, including:
  - layer table
  - operation table
  - memory usage

The following network model formats are supported:

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

Example:

```
$ ./synap_convert.py --model mobilenet_v1_quant.tflite --target VS680 --out-dir mnv1
$ ls mnv1
model.info.txt  model.json  model.nb
```

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.py --model mnist.prototxt --weights mnist.caffemodel --target VS680 --out-
↪dir mnist
```

Note: 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.

6.2. Host Environment

The conversion tool works natively on linux **Ubuntu 18.04** with python3 installed. The following packages are required:

numpy==1.19.2 tensorflow-cpu==2.5.0 scipy==1.4.1 networkx==1.11 image==1.5.5 lmdb==0.93 onnx==1.6.0 h5py==3.1.0 dill==0.2.8.2 ruamel.yaml==0.16.12 onnx_tf==1.2.1 ply==3.11 flatbuffers==1.12 Pillow==5.3.0 future==0.18.2 torch==1.8.1+cpu torchvision==0.9.1+cpu torchaudio==0.8.1

6.3. Using Docker

The conversion tool can also be run on Windows or OSX hosts inside a *Docker* container. Please see https://docs.docker.com/get-docker/ for how to install Docker on your host. No additional dependencies have to be installed.

Once docker is installed, the `synap_run.sh` shell script can be used to automatically build and run the *synap* docker container. In this case the desired synap tool command line is just passed as a parameter:

```
$ synap_run.sh synap_convert.py --model mobilenet_v1_quant.tflite --target VS680 \
  --out-dir mnv1
```

Note: The model file and the output directory specified must be inside or below the current working directory since this is the directory mounted by default inside the docker container. If this is not the case you can use the `synap_run.sh` with `--mrd` option to specify the root directory to mount in the docker.

6.4. 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

Example:

```
$ ./synap_convert.py --model mobilenet_v1_quant.tflite --meta mobilenet.json --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.

```
data_layout: nhwc

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

inputs:
  - name: input
```

(continues on next page)

(continued from previous page)

```
    shape: [1, 224, 224, 3]
    means: [128, 128, 128]
    scale: 128
    format: rgb
    security: any

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

quantization:
    data_type: uint8
    scheme: asymmetric_affine
    dataset:
      - ../../sample/*_224x224.jpg
```

- `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.

  - `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.

- **`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 selected data format. 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 values of input images before quantization. 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.

  - `scale`

    Used to normalize the range of values of input images before quantization. The scale value, a single value for all the channels in the corresponding input. The *i-th* channel of each input is normalized as: `norm = (in - means[i]) / scale`

  - `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 `Preprocessor` recognises the following values:

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

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

  - `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 supprted.

- `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. Importing an already-quantized model allows to do the quantization in the Neural Network framework directly (eg Tensorflow Lite) which can also provide additional features such as quantization-aware training. Quantizing the model at conversion time can be the useful when importing an existing floating point model, or when requiring some specific features not generally supported in NN frameworks (eg. 16 bits quantization).

  - `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`

    `qbfloat16`

  - `scheme`

    Select the quantization algorithm. Available schemes are:

    `asymmetric_affine` (default)

    `dynamic_fixed_point`

    `perchannel_symmetric_affine`

    `symmetric_affine`

    `qbfloat16`

    Scheme `symmetric_affine` is not supported with data type `int8`.

  - `dataset` [q]

    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.

A sample 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.

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 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.

6.5. 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 <mount-root-dir> is above both VSSDK and the current directory):

```
$ toolkit/synap_run.sh --mrd <mount-root-dir> "synap_convert.py --model ssd_mobilenet_v1_
↪1_default_1.tflite --meta ssd_mobilenet.yaml --target VS680 --out-dir ssd_mobilenet"
```

4. Push the model to the board:

```
$ adb root
$ adb remount
$ adb shell mkdir /vendor/firmware/models/object_detection/coco/model/ssd_mobilenet/
$ adb push ssd_mobilenet/* /vendor/firmware/models/object_detection/coco/model/ssd_
↪mobilenet/
```

5. Execute the model:

```
$ adb shell
# cd /vendor/firmware/models/object_detection/coco/model/ssd_mobilenet
# synap_cli_od --nb model.nb --meta model.json ../../sample/sample001_640x480.jpg"

Input image: ../../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
```

6.6. 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:

```
$ toolkit/synap_run.sh synap_convert.py --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 /vendor/firmware/models/image_classification/imagenet/model/
$ adb shell
# cd /vendor/firmware/models/image_classification/imagenet/model/mobilenet_profiling
# synap_cli --nb model.nb --meta model.json --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
|   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
```

7. Framework API

The core functionality of the Synap framework is to execute a precompiled neural network on the NPU accelerator. This is done via the `Network` class. The Network class has be designed to be simple to use in the most common cases while still being flexible enough for most advanced use-cases.

Please refer to Android NNAPI documentation for how to execute a model with online conversion.

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 three things that can be done with a network:

- load a model, by providing a *Network Binary Graph* and the corresponding *Meta* information
- 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.

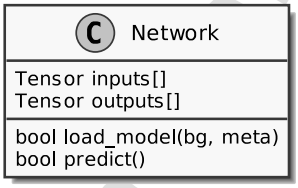

Figure 3. Network class

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

Public Functions

bool **load_model**(**const** std::string &*ebg_file*, **const** std::string &*ebg_meta_file*)
Load model.

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

Parameters

- **ebg_file** – path to a network executable binary graph file
- **ebg_meta_file** – path to the network's metadata

Returns true if success

bool **load_model**(**const** void **ebg_data*, size_t *ebg_data_size*, **const** char **ebg_meta_data*)
Load model.

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

Parameters

- **ebg_data** – network executable binary graph data, as from e.g. fread()
- **ebg_data_size** – size in bytes of ebg_data

- **ebg_meta_data** – network's metadata (JSON-formatted text)

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 weights and meta-information. Both these files are 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)

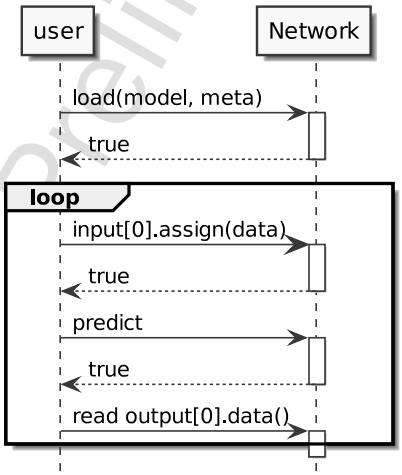

Figure 4. Running inference

Example:

```
Network net;
net.load_model("model.nb", "model.json");
vector<uint8_t> in_data = custom_read_input_data();
net.inputs[0].assign(in_data);
net.predict();
custom_process_result(net.outputs[0]);
```

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 NPU HW accelerator. In most cases the cost of this extra copy is negligible, when this is not the case the copy can 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();
  ```

- 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

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.

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

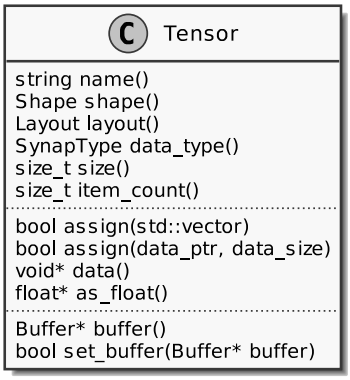

Figure 5. Tensor class

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 **assign**(**const** std::vector<uint8_t> &*data*)
Copy the content of a vector to the tensor data buffer.

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

Parameters **data** – data to be copied in tensor. Data size must match tensor size.

Returns true if success

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

The data is considered as raw data so no type 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 type 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

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

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 match the tensor size or the operation will be rejected. 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.

Returns true if success

struct Private

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

enum 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.

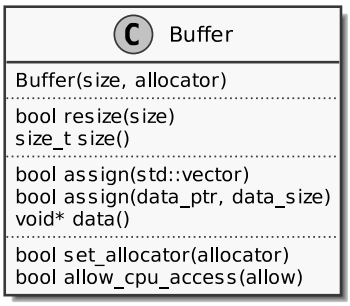

Figure 6. Buffer class

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 dmabuf (or part of it) registered with the TZ kernel.
- **offset** – offset of the actual data insithe the memory area
- **size** – size of the actual data

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** std::vector<uint8_t> &*data*)
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** – data to be copied

Returns true if all right

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. 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.nb", "model.json");

// 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();
```

7.3.2. 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.nb", "model.json");

// 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();
```

7.3.3. 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.nb", "meta1.json");
Network net2;
net2.load_model("nbg2.nb", "meta2.json");

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

7.3.4. 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.nb", "model.json");

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

7.3.5. 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.

7.3.6. 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.nb", "model.json");
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.nb", "model.json");
    ...
}
```

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*

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.

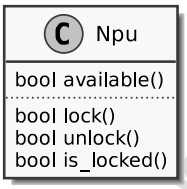

Figure 7. Npu class

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.

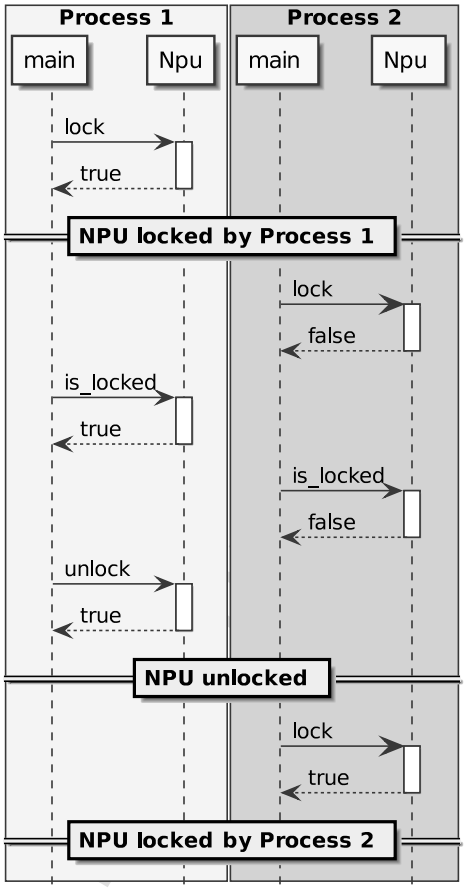

Figure 8. Locking the NPU

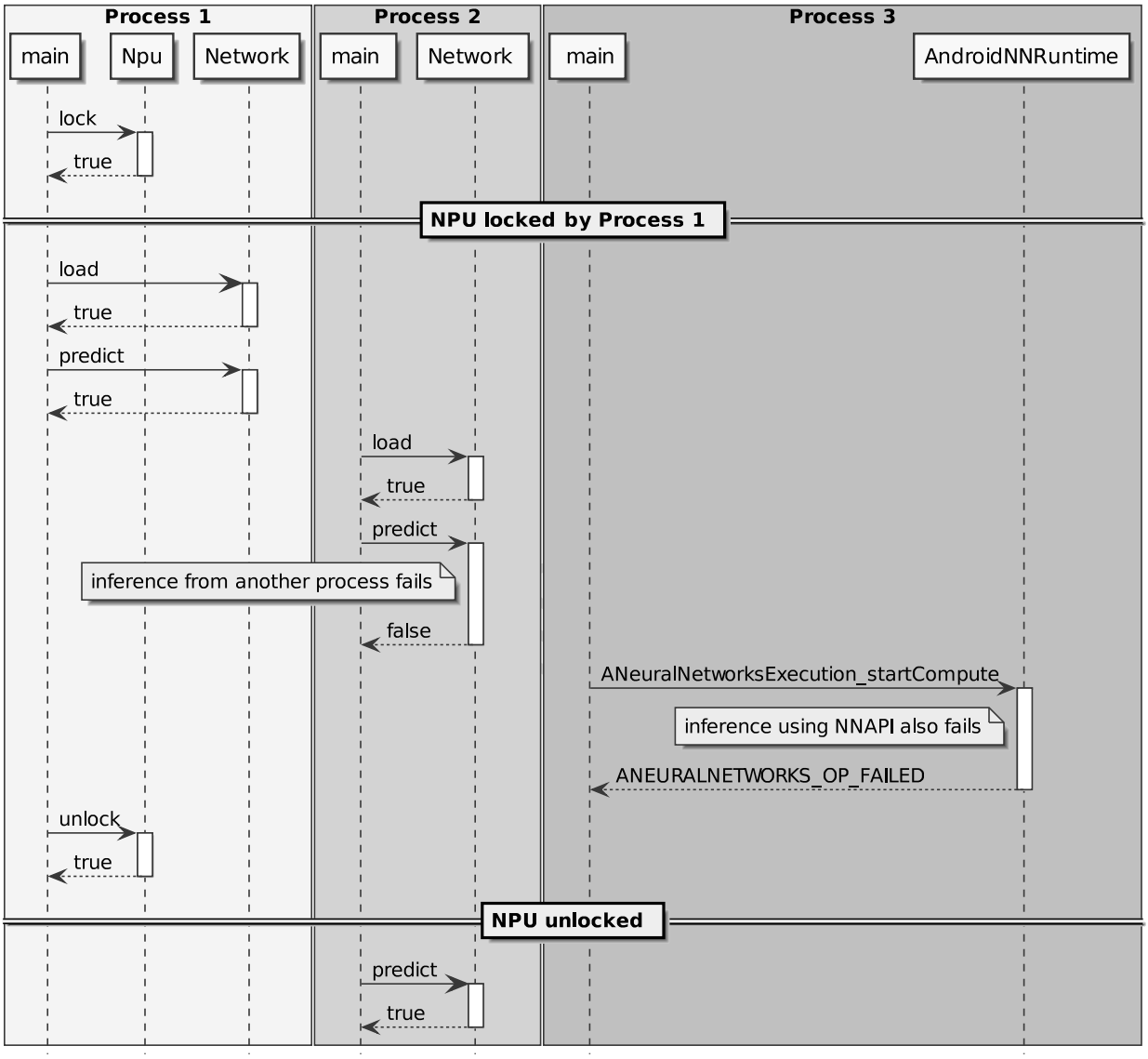

Figure 9. Locking and inference

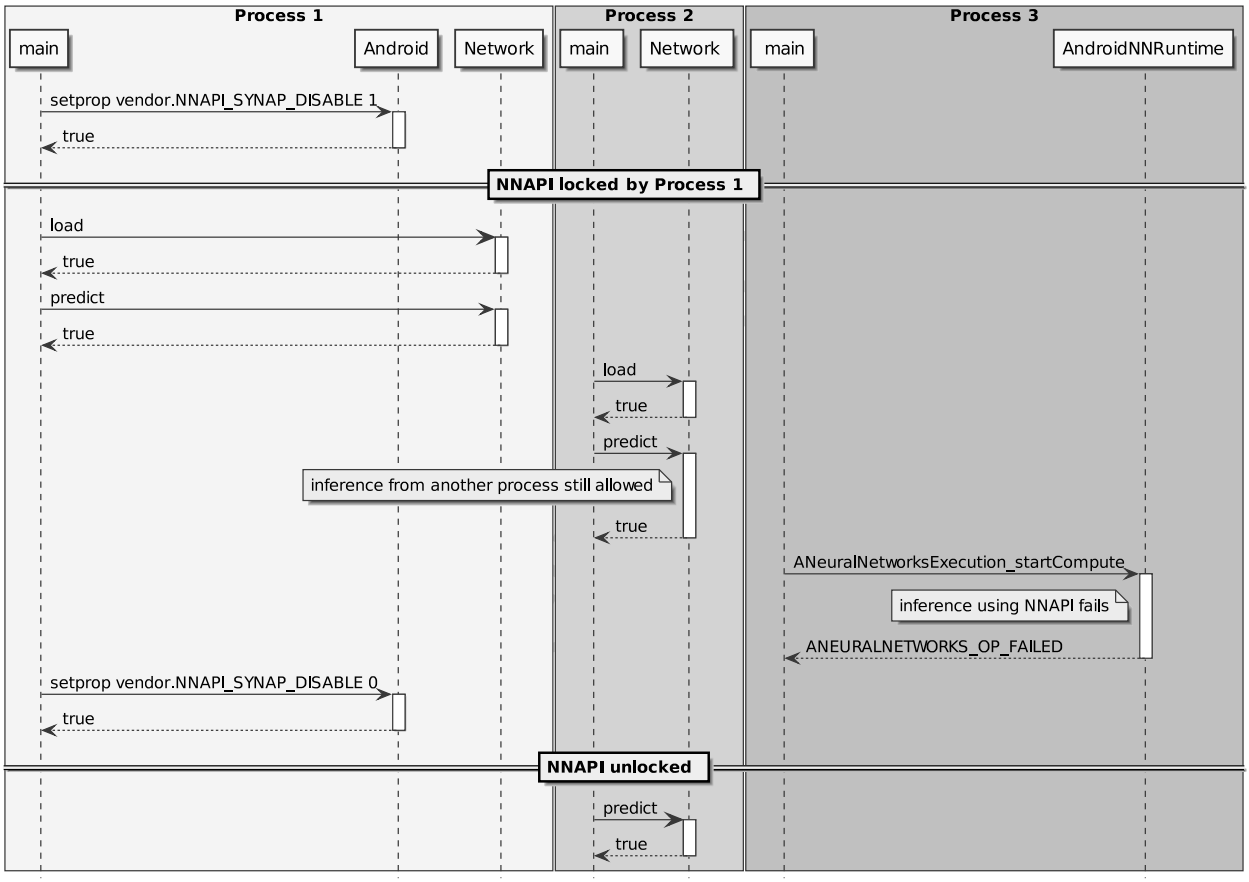

Figure 10. Locking NNAPI

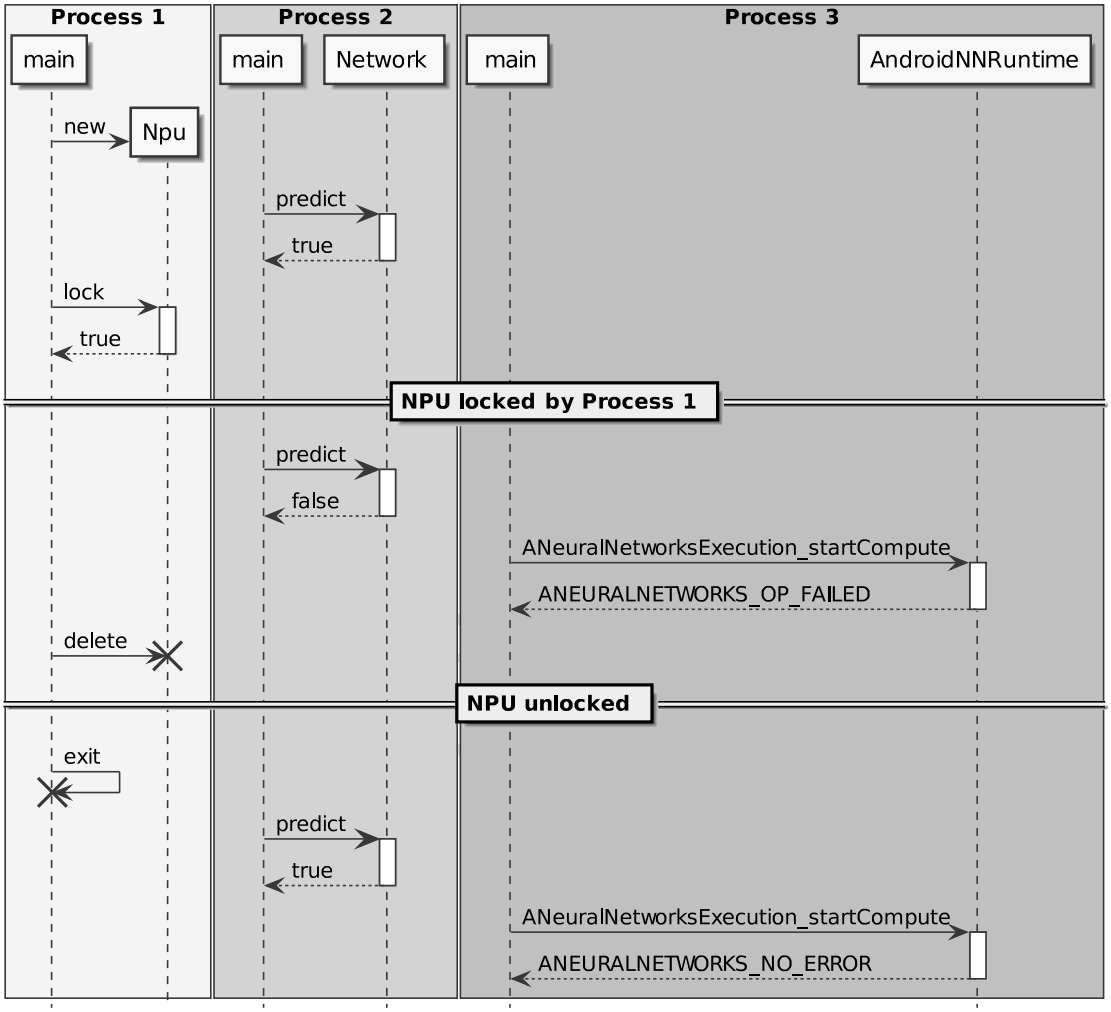

Figure 11. Automatic lock release

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

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

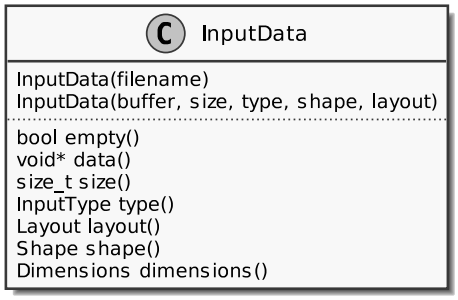

Figure 12. InputData class

7.5.2. Preprocessor Class

This class takes in input an `InputData` object and assigns its content to the input Tensor 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 scaling to fit the tensor dimensions

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

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.

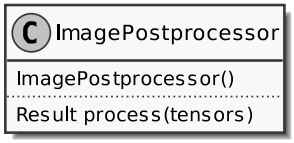

Figure 13. ImagePostprocessor class

Example:

```
Preprocessor preprocessor
Network net;
ImagePostprocessor postprocessor;

net.load_model("model.nb", "model.json");
InputData image("sample_image.jpg");
```

(continues on next page)

(continued from previous page)

```
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

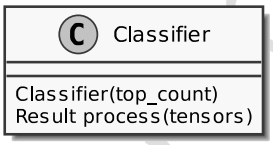

Figure 14. Classifier class

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

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

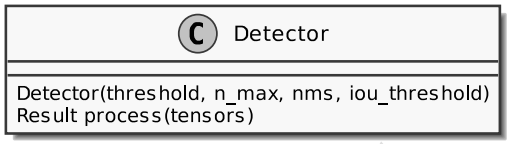

Figure 15. Detector class

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

Public Functions

Detector(float *score_threshold* = 0.5, int *n_max* = 0, bool *nms* = true, float *iou_threshold* = .5)
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)

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*, Dim2d *input_dim*)
Perform detection on network output tensors.

Parameters

- **tensors** – output tensors of the network Currently only models in the standard TFLite_Detection_PostProcess format are supported.
- **input_dim** – actual x,y dimensions of the input image

Returns detection results

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)

struct Scaling

Example:

```
Preprocessor preprocessor
Network net;
Detector detector;
net.load_model("model.nb", "model.json");
InputData image("sample_image.jpg");
preprocessor.assign(net.inputs[0], image);
net.predict();
Detector::Result objects = detector.process(net.outputs);
```

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. Build of the sample 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 VS640/VS680 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 5. 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 particular 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.