Qualcomm Technologies, Inc.

AI Developer Workflow

80-70018-15B AD

May 5, 2025

Contents

1 AI developer workflow documentation

The Qualcomm AI Runtime SDK facilitates high-performance machine learning inference on Qualcomm hardware. It supports frameworks such as TensorFlow, PyTorch, ONNX, and LiteRT. It's designed to run already-trained networks quickly and efficiently on Qualcomm hardware.

1.1 Overview

Featured topics

- Overview of the AI-ML developer workflow

1.2 Compile and optimize an AI model

Covers model conversion, quantization, and accuracy fine tuning.

- AI Hub
- LiteRT
- Qualcomm AI Runtime SDK

1.3 Run AI/ML sample applications

Run the default sample app to run AI/ML models or customize the sample app using either a Qualcomm evaluation kit or Qualcomm Device Cloud (QDC).

- Run the default reference app
- Customize the reference app and run the modified version

1.4 Develop your own application

Covers how to implement your own application on a Qualcomm platform. This is an advanced topic and is for experienced developers.

- Develop using the Qualcomm Visual Studio Code Extension
- Develop using the Qualcomm IM SDK

1.5 Use AI Hub models and labels with the GStreamer API

Explains how to use AI Hub models and labels with GStreamer commands on Qualcomm evaluation kits.

- Image classification
- Object detection
- Semantic segmentation
- Super resolution

1.6 Troubleshooting and FAQ

Get answers to common questions and learn how to get additional help.

1.7 Model porting best practices

Export YoloV8 model using Neural Processing Engine SDK Run the demo

2 Overview

Join us and watch this training session focused on the Qualcomm Linux AI Developer Workflow.

It is tailored for developers and professionals in the IoT sector who are keen to enhance their understanding and skills in deploying on-device AI solutions on a Qualcomm platform.

The AI/ML developer workflow on Qualcomm Linux has two major steps:

Step 1 Compile and optimize a model	• Compile and optimize the model from the third-party AI framework to efficiently run on Qualcomm hardware. For example, a Tensorflow model can be exported to a LiteRT model. • Optionally, quantize, fine-tune performance, and accuracy using hardware-specific customizations.
Step 2 Build an application to use the optimized model to run on device inference	• Integrate the AI model into the use case pipeline. • Cross-compile the application to generate an executable binary using dependent libraries.

Important:

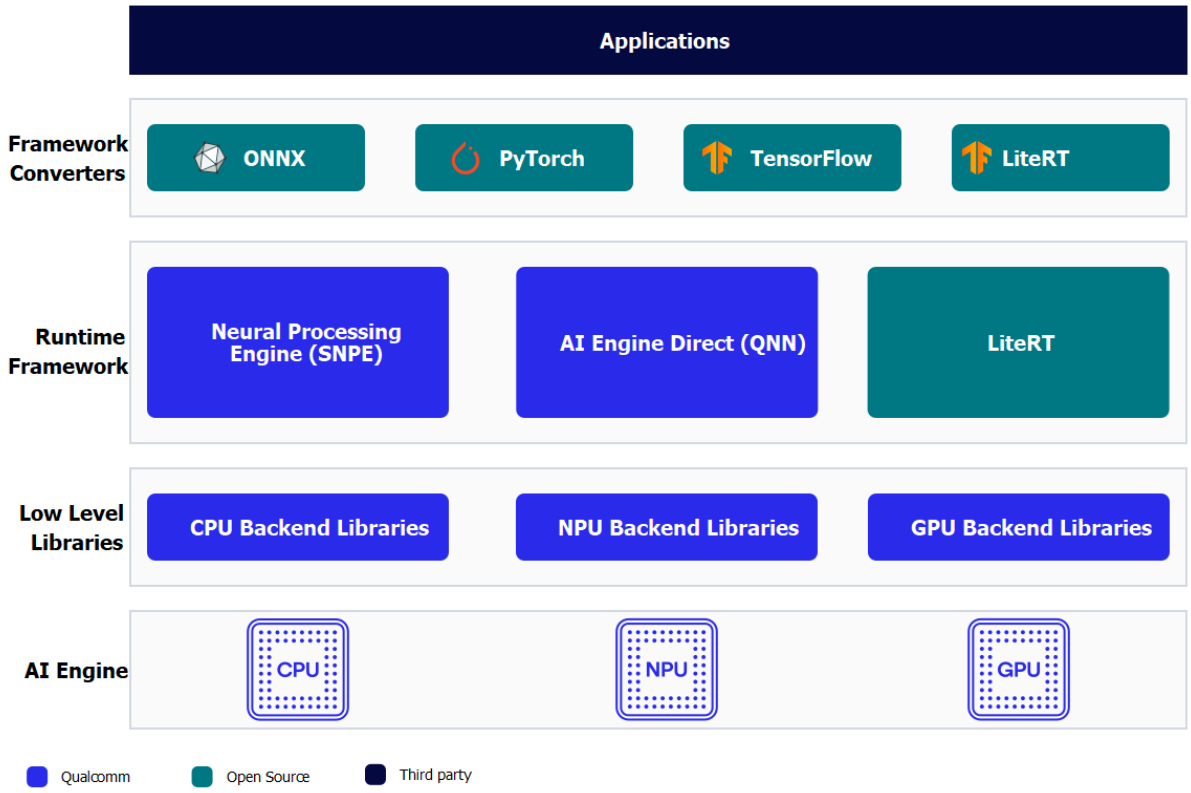
- Ensure that the host computer uses Ubuntu 22.04.
- The commands in this document are compatible with Qualcomm Linux 1.4.

  Verify your Qualcomm Linux release version by running the commands described in the Dev Kit Quick Start guide

  If your release version isn't 1.4, update your software.
- Sample applications and AI procedures in this document are compatible with the supported versions.

  Ensure you download the matching SDKs to your host computer before starting AI/ML development.

2.1 AI architecture



Developers can bring models from ONNX, PyTorch, TensorFlow or TFLite and run them efficiently on Qualcomm AI Hardware–HTP (NPU), GPU, CPU using Qualcomm AI SDKs.

2.2 AI hardware

- **Qualcomm Kryo™ CPU**: Best-in-class CPU with high performance and remarkable power efficiency.
- **Qualcomm Adreno GPU**: Suitable to run AI workloads with balanced power and performance. AI workloads are accelerated with OpenCL kernels. The GPU can also be used to accelerate model pre/postprocessing.
- **Qualcomm Hexagon Tensor Processor (HTP)**: Also known as NPU/DSP/HMX, suitable to run AI workloads with low-power and high-performance. For optimized performance, pretrained models need be quantized to one of the supported precisions.

2.3 AI software

AI stack contains SDKs to harness the power of AI hardware accelerators. Developers can use one of the SDKs of their choice to deploy AI workloads. Pretrained models (except for LiteRT models) need to be converted to an executable format with the selected SDK before running them. LiteRT models can be run directly using TFLite Delegate.

TFLite

LiteRT models can be natively run on Qualcomm hardware with acceleration using the following Delegates.

Delegate	Acceleration
AI Engine Direct Delegate (QNN Delegate)	CPU, GPU and HTP
XNNPACK Delegate	CPU
GPU Delegate	GPU

Qualcomm Neural Processing Engine SDK (SNPE)

Qualcomm Neural Processing Engine (SNPE) is a software accelerated runtime for execution of deep neural networks. SNPE offers tools to convert, quantize neural networks and accelerate them on hardware accelerators including CPU, GPU, and HTP.

Qualcomm AI Engine Direct (QNN)

Qualcomm AI Engine Direct is a software architecture for AI/ML use cases on Qualcomm chipsets and AI acceleration cores. The architecture is designed to provide a unified API and modular and extensible per-accelerator libraries, which form a reusable basis for full stack AI solutions. It provides support for runtimes such as Qualcomm Neural Processing SDK, LiteRT AI Engine Direct Delegate.

AI Model Efficiency Toolkit (AIMET)

Open-source library to optimize (compress and quantize) trained neural network models. This is a complex SDK designed to generate optimized quantized models and is intended only for advanced developers.

3 Compile and optimize an AI model

Users can take one of two paths available to compile and optimize their models.

AI Hub

Use AI Hub to optimize an AI model

Bring your own model or try preoptimized models on Snapdragon in < 5 min.

Note: To bring your own model (BYOM), see Integrate a custom AI model in an application for more information.

AI Software Stack

Use LiteRT to optimize an AI model

Port your LiteRT AI models to Snapdragon

Use Qualcomm AI Runtime SDK to optimize an AI model

Port your models using the all-in- one, easily customizable Qualcomm AI Runtime (QAIRT) SDK

3.1 Set up AI Hub to optimize an AI model

For quick prototyping of models on Qualcomm AI hardware, AI Hub provides a way to optimize, validate, and deploy machine learning models on-device for vision, audio, and speech use cases

Set up your environment

1. Setup your Python environment. Install miniconda on your machine.

   Windows

   When the installation finishes, open an Anaconda prompt from the Start menu.

macOS/Linux

When the installation finishes, open a new shell window.

Set up a Python virtual environment for AI Hub.

```
conda activate
```

```
conda create python=3.10 -n qai_hub
```

```
conda activate qai_hub
```

2. Install git.

```
sudo apt-get install git
```

3. Install the AI Hub Python client.

```
pip3 install qai-hub
```

```
pip3 install "qai-hub[torch]"
```

4. Sign in to AI Hub.

   Go to AI Hub and sign in with your Qualcomm ID to view information about jobs you create.

   Once signed in, go to *Account > Settings > API Token*. This should provide an API token that you can use to configure your client.

5. Configure the client with your API token using the following command in your terminal.

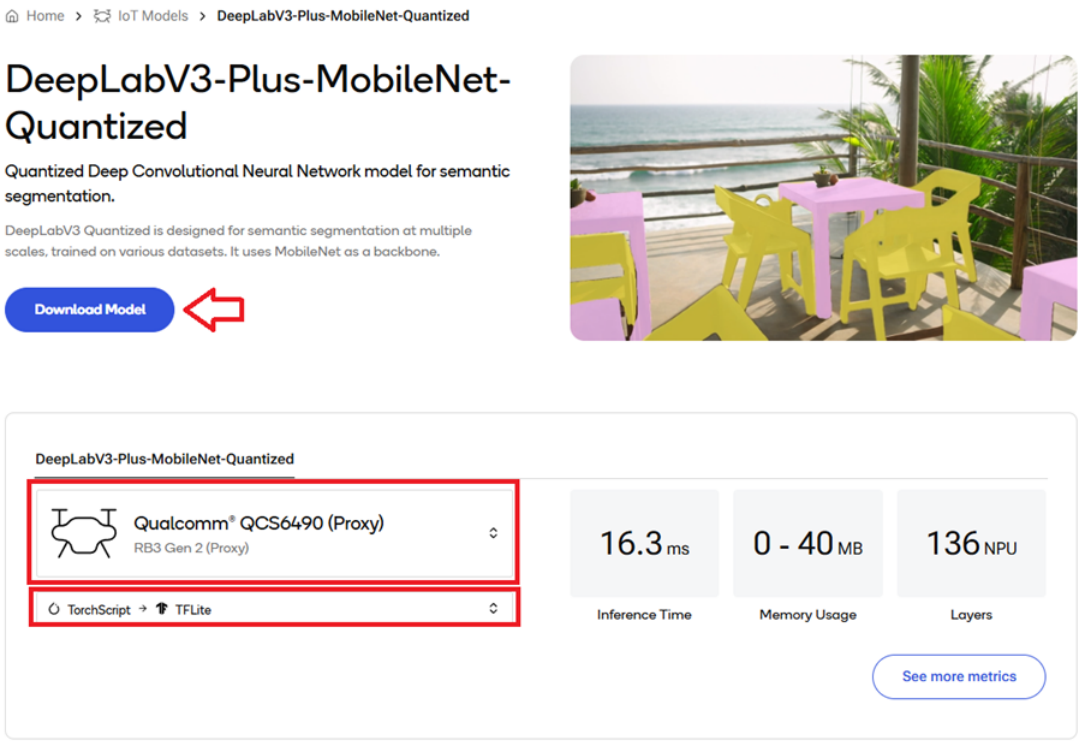
```
qai-hub configure --api_token <INSERT_API_TOKEN>
```

Choose an AI Hub workflow

Try a pre-optimized model

1. Go to AI Hub Model Zoo to access pre-optimized models available for Qualcomm evaluation kits.
2. Filter models available for your EVK. For example, pre-optimized models for RB3Gen2 can be downloaded by selecting *Qualcomm QCS6490* as the chipset in the left pane.
3. Select a model from the filtered view to go to the model page.
4. On the model page, select the chipset from the dropdown list and choose the *TorchScript > LiteRT* path.

5. Select download to begin model download. The downloaded model is preoptimized and ready for deployment. See Develop your own AI/ML application for more information about deploying the model.



Bring your own model

1. Select a pretrained model in PyTorch or ONNX format.
2. Submit a model for compilation or optimization to AI Hub using python APIs.

   When submitting a compilation job, you must select a device or the chipset for your EVK and the target runtime to compile the model. For RB3Gen2, the LiteRT runtime is supported.

Chipset	Runtime	CPU	GPU	HTP
QCS6490	LiteRT	INT8,FP16, FP32	FP16,FP32	INT8,INT16

   On submission, AI Hub generates a unique ID for the job. You can use this job ID to view job details.

3. AI Hub optimizes the model based your device and runtime selections.
   - Optionally, you can submit a job to profile or inference the optimized model (using Python APIs) on a real device provisioned from a device farm.
     – Profiling: Benchmarks the model on a provisioned device and provides statistics,

including average inference times at the layer level, runtime configuration, etc.

  - Inference: Performs inference using an optimized model on data submitted as part of the inference job by running the model on a provisioned device.

4. Each submitted job will be available for review in the AI Hub portal. A submitted compilation job will provide a downloadable link to the optimized model. This optimized model can then be deployed on a local development device like RB3Gen2.

The following is an example of the described workflow taken from the AI Hub documentation. In this example, a MobileNet V2 pretrained model from PyTorch is uploaded to AI Hub and compiled to an optimized LiteRT model to run on an RB3Gen2 target.

```
import qai_hub as hub
import torch
from torchvision.models import mobilenet_v2
import numpy as np

# Using pre-trained MobileNet
torch_model = mobilenet_v2(pretrained=True)
torch_model.eval()

# Trace model (for on-device deployment)
input_shape = (1, 3, 224, 224)
example_input = torch.rand(input_shape)
traced_torch_model = torch.jit.trace(torch_model, example_input)

# Compile and optimize the model for a specific device
compile_job = hub.submit_compile_job(
    model=traced_torch_model,
    device=hub.Device("QCS6490 (Proxy)"),
    input_specs=dict(image=input_shape),
    #compile_options="--target_runtime tflite",
)

# Profiling Job
profile_job = hub.submit_profile_job(
    model=compile_job.get_target_model(),
    device=hub.Device("QCS6490 (Proxy)"),
)

sample = np.random.random((1, 3, 224, 224)).astype(np.float32)

# Inference Job
inference_job = hub.submit_inference_job(
    model=compile_job.get_target_model(),
```

```
    device=hub.Device("QCS6490 (Proxy)"),
    inputs=dict(image=[sample]),
)

# Download model
compile_job.download_target_model(filename="/tmp/mobilenetv2.tflite")
```

Note: To deactivate a previously activated `qai_hub` environment use the following command.

```
conda deactivate
```

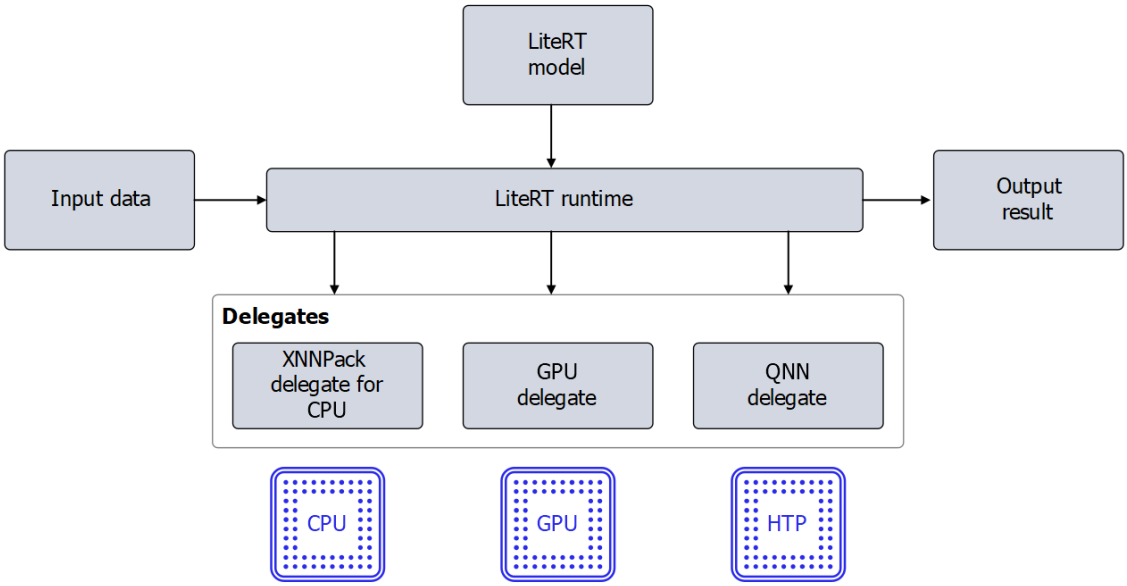
Once the model is downloaded, it's ready to be used for you to Develop your own AI/ML application.

For more details about the AI Hub workflow and APIs, see the AI Hub documentation, explore the AI Hub tutorial videos, or watch the following video about how to profile models in AI Hub.

Note: The video above uses Python 3.8 as an example.

Python 3.8 and Python 3.10 are supported.

3.2 Set up LiteRT to optimize an AI model



LiteRT is an open-source deep learning framework for on-device inference. LiteRT helps developers run their models on mobile, embedded, and edge platforms by optimizing the model for latency, model size, power consumption, etc. Qualcomm supports executing LiteRT models natively on Qualcomm Linux hardware through LiteRT Delegates as listed below.

Delegate	Acceleration
AI Engine Direct Delegate (QNN Delegate)	CPU, GPU and HTP
XNNPack Delegate	CPU
GPU Delegate	GPU

Reference guide

→ Reference guide

Run sample app

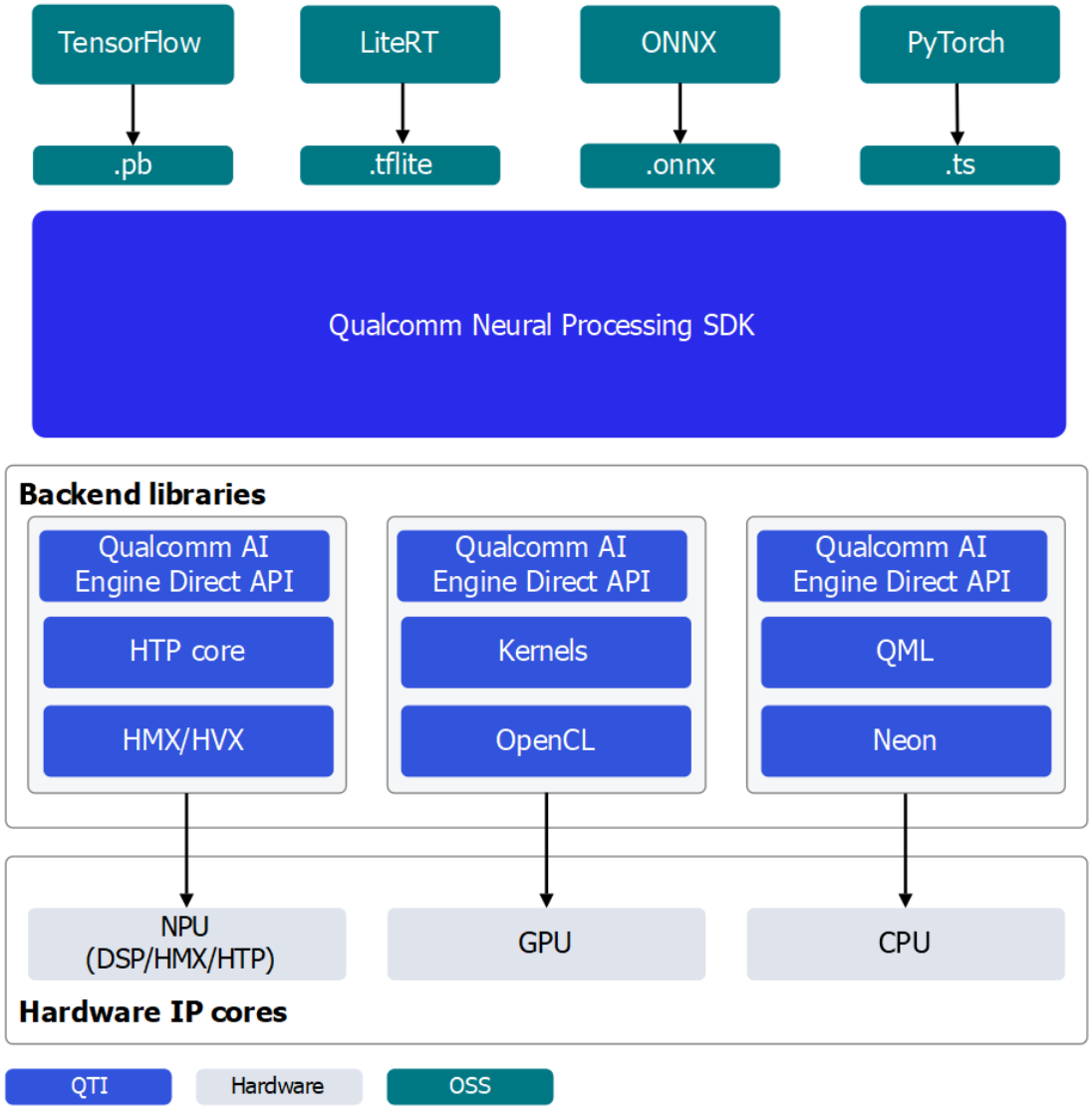
→ Run

API Reference

↗ C/C++

3.3 Qualcomm AI Runtime SDK

Qualcomm AI Runtime SDK is an all-in-one SDK to port ML models to run on Qualcomm hardware accelerators. The SDK contains Qualcomm Neural Processing Engine (also known as SNPE) and AI Engine Direct (also known as QNN) which offer tools to convert and quantize models trained in PyTorch and TensorFlow and runtimes to run these models on CPU, GPU, and HTP. Learn more about SNPE and QNN



Setup

→ Download SDK

→ Setup SDK

Customize

→ Convert model (SNPE)

→ Quantize model (SNPE)

→ Convert and quantize model (QNN)

→ Generate model (QNN)

Run inference

→ Run model (SNPE)

→ Run model (QNN)

API reference

→ Qualcomm Neural Processing Engine C/C++

→ Qualcomm AI Engine Direct C/C++

Install Qualcomm AI Runtime SDK

Qualcomm AI Runtime SDK requires an Ubuntu 22.04 host computer.

Note: If the host computer uses Windows or macOS operating system software, install a virtual machine. Subsequent steps must be run in the virtual machine running Ubuntu 22.04 LTS.

The Qualcomm AI Runtime SDK workflow has been validated with Ubuntu 22.04 running on bare metal or inside a Virtual Machine in the following configurations.

Note: Ubuntu 22.04 running inside a VM on macOS on Arm|reg| architecture isn't supported.

Download method	Prerequisites	Version available
Direct Download	No prerequisites for download	2.32.6.250402 (SDK updated every quarter)
Qualcomm Package Manager	• A valid Qualcomm ID. • Qualcomm Package Manager tool.	SDK updated every month

Direct download

Download the Qualcomm AI Runtime SDK. Once downloaded, extract or unzip the SDK.

Note: The SDK version hosted at the above link updates quarterly.

```
unzip v2.32.6.250402.zip
```

```
cd qairt/2.32.6.250402
```

```
export QAIRT_ROOT=`pwd`
```

Once finished, proceed to Set up Qualcomm AI Runtime SDK

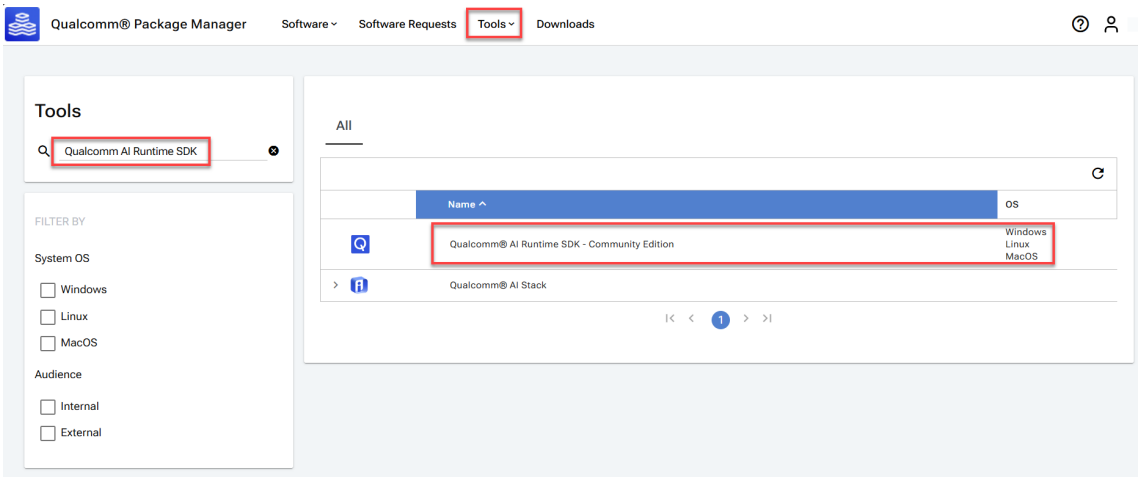
Qualcomm Package Manager

Qualcomm AI Runtime SDK is available for download through Qualcomm Package Manager (QPM). This section demonstrates how to download Qualcomm AI Runtime SDK using QPM.

To download QPM3, install QPM3 from https://qpm.qualcomm.com/#/main/tools/details/QPM3

Note: To download Qualcomm AI Runtime SDK from Qualcomm Package Manager, ensure that you have registered for a Qualcomm ID. If you don't have a Qualcomm ID, you will be prompted to register. Then follow the instructions below to download and install the SDK.

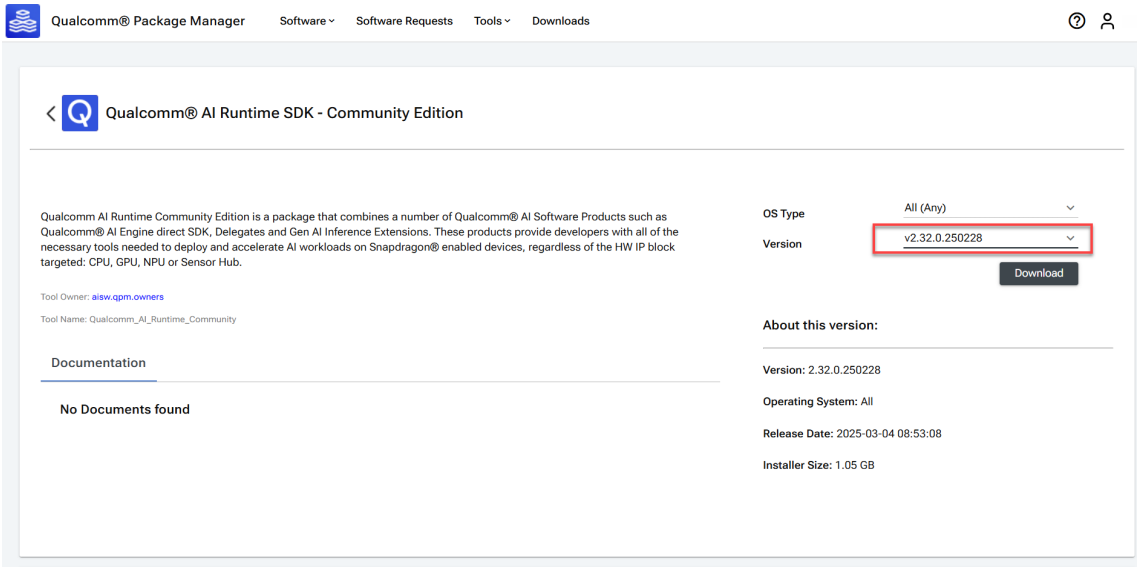
1. Go to Qualcomm Package Manager and sign in using your Qualcomm ID.
2. Go to the **Tools** tab and search for *Qualcomm AI Runtime SDK* in the left pane. From the filtered results, select **Qualcomm AI Runtime - Community Edition**.



3. Choose *v2.32.6.250402** from the Version drop-down list.

   If using the Qualcomm Package Manager desktop tool, the **Extract** button is present instead of **Download**. This automatically installs the Qualcomm AI Runtime SDK.

   If you are using the web interface, the **Download** button will download the Qualcomm AI Runtime SDK zip file.



Note: These instructions use version 2.32.6.250402 as an example.

4. Install Qualcomm AI Runtime SDK using the QPM CLI tool.

   a. Sign in using QPM CLI.

```
qpm-cli --login <username>
```

```
Password:
[Info] : Login is successful
[Info] : Fetching product catalog ....
[Info] : Product catalog refreshed successfully
```

   b. Activate your SDK license.

```
qpm-cli --license-activate qualcomm_ai_runtime_sdk
```

```
[Info] : Activating license : qualcomm_ai_runtime_sdk
[Info] : License activation is successful ActivationId=[fb]
```

   c. Download, extract, and install the SDK.

```
qpm-cli --install qualcomm_ai_runtime_sdk --version 2.32.6.
250402
```

```
[Info] : Current logged in user is <user name>
[Info] : Downloading qualcomm_ai_runtime_sdk 2.32.6.250402 ..
..

------------------------------------------------------

Access to and use of tools managed by the Qualcomm Package
Manager are subject to the terms and conditions of the
corresponding agreement(s) in place with Qualcomm
Technologies, Inc. or its affiliates.
Unauthorized access or use is prohibited. Information such as
tool version, operating system, user ID, company ID, IP
address, computer mac address, date, timestamp, or features
and functions of our tools that you use may be collected for
internal business purposes or tool improvements and is
subject to the Qualcomm Privacy Policy [http://www.qualcomm.
com/site/privacy].
By accessing or using this tool, you agree to the foregoing.
Copyright (c) 2018-2025 Qualcomm Technologies, Inc. All
rights reserved.

Accept and continue with installation [y/n] :

[Warning] : Install can't be used for ExtractOnly Type
products, proceeding with extract command instead
[Info] : Config File Present
[Info] : Step 1 of 7: Checking environment
[Info] : Step 2 of 7: Checking previous version
[Info] : Step 3 of 7: Checking dependencies
[Info] : Step 4 of 7: Preparing system
[Info] : Step 5 of 7: Extracting files
[Info] : Step 6 of 7: Configuring system
[Info] : Step 7 of 7: Finishing installation
[Info] : SUCCESS: Installed qualcomm_ai_runtime_sdk.Core at /
opt/qcom/aistack/qairt/2.32.6.250402
```

Set up Qualcomm AI Runtime SDK

Prerequisites:

- Host OS: **Ubuntu22.04 LTS**

Note: If the host computer uses Windows or Mac OS, install a virtual machine.

Subsequent steps must be run in the virtual machine running Ubuntu 22.04 LTS.

1. Enable SSH in Permissive mode to securely sign in to the host device. For instructions, see How to SSH
2. For the rest of the document, the `QAIRT_ROOT` environment variable represents the full path to the Qualcomm AI Runtime SDK root directory.
   - If installed using the direct download method, `${QAIRT_ROOT}` is the path to the unzipped SDK.

     ```
     unzip 2.32.6.250402.zip
     ```

     ```
     cd qairt/2.32.6.250402
     ```

     ```
     export QAIRT_ROOT=`pwd`
     ```

   - If installed using QPM, the SDK is installed under `/opt/qcom/aistack/qairt/<version>`

     ```
     export QAIRT_ROOT=/opt/qcom/aistack/qairt/2.32.6.250402/
     ```

3. Python: **v3.10**

   If Python is already installed, ensure that the environment path is updated with the Python 3.10 path. If Python 3.10 isn't installed on your system, you may install it with the following commands:

   ```
   sudo apt-get update
   ```

   ```
   sudo apt-get install python3.10 python3-distutils libpython3.10
   ```

4. System Dependencies:

   Note: Run the following command as administrator/root to install the system libraries. The command takes time to fully complete.

```
sudo bash ${QAIRT_ROOT}/bin/check-linux-dependency.sh
```

5. Virtual environment (VENV)

Note: <venv_path> is the path for new the virtual environment.

```
sudo apt-get install python3.10-venv
```

```
python3.10 -m venv "<venv_path>"
```

```
source <venv_path>/bin/activate
```

6. Run the following script to check and install missing dependencies:

```
python3 -m pip install --upgrade pip
```

```
${QAIRT_ROOT}/bin/check-python-dependency
```

```
Summary:

Package                 Recommended     Installed
~~~~~~~                 ~~~~~~~~~~~     ~~~~~~~~~
absl-py                 2.1.0           2.1.0
attrs                   23.2.0          23.2.0
dash                    2.12.1          2.12.1
decorator               4.4.2           4.4.2
invoke                  1.7.3           1.7.3
joblib                  1.4.0           1.4.0
jsonschema              4.19.0          4.19.0
lxml                    5.2.1           5.2.1
mako                    1.1.0           1.1.0
matplotlib              3.3.4           3.3.4
mock                    3.0.5           3.0.5
numpy                   1.26.4          1.26.4
opencv-python           4.5.4.58        4.5.4.58
optuna                  3.3.0           3.3.0
packaging               24.0            24.0
pandas                  2.0.1           2.0.1
paramiko                3.4.0           3.4.0
pathlib2                2.3.6           2.3.6
pillow                  10.2.0          10.2.0
plotly                  5.20.0          5.20.0
```

```
protobuf              3.19.6          3.19.6
psutil                5.6.4           5.6.4
pytest                8.1.1           8.1.1
pyyaml                5.3             5.3
scikit-optimize       0.9.0           0.9.0
scipy                 1.10.1          1.10.1
six                   1.16.0          1.16.0
tabulate              0.9.0           0.9.0
typing-extensions     4.10.0          4.10.0
xlsxwriter            1.2.2           1.2.2
```

Setup ML Frameworks

To convert ML models trained on different frameworks into intermediate representations consumable by the Qualcomm AI Runtime SDK, you may need to download and install the corresponding frameworks on your host computer.

This Qualcomm AI Runtime SDK release is verified to work with the following versions of the ML training frameworks:

	Version
TensorFlow	2.10.1
TFLite	2.3.0
PyTorch	1.13.1
ONNX	1.16.1

- Install TensorFlow

```
pip install tensorflow==2.10.1
```

- Install TFLite

```
pip install tflite==2.3.0
```

- Install PyTorch

```
pip install torch==1.13.1+cpu torchvision==0.14.1+cpu
torchaudio==0.13.1 --extra-index-url https://download.pytorch.
org/whl/cpu
```

- Install ONNX

```
pip install onnx==1.16.1 onnxruntime==1.18.0 onnxsim==0.4.36
```

> **Attention:** The TensorFlow and PyTorch versions mentioned above have different package requirements.
>
> Create separate virtual environments for the two frameworks.

Setup Qualcomm AI Runtime SDK environment

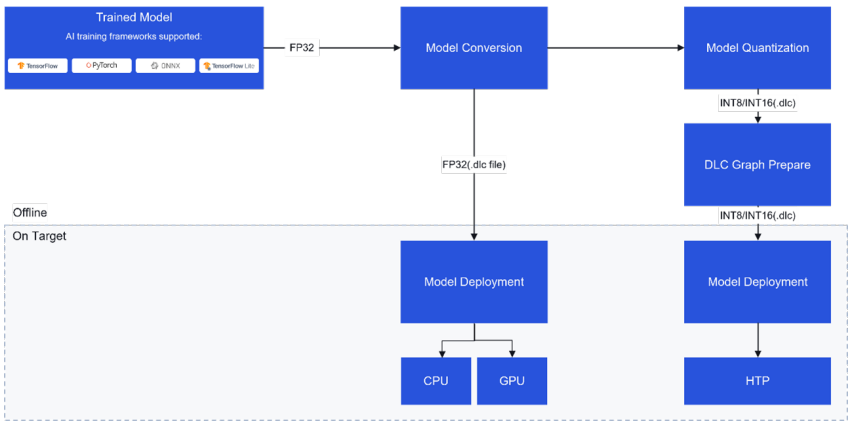
Source the environment setup script provided by Qualcomm AI Runtime SDK to ensure all necessary tools and libraries are available in `$PATH` for the workflow.

```
source ${QAIRT_ROOT}/bin/envsetup.sh
```

Port models

Neural Processing Engine

Port a model using Qualcomm Neural Processing Engine SDK



Model conversion

A pretrained floating point, 32-bit precision model from PyTorch, Onnx, TensorFlow, or TFLite is input to SNPE converter tools (snpe-<framework>-to-dlc) to convert the model to a Qualcomm- specific intermediate representation of the model called a deep learning container (DLC).

In addition to the input model from a source framework, the converters require additional details about the input model, such as the input node name, its corresponding input dimensions, and any output tensor names (for models with multiple outputs).

Refer to converters for all available configurable parameters or see the command line help by running:

```
snpe-<framework>-to-dlc --help
```

```
required arguments:

-d INPUT_NAME INPUT_DIM, --input_dim INPUT_NAME INPUT_DIM
    The names and dimensions of the network input layers
specified in the format
    [input_name comma-separated-dimensions], for example:
    'data' 1,224,224,3
     Note that the quotes should always be included in order to
handle special
     characters, spaces, etc.
     For multiple inputs specify multiple --input_dim on the
command line like:
     --input_dim 'data1' 1,224,224,3 --input_dim 'data2' 1,50,
100,3
--out_node OUT_NAMES, --out_name OUT_NAMES
     Name of the graph's output Tensor Names. Multiple output
names should be
     provided separately like:
     --out_name out_1 --out_name out_2
--input_network INPUT_NETWORK, -i INPUT_NETWORK
     Path to the source framework model.
```

Note: If the `yaml` package is not present in your working environment, install it using the following command:

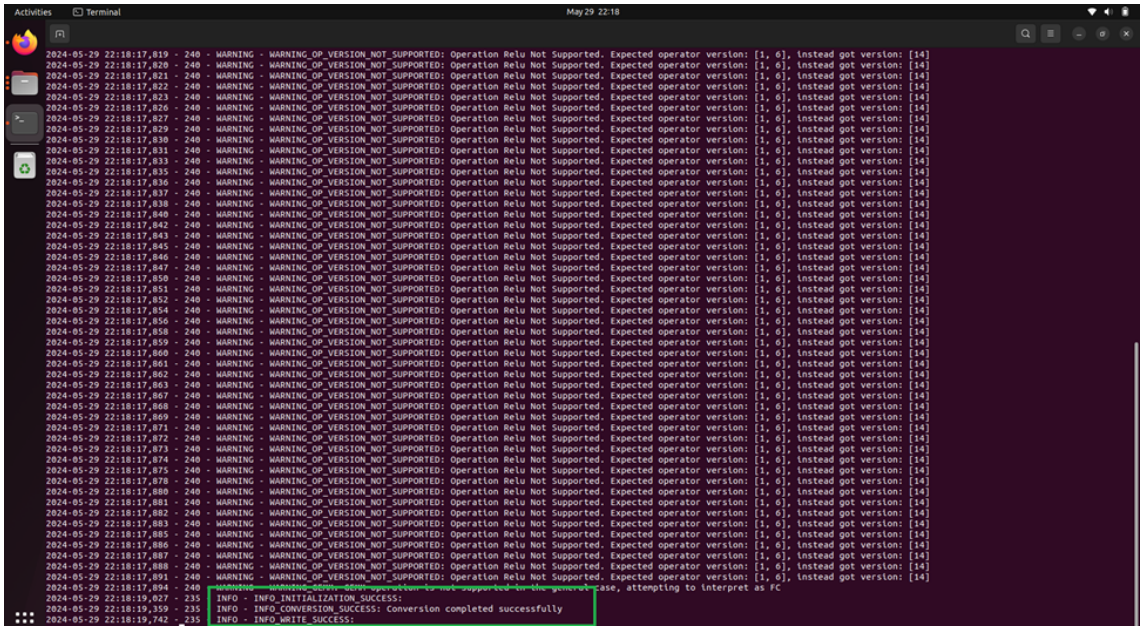
```
pip install pyyaml
```

The following example uses an ONNX model (*inception_v3_opset16.onnx*) downloaded from the ONNX Model Zoo.

Download the model as `inception_v3.onnx` to your workspace. In this example, we download the model to the `~/models` directory.

Run the following command to generate the `inception_v3.dlc` model.

```
${QAIRT_ROOT}/bin/x86_64-linux-clang/snpe-onnx-to-dlc --input_
network ~/models/inception_v3.onnx --output_path ~/models/
inception_v3.dlc --input_dim 'x' 1,3,299,299
```



Model quantization

To run a model on Hexagon Tensor Processor (HTP), the converted DLC must be quantized. SNPE offers a tool (snpe-dlc-quant) to quantize a DLC model to INT8/INT16 DLC using its own quantization algorithm. More information about SNPE quantization is available here.

The quantization process in SNPE is requires two-steps:

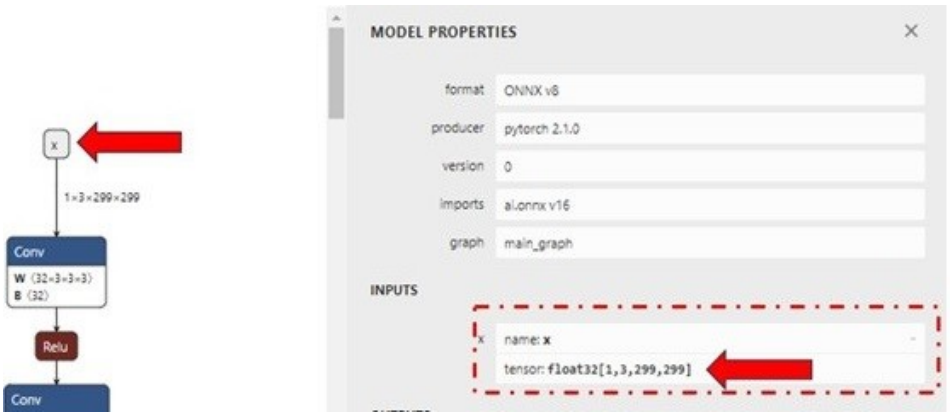
1. Quantization of weights and biases within the model.
    - Quantization of weights and biases is a static step, i.e., no additional input data is required from the user.
2. Quantization of activation layers (or layers with no weights).
    - Quantizing activation layers requires a set of input images from a training dataset as calibration data.
    - These calibration dataset images are input as a list of preprocessed image files in .raw format. The file sizes of these input .raw files must match the input size of the model.

Inputs to snpe-dlc-quant are a converted DLC model and a plain text file with the paths to the calibration dataset images. This input list holds paths to preprocessed images saved as NumPy arrays in .raw format. The size of the preprocessed image must match the input resolution of the model.

The output of the snpe-dlc-quant tool is a quantized DLC.

```
[ --input_dlc=<val> ]
             Path to the dlc container
             containing the model for which fixed-point encoding
metadata should be generated.
             This argument is required.
[ --input_list=<val> ] Path to a file
             specifying the trial inputs. This file should be a
plain text file, containing one
             or more absolute file paths per line. These files
will be taken to constitute the
             trial set. Each path is expected to point to a
binary file containing one trial
             input in the 'raw' format, ready to be consumed by
the tool without any further
             modifications. This is similar to how input is
provided to snpe-net-run
             application.
[ --output_dlc=<val> ] Path at which the
             metadata-included quantized model container should
be written. If this argument is
             omitted, the quantized model will be written at
             <unquantized_model_name>_quantized.dlc.
```

Note: Use Netron graph visualization tool to identify the model's input/output layer dimensions.



For demo purposes, we can evaluate the quantization process with random input files. The input file can be generated using a simple Python script shown below for the `inception_v3.onnx` model. Save the script as `generate_random_input.py` in your workspace `~/models/` directory and run it using `python ~/models/generate_random_input.py` on your host computer.

The following example Python code creates an input_list that holds paths to calibration dataset images used to quantize the model.

```
import os
import numpy as np

input_path_list =[]
BASE_PATH = "/tmp/RandomInputsForInceptionV3"

if not os.path.exists(BASE_PATH):
    os.mkdir(BASE_PATH)

# generate 10 random inputs and save as raw
NUM_IMAGES = 10

#binary files
for img in range(NUM_IMAGES):
    filename = "input_{}.raw".format(img)
    randomTensor = np.random.random((1, 299, 299, 3)).astype(np.
float32)
    filename = os.path.join(BASE_PATH, filename)
    randomTensor.tofile(filename)
    input_path_list.append(filename)

#for saving as input_list text
with open("input_list.txt", "w") as f:
    for path in input_path_list:
        f.write(path)
        f.write("\n")
```

The above script generates 10 sample input files saved in the `/tmp/RandomInputsForInceptionV3/` directory and an `input_list.txt` file that contains the path to each sample generated.

Now that all needed inputs to the `snpe-dlc-quant` tool are available, the model can be quantized.

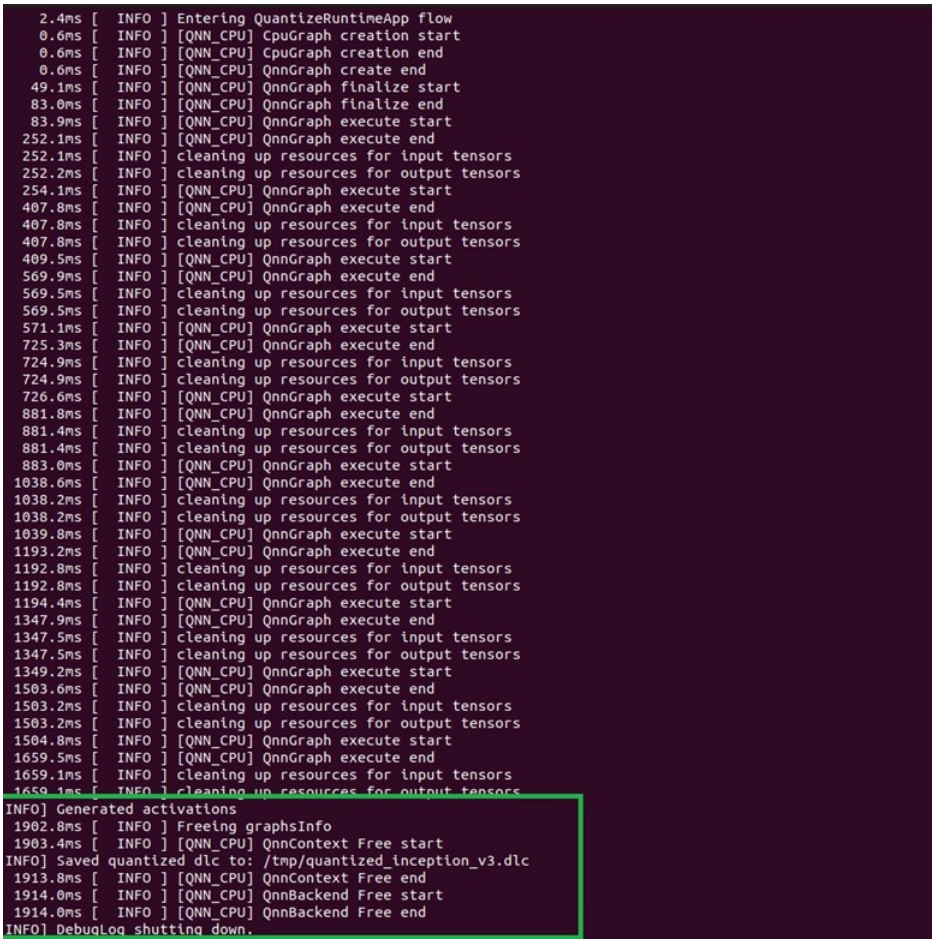
```
${QAIRT_ROOT}/bin/x86_64-linux-clang/snpe-dlc-quant --input_dlc
~/models/inception_v3.dlc --output_dlc ~/models/inception_v3_
quantized.dlc --input_list ~/models/input_list.txt
```

This generates a quantized inception_v3 DLC model (`inception_v3_quantized.dlc`). By default, the model is quantized for INT8 bit width.

Customize the quantization to use 16-bit instead of default INT8 by specifying the [`--act_bitwidth 16`] and/or [`--weights_bitwidth 16`] options to the `snpe-dlc-quant`

tool.

Refer to the snpe-dlc-quant tool documentation, or `run snpe-dlc-quant --help` to view all available customizations including quantization modes, optimizations, etc.



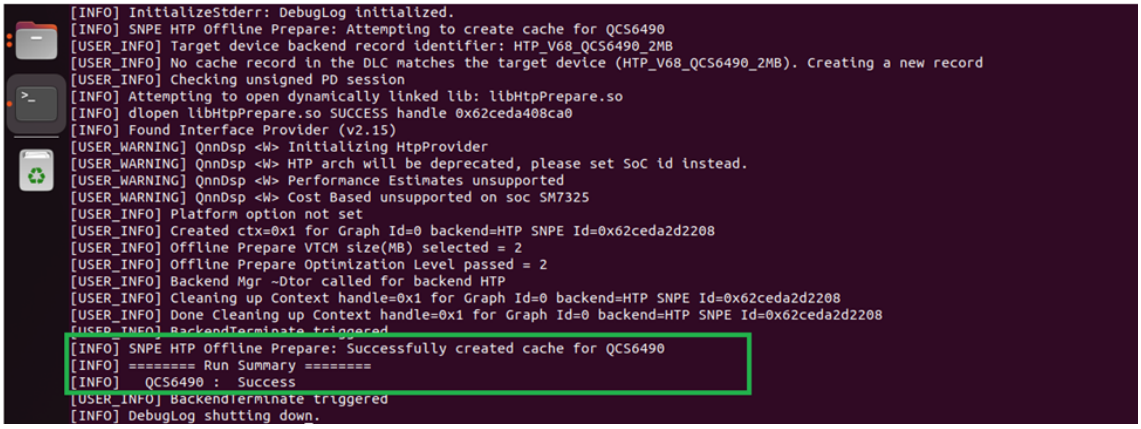
Model optimization

Quantized model DLC requires a graph preparation step that optimizes the model for execution on HTP. To prepare the model DLC to execute on HTP, SNPE provides a `snpe-dlc-graph-prepare` tool that takes a quantized model and hardware-specific details, such as chipset, as input.

Note: Optimizations for hardware, such as HTP, depend on the specific version of HTP present on the chipset. To ensure the correct set of optimizations are applied to the execution graph for optimal utilization of the HTP, it is important to provide the correct chipset ID to the `snpe-dlc-graph-prepare` tool.

Based on the HTP version and chipset ID, the tool creates a cache that contains an execution strategy to execute model DLC on the HTP hardware. Without this step, there will be additional overhead during network initialization as the SNPE runtime will have to create an execution strategy on the fly.

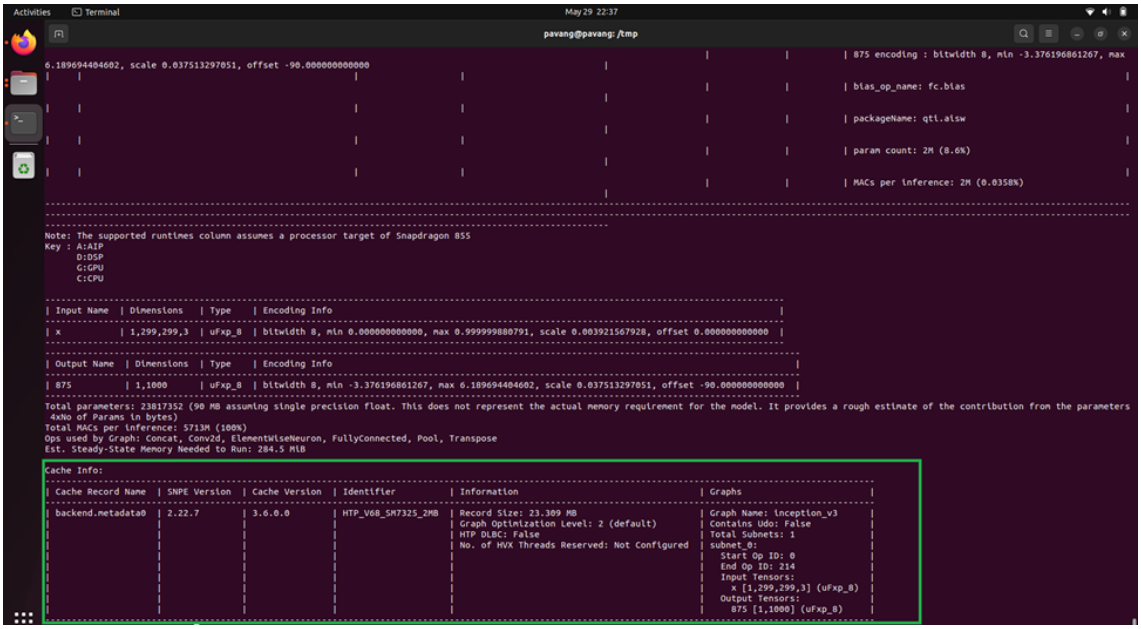
```
${QAIRT_ROOT}/bin/x86_64-linux-clang/snpe-dlc-graph-prepare --
input_dlc ~/models/inception_v3_quantized.dlc --output_dlc ~/
models/inception_v3_quantized_with_htp_cache.dlc --htp_socs
qcs6490
```



HTP cache information

Once the `snpe-dlc-graph-prepare` step is completed, the HTP cache record is added to the DLC. This cache information can be viewed using the snpe-dlc-info tool.

```
${QAIRT_ROOT}/bin/x86_64-linux-clang/snpe-dlc-info -i ~/models/
inception_v3_quantized_with_htp_cache.dlc
```



AI Engine Direct

Port a model using AI Engine Direct

Model conversion and quantization

A pretrained FP32 model from PyTorch, ONNX, TensorFlow, or TFLite is input to the QNN converter tool (`qnn-<framework>-converter`) to convert to a QNN graph representation in the form of a high-level readable C++ graph.

If accelerating the model on HTP, the model must be quantized. Model quantization can be done in the same step as conversion. A calibration dataset must be provided to perform this quantization step to perform static quantization.

To enable quantization along with conversion, use the `--input_list INPUT_LIST` option for static quantization.

For more information, see quantization support.

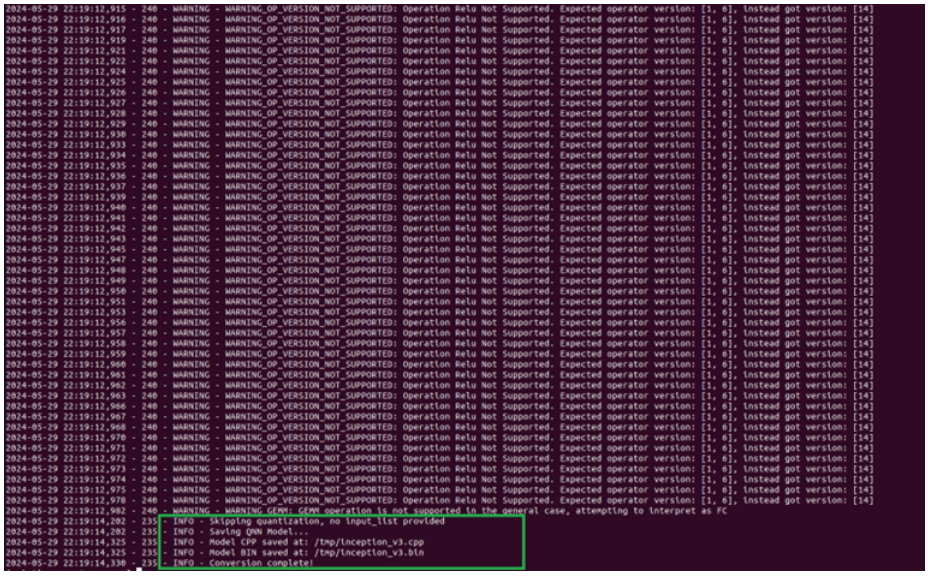
The following example uses an ONNX model (`inception_v3_opset16.onnx`) downloaded from the ONNX Model Zoo.

Download the model as `inception_v3.onnx` to your workspace. In this example, the model is downloaded to the `~/models` directory.

Model conversion: CPU backend

To convert the model to run on x86/Arm-based CPU, run the following command to generate `inception_v3.cpp` and `inception_v3.bin`.

```
${QAIRT_ROOT}/bin/x86_64-linux-clang/qnn-onnx-converter --input_
network ~/models/inception_v3.onnx --output_path ~/models/
inception_v3.cpp --input_dim 'x' 1,3,299,299
```



The `inception_v3.cpp` file contains a high-level graph representation of the converted model.

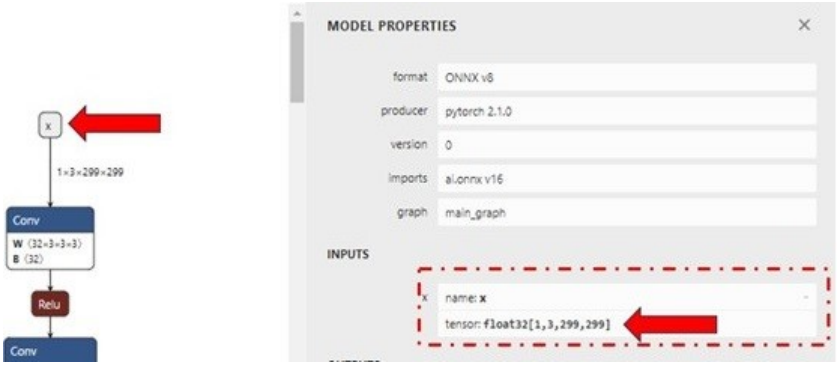
The `inception_v3.bin` file contains weights/biases from the model.

Model conversion and quantization: HTP backend

To run the model on HTP, the quantization step is required.

For quantization in AI Engine Direct (QNN) SDK, a representative dataset of 50 to 200 images from a training dataset are provided to the QNN converter as a calibration dataset. The images in the calibration dataset are preprocessed (resized, normalized, etc.) and saved as NumPy arrays in `.raw` format. The size of these input `.raw` files must match the input size of the model.

Note: Use the Netron graph visualization tool to identify the model's input/output layer dimensions.



For demonstration purposes, you can evaluate the quantization process with random input files. The input files can be generated using the Python script shown below for the `inception_v3.onnx` model. Save the script as `generate_random_input.py` in the `~/models/` directory and run it using `python ~/models/generate_random_input.py`.

The following Python code creates an input_list that contains the calibration dataset used to quantize the model.

```
import os
import numpy as np

input_path_list =[]

BASE_PATH = "/tmp/RandomInputsForInceptionV3"
```

```
if not os.path.exists(BASE_PATH):
    os.mkdir(BASE_PATH)

# generate 10 random inputs and save as raw
NUM_IMAGES = 10

#binary files
for img in range(NUM_IMAGES):
    filename = "input_{}.raw".format(img)
    randomTensor = np.random.random((1, 299, 299, 3)).astype(np.
float32)
    filename = os.path.join(BASE_PATH, filename)
    randomTensor.tofile(filename)
    input_path_list.append(filename)

#for saving as input_list text
with open("~/models/input_list.txt", "w") as f:
    for path in input_path_list:
        f.write(path)
        f.write("\n")
```

Run the following command to convert and quantize.

By default, the model is quantized for INT8 bit width. You can specify [`--act_bitwidth 16`] and/or [`--weights_bitwidth 16`] to use INT16 quantization.

```
${QAIRT_ROOT}/bin/x86_64-linux-clang/qnn-onnx-converter --input_
network ~/models/inception_v3.onnx --output_path ~/models/
inception_v3_quantized.cpp --input_list   ~/models/input_list.
txt --input_dim "x" 1,3,299,299
```

This generates `inception_v3_quantized.cpp` and `inception_v3_quantized.bin` files in the `~/models/` directory.

See qnn-<framework>-converter or run `qnn-<framework>-converter --help` to view all available customizations to quantization, including quantization modes, optimizations, etc.

Model compilation

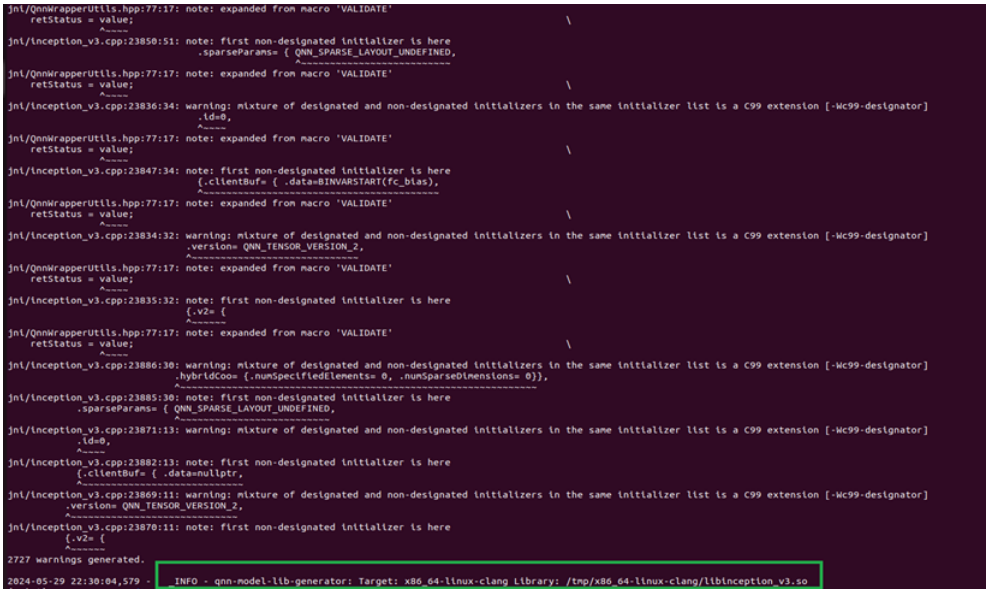
Once the conversion/quantization step is complete, `qnn-model-lib-generator` is used to compile the generated C++ graph into a shared object (.so) enabling the model to be dynamically loaded by an application to perform inference.

For x86, the Clang compiler toolchain is used to compile the C++ graph into a .so library. For a Linux Embedded device, such as QCS6490 and QCS9075, the appropriate compiler toolchain (`aarch64-oe-linux-gcc11.2`) must be used.

Compiling a model to run on x86

Generate a shared object model to run on an x86-based Linux machine with the following command. This generates `inception_v3.so` using the Clang-14 compiler toolchain to compile the C++ graph to a QNN model .so compatible with x86 host machines.

```
${QAIRT_ROOT}/bin/x86_64-linux-clang/qnn-model-lib-generator -c
~/models/inception_v3.cpp -b ~/models/inception_v3.bin -o ~/
models/libs/ -t x86_64-linux-clang
```



Compiling a model to run on target

When compiling a model for on-device execution (aarch64 architecture), it is important to use the right cross-compiler toolchain to ensure the compiled shared object (.so) is compatible with the device OS.

The following steps install the cross-compiler toolchain required to compile a model cpp file to a .so library. Instructions to install the appropriate cross-compiler toolchain are available under *Download and install the Platform eSDK <https://docs.qualcomm.com/bundle/resource/topics/80-70018-51/install-sdk.html#section-b5c-z3k-5bc>*.

After installing the platform eSDK, setup the cross-compiler environment in a new command line terminal.

1. Source the environment setup script under `$ESDK_ROOT`.

   ```
   source $ESDK_ROOT/environment-setup-armv8-2a-qcom-linux
   ```

2. Check if the environment is properly setup.

   ```
   echo $SDKTARGETSYSROOT
   echo $TARGET_PREFIX
   ```

   If the above environmental variables were not populated, repeat Step 1 in a new command line terminal.

3. Setup the QAIRT environment.

   ```
   conda activate qairt
   source ~/qairt/2.29.0.241129/bin/envsetup.sh
   ```

Compiling a model to run on Arm-based CPU

Once the cross-compiler is setup, use the following command to generate `libinception_v3.so` in `~/model/libs/aarch64-oe-linux-gcc11.2`. Provide this location to the `qnn-model-lib-generator` tool through a command line argument.

Note: The compiler toolchain used here is `aarch64-oe-linux-gcc11.2`.

```
${QAIRT_ROOT}/bin/x86_64-linux-clang/qnn-model-lib-generator -c
~/models/inception_v3.cpp -b ~/models/inception_v3.bin -o ~/
models/libs -t aarch64-oe-linux-gcc11.2
```

Compiling a model to run on HTP

To run the model on HTP, the following command generates `libinception_v3_quantized.so` in `~/models/libs/aarch64-oe-linux-gcc11.2`.

Note: The compiler toolchain used here is `aarch64-oe-linux-gcc11.2`.

```
${QAIRT_ROOT}/bin/x86_64-linux-clang/qnn-model-lib-generator -c
~/models/inception_v3_quantized.cpp -b ~/models/inception_v3_
quantized.bin -o ~/models/libs/ -t aarch64-oe-linux-gcc11.2
```

Run models

Neural Processing Engine

Deploy a model using Neural Processing Engine

A model DLC (quantized or non-quantized) can be deployed via a SNPE enabled app (an application written using SNPE C/C++ APIs). SNPE offers APIs to load a DLC, select a runtime to run the model, and perform inference, etc.

SNPE provides a prebuilt snpe-net-run tool (application written using C APIs) that can load an arbitrary model DLC and run it on provided inputs.

- **Model file:** DLC model file generated by the SNPE converter tool or `snpe-dlc-graph-prepare` tool (if running on HTP).
- **Input list:** Text file, like the `input_list.txt` file used during quantization, except input raw files in this list are used for inference. For simplicity in this example, the same input list used for quantization is used for inference.
- **Runtime:** User must select a specific runtime to run the model on target. Available runtime options are CPU, GPU, and DSP (HTP).

Note: See `snpe-net-run --help` for more details.

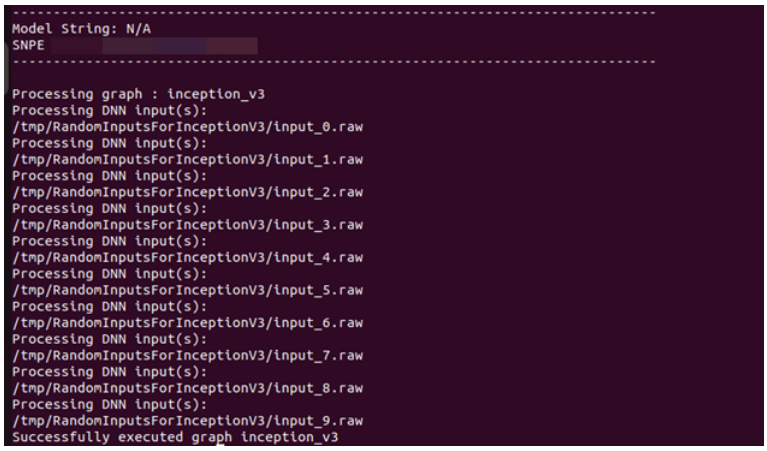
Run SNPE .dlc: x86 host computer

Converted SNPE .dlc can be run using the `snpe-net-run` tool which takes a .dlc and input list as arguments. Running SNPE DLC on x86 is purely for debugging purposes.

For example, the following command generates loads the `inception_v3.dlc` model and runs the model on x86 CPU. It generates output files to `/output_x86/`.

Note: The default runtime in SNPE is CPU. When executing a model using `snpe-net-run`, there is no need to specify the CPU runtime.

```
${QAIRT_ROOT}/bin/x86_64-linux-clang/snpe-net-run --container ~/
models/inception_v3.dlc --input_list ~/models/input_list.txt --
output_dir ~/models/output_x86
```



Prepare a SNPE model to run on target

To run the model on target, `snpe-net-run` requires the model .dlc, SNPE runtime libraries, and input list to run inferences to generate outputs.

Note: Before running on target, ensure that the SNPE SDK binaries and libraries are pushed to the target.

Use artifacts from `${QAIRT_ROOT}/lib/aarch64-oe-linux-gcc11.2` and `${QAIRT_ROOT}/bin/aarch64-oe-linux-gcc11.2`

Use the correct DSP Hexagon architecture libraries for Qualcomm evaluation kits.

QCS8275

Use v75 libraries

QCS9075

Use v73 libraries

QCS6490

Use v68 libraries

File	Source location
snpe-net-run	${QAIRT_ROOT}/bin/aarch64-oe-linux-gcc11.2
libSNPE.so	${QAIRT_ROOT}/lib/aarch64-oe-linux-gcc11.2/
libSnpeHtpPrepare.so	${QAIRT_ROOT}/lib/aarch64-oe-linux-gcc11.2
libSnpeHtpV68Stub.so	${QAIRT_ROOT}/lib/aarch64-oe-linux-gcc11.2
libSnpeHtpV68Skel.so	${QAIRT_ROOT}/lib/hexagon-<dsp-arch>/unsigned
Inception_v3_quantized_with_htp_cache.dlc	~/models
Inception_v3.dlc	~/models
input_list.txt	~/models
Inception V3 sample input images	/tmp/RandomInputsForInceptionV3

The following scp commands needs to be executed on the host computer to copy SNPE SDK libraries and binaries to the device.

Note: Replace `<dsp-arch>` with `75` for QCS8275, `73` for QCS9075, and `68` for QCS6490.

```
scp ${QAIRT_ROOT}/bin/aarch64-oe-linux-gcc11.2/snpe-net-run
root@[ip-addr]:/opt/
```

```
scp ${QAIRT_ROOT}/lib/aarch64-oe-linux-gcc11.2/libSNPE.so
root@[ip-addr]:/opt/
```

```
scp ${QAIRT_ROOT}/lib/aarch64-oe-linux-gcc11.2/libSnpeHtpV<dsp-
arch>Stub.so root@[ip-addr]:/opt/
```

```
scp ${QAIRT_ROOT}/lib/aarch64-oe-linux-gcc11.2/
libSnpeHtpPrepare.so root@[ip-addr]:/opt/
```

```
scp ${QAIRT_ROOT}/lib/hexagon-v<dsp-arch>/unsigned/libSnpeHtpV
<dsp-arch>Skel.so root@[ip-addr]:/opt/
```

```
scp ~/models/inception_v3.dlc root@[ip- addr]:/opt/
```

```
scp ~/models/inception_v3_quantized_with_htp_cache.dlc root@[ip-
addr]:/opt/
```

```
scp ~/models/input_list.txt root@[ip- addr]:/opt/
```

```
scp /tmp/RandomInputsForInceptionV3 root@[ip- addr]:/tmp/
```

```
ssh root@[ip-addr]
```

```
cd /opt
```

The above steps prepared the device for model execution. The following sections provide details on running the model on the available runtimes.

Run SNPE .dlc: Arm-based CPU

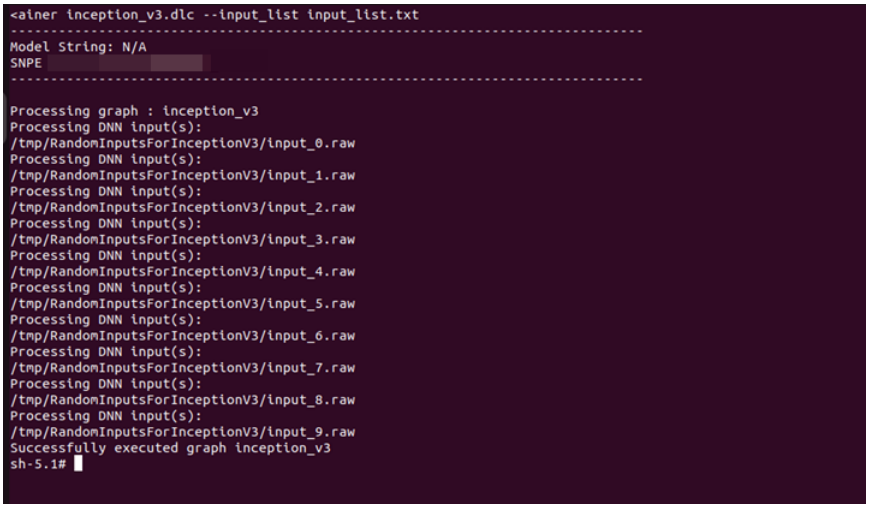
Setup environment variables before executing the model to ensure SNPE binaries and libraries are accessible to run the model.

Note: The default runtime in SNPE is CPU. When executing a model using `snpe-net-run`, there is no need to specify the CPU runtime.

```
export LD_LIBRARY_PATH=/opt:$LD_LIBRARY_PATH
```

```
export PATH=/opt/:$PATH
```

```
snpe-net-run --container inception_v3.dlc --input_list input_
list.txt --output_dir output_cpu
```



Run SNPE DLC: GPU

To run a model on the GPU runtime, use `inception_v3.dlc` and save outputs in `output_gpu`.

Note: To run on GPU, specify the runtime with the `--use_gpu` command line argument.

```
export LD_LIBRARY_PATH=/opt:$LD_LIBRARY_PATH
```

```
export PATH=/opt:$PATH
```

```
snpe-net-run --container inception_v3.dlc --input_list input_
list.txt --output_dir output_gpu --use_gpu
```

Run SNPE DLC: HTP

To run a model on the HTP backend, use `inception_v3_quantized_with_htp_cache.dlc` and save outputs in `output_htp`.

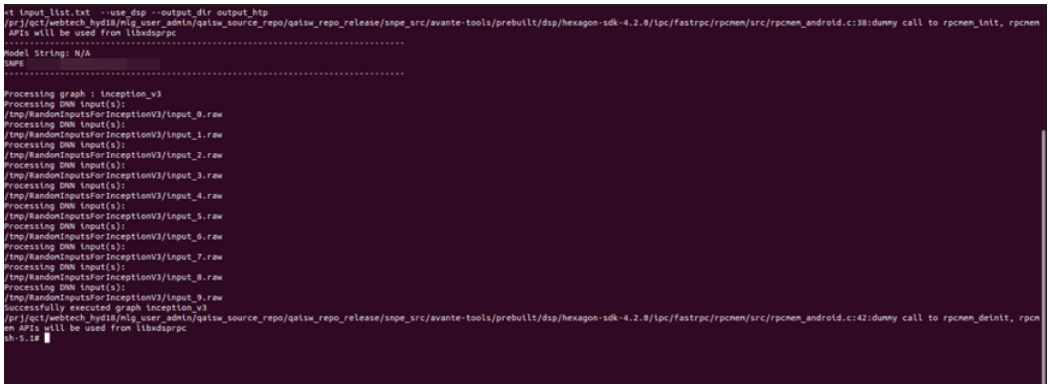
Note: To run on HTP, specify the runtime with the `--use_dsp` command line argument.

```
export LD_LIBRARY_PATH=/opt:$LD_LIBRARY_PATH
```

```
export PATH=/opt:$PATH
```

```
export ADSP_LIBRARY_PATH="/opt;/usr/lib/rfsa/adsp;/dsp"
```

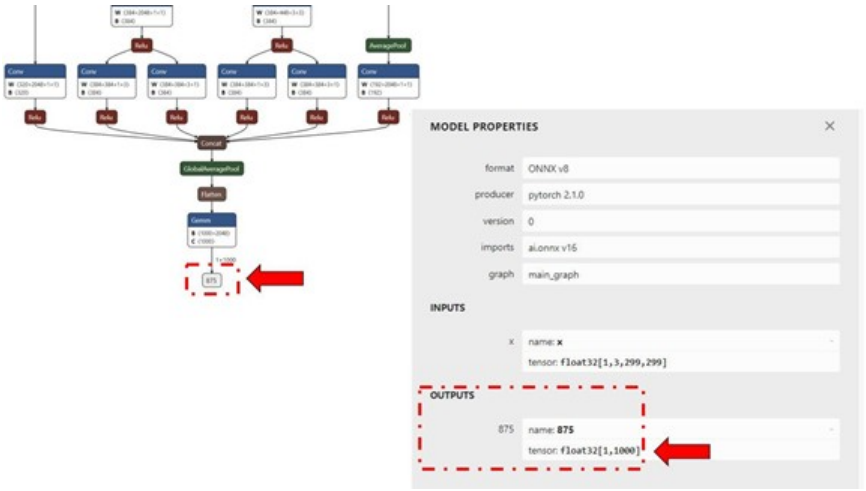
```
snpe-net-run --container inception_v3_quantized_with_htp_cache.
dlc --input_list input_list.txt --output_dir output_htp --use_
dsp
```



Validate output

For each input raw file fed to snpe-net-run (via the input_list file), an output folder is generated that contains output tensor saved as raw file(s) whose size will match the model's output layer as shown in the image below.

Note: The Netron tool was used to visualize the model.



In the `inception_v3` example, the output raw file is a binary file that contains probabilities for 1000 classification classes.

We use a Python script to read the file as a NumPy array to perform postprocessing and output validation. The example below checks whether HTP prediction is the same as CPU prediction.

1. Copy the output files from the target device to the host computer so the outputs can be validated.

```
ssh root@[ip-addr]
```

```
cd /opt/
```

```
scp -r /opt/output_cpu user@host-ip:<path>
```

```
scp -r /opt/output_htp user@host-ip:<path>
```

Note: Use `<path>` from the above commands in the following (`compare.py`) python script.

---

2. Once outputs from on-device inference are copied to the host computer, prepare a script to load the output tensors saved in `output_htp` and `output_cpu` to compare.

   In this example, both `output_htp` and `output_cpu` are copied to the `~/models` directory.

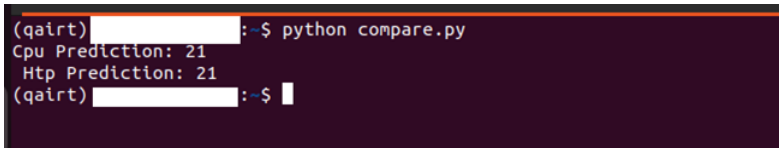
   The following is an example script to compare output from one of the example inputs used. Outputs are saved to the `/opt/` directory on the host computer. Outputs from `snpe-net-run` can be loaded into NumPy `ndarrays` using the `numpy.fromfile(...)` API.

Note: By default, `snpe-net-run` saves the output tensors to NumPy files in **float32** format.

```
# python postprocessing script (compare.py) import numpy as
np

import numpy as np

htp_output_file_path = "<path>/output_htp/Result_1/875.raw"
cpu_output_file_path = "<path>/output_cpu/Result_1/875.raw"
```

```
htp_output = np.fromfile(htp_output_file_path, dtype=np.
float32)
htp_output = htp_output.reshape(1,1000)

cpu_output = np.fromfile(cpu_output_file_path, dtype=np.
float32)
cpu_output = cpu_output.reshape(1,1000)

# np.argmax gives the cls_id with highest probability from
tensor.

cls_id_htp = np.argmax(htp_output)
cls_id_cpu = np.argmax(cpu_output)

# Let's compare CPU output vs HTP output

print("Cpu prediction {} \n Htp Prediction {}".format(cls_id_
cpu, cls_id_htp))
```

Output



Deployment using SNPE APIs

SNPE SDK provides C/C++ APIs to create/develop applications that run a model on chosen hardware (CPU, GPU, or HTP) with acceleration. A See the sample application that demonstrates SNPE C/C++ APIs.

Deployment using Qualcomm Intelligent Multimedia SDK (IM SDK)

To improve the developer experience when building entire use case pipelines (stream from camera, preprocess images, perform inference, etc.), SNPE has been integrated into the Qualcomm IM SDK as a plugin qtimlsnpe.

The plugin has been developed on top of the SNPE C APIs and provides SNPE capabilities (load and run models). With the `qtimlsnpe` plugin, you can use your converted model DLC in a Qualcomm IM SDK pipeline to realize the use case.

See Develop your own AI/ML application for instructions on how to deploy SNPE DLC using the Qualcomm IM SDK.

AI Engine Direct

Deploy a model using AI Engine Direct

A model .so (quantized or non-quantized) can be deployed through a QNN enabled app (an application written using QNN C/C++ APIs). QNN offers APIs to load a model .so dynamically and run the model on hardware with the selected backend.

QNN provides a prebuilt tool (qnn-net-run) that can dynamically load this model .so and perform inference on a selected backend using provided inputs.

For CPU, GPU, or HTP execution, `qnn-net-run` requires the following arguments:

- **Model file:** .so file generated by `qnn-model-lib-generator`
- **Backend file:** .so file for the targeted backend
  - `libQnnCpu.so` for the CPU backend.
  - `libQnnGpu.so` for the GPU backend.
  - `libQnnHtp.so` for the HTP backend.
- **Input list:** Text file like the input_list.txt file used during quantization, except input raw files in this list are used for inference. For simplicity, this example uses the same input list used for quantization for inference.

Run QNN .so: x86 host computer

Converted QNN .so can be run using the `qnn-net-run` too which takes a model .so, backend library .so, and input list as arguments.

For example, the following command loads the `libinception_v3.so` model and runs the model on x86 CPU. It generates output files to `~/models/output_x86/`.

```
${QAIRT_ROOT}/bin/x86_64-linux-clang/qnn-net-run --model ~/
models/libs/x86_64-linux-clang/libinception_v3.so --backend $
{QAIRT_ROOT}/lib/x86_64-linux-clang/libQnnCpu.so --input_list
input_list.txt --output_dir ~/models/output_qnn_x86
```

Prepare a QNN model to run on target

To run the model on target, `qnn-net-run`, requires the model .so, QNN binaries and runtime libraries, and input list to run inferences to generate outputs.

Note: Before running on target, ensure that the QNN SDK binaries and runtime libraries are pushed to the target.

Use artifacts from `${QAIRT_ROOT}/bin/aarch64-oe-linux-gcc11.2` and `${QAIRT_ROOT}/lib/aarch64-oe-linux-gcc11.2`

Use the correct DSP Hexagon architecture libraries for Qualcomm evaluation kits.

QCS8275

Use v75 libraries

QCS9075

Use v73 libraries.

QCS6490

Use v68 libraries.

File	Source location
qnn-net-run	${QAIRT_ROOT}/bin/aarch64-oe-linux-gcc11.2
libQnnHtp.so	${QAIRT_ROOT}/lib/aarch64-oe-linux-gcc11.2/libQnnHtp.so
libQnnCpu.so	${QAIRT_ROOT}/lib/aarch64-oe-linux-gcc11.2/libQnnCpu.so
libQnnGpu.so	${QAIRT_ROOT}/lib/aarch64-oe-linux-gcc11.2/libQnnGpu.so
libQnnHtpPrepare.so	${QAIRT_ROOT}/lib/aarch64-oe-linux-gcc11.2
libQnnHtpV68Stub.so	${QAIRT_ROOT}/lib/aarch64-oe-linux-gcc11.2
libinception_v3_quantized.s o	~/models/libs/aarch64-oe-linux-gcc11.2
libinception_v3.so	~/models/libs/aarch64-oe-linux-gcc11.2
libQnnHtpV68Skel.so	${QAIRT_ROOT}/lib/hexagon-<dsp-arch>/unsigned
libqnnhtpv68.cat	${QAIRT_ROOT}/lib/hexagon-<dsp-arch>/unsigned
libQnnSaver.so	${QAIRT_ROOT}/lib/hexagon-<dsp-arch>/unsigned
libQnnSystem.so	${QAIRT_ROOT}/lib/hexagon-<dsp-arch>/unsigned

The following `scp` commands need to be run on the host computer to copy QNN SDK libraries and binaries to the device.

Note: Replace `<dsp-arch>` with `75` for QCS8275, `73` for QCS9075, and `68` for QCS6490.

```
scp ${QAIRT_ROOT}/bin/aarch64-oe-linux-gcc11.2/qnn-* root@[ip-addr]:/opt/
```

```
scp ${QAIRT_ROOT}/lib/aarch64-oe-linux-gcc11.2/libQnn*.so root@[ip-addr]:/opt/
```

```
scp ${QAIRT_ROOT}/lib/hexagon-v<dsp-arch>/unsigned/* root@[ip-addr]:/opt/
```

```
scp ~/models/libs/aarch64-oe-linux-gcc11.2/* root@[ip-addr]:/opt/
```

```
scp ~/models/input_list.txt root@[ip-addr]:/opt/
```

```
scp /tmp/RandomInputsForInceptionV3 root@[ip-addr]:/tmp/
```

```
ssh root@[ip-addr]
```

```
cd /opt/
```

The above steps prepared the device for model execution. The following sections provide details on running the model on the available runtimes.

Run QNN .so: Arm-based CPU

When running a model on an Arm-based CPU, `qnn-net-run` requires the model .so, backend .so library, and input list to run inference to generate outputs.

For RB3Gen2 targets, the .so must be cross-compiled with the `aarch64-oe-linux-gcc11.2` toolchain)

For example, the following commands write output files to `/opt/output_cpu/`.

```
export LD_LIBRARY_PATH=/opt/:$LD_LIBRARY_PATH
```

```
export PATH=/opt:$PATH
```

```
qnn-net-run --model libinception_v3.so --backend libQnnCpu.so --
input_list input_list.txt --output_dir output_cpu
```

Run a QNN model on GPU

When executing a model on Adreno GPU, `qnn-net-run` requires the model .so, backend .so library (`libQnnGpu.so`), and input list to run inference to generate outputs.

For example, the following commands write output files to `/opt/output_gpu/`.

```
export LD_LIBRARY_PATH=/opt/:$LD_LIBRARY_PATH
```

```
export PATH=/opt:$PATH
```

```
qnn-net-run --model libinception_v3.so --backend libQnnGpu.so --
input_list input_list.txt --output_dir output_gpu
```

Run QNN model on HTP backend

To run a model on HTP backend, use the `libquantized_inception_v3.so` library and `libQnnHtp.so` backend library and save outputs in the `output_htp` directory.

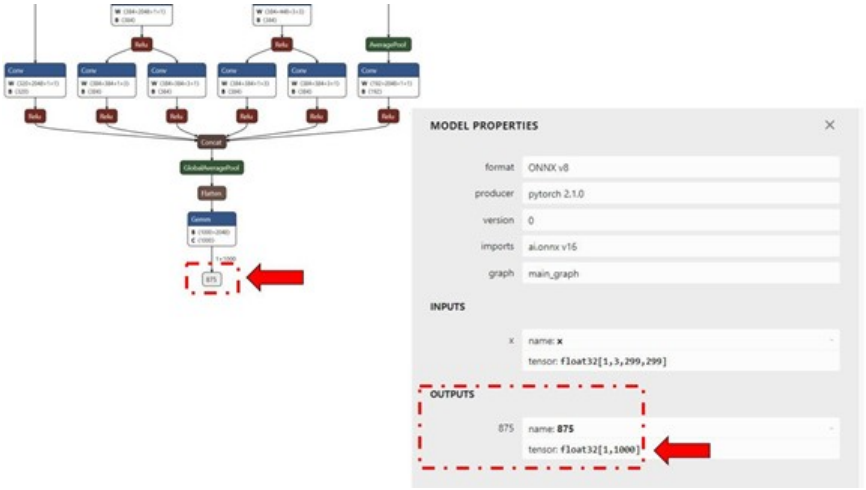
```
export LD_LIBRARY_PATH=/opt/:$LD_LIBRARY_PATH
export PATH=/opt:$PATH
```

```
export ADSP_LIBRARY_PATH="/opt/;/usr/lib/rfsa/adsp;/dsp"
```

```
qnn-net-run --model libinception_v3_quantized.so --backend
libQnnHtp.so --input_list input_list.txt --output_dir output_htp
```

Validate output

For each raw input file fed to `qnn-net-run` (through the input_list file), an output folder is generated that contains raw output file(s) whose size matches the model's output layer as shown in the image below.



In this `inception_v3` example, the raw output file is a binary file that contains probability for 1000 classification classes.

You can use a Python script to read the file as a NumPy array to perform postprocessing and validation of the output. The example script below checks whether HTP prediction is the same as CPU prediction.

```
ssh root@[ip-addr] cd /opt/
```

```
scp -r /opt/output_cpu user@host-ip:<path>
```

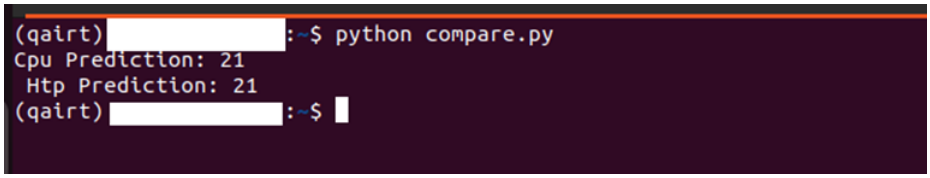
```
scp -r /opt/output_htp user@host-ip:<path>
```

Note: Use the <path> from the above commands in the following Python script.

Once the outputs from the qnn-net-run tool are copied from the device to the host computer, create a simple Python script to load outputs from CPU and HTP execution and compare them using NumPy.

```
#python postprocessing script (compare.py)

import numpy as np

htp_output_file_path = "<path>/output_htp/Result_1/875.raw"
cpu_output_file_path = "<path>/output_cpu/Result_1/875.raw"

htp_output = np.fromfile(htp_output_file_path, dtype=np.float32)
htp_output = htp_output.reshape(1,1000)

cpu_output = np.fromfile(cpu_output_file_path, dtype=np.float32)
cpu_output = cpu_output.reshape(1,1000)

cls_id_htp = np.argmax(htp_output)
cls_id_cpu = np.argmax(cpu_output)


# Let's compare CPU output vs HTP output
print("CPU prediction {} \n HTP prediction {}".format(cls_id_
cpu, cls_id_htp))
```

Output



Deployment using QNN APIs

The Qualcomm AI Engine Direct SDK provides C/C++ APIs to create/develop applications that can load a compiled .so model and run it on a chosen backend (CPU, GPU, or HTP) with acceleration. See the sample application application that demonstrates QNN C/C++ APIs.

Deployment using Qualcomm IM SDK

To improve the developer experience when building entire use case pipelines (stream from camera, preprocess images, perform inference, etc.), QNN has been integrated into the Qualcomm IM SDK as the qtimlqnn plugin.

The plugin has been developed on top of QNN C APIs and provides capabilities to dynamically load and run models. With the `qtimlqnn` plugin, developers can use their converted and compiled model library in the Qualcomm IM SDK pipeline to realize their use case.

See Develop your own AI/ML application for instructions on how to deploy QNN models using Qualcomm IM SDK.

4 Run AI/ML sample applications

The AI/ML reference applications showcase practical scenarios for running models on a live camera feed, file source, or real-time streaming protocol (RTSP) on Qualcomm evaluation kits. To try out these AI reference applications, you must first obtain a model from one of the mentioned model repositories, such as AI Hub or LiteRT model zoo. The subsequent steps are detailed below, including the steps to run the reference applications.

The Qualcomm IM SDK contains sample applications that enable you to experience the features of the Qualcomm evaluation kit. These applications serve as a reference for custom use cases on the Qualcomm evaluation kit. See Qualcomm IM SDK sample applications for the list of default reference apps.

You can run the default reference application or customize the sample application by integrating your own model using either a Qualcomm evaluation kit or Qualcomm Device Cloud

4.1 Run a sample application using EVK

To run a sample application using a Qualcomm evaluation kit, ensure that the eSDK is setup and that your environment meets the prerequisites.

Prerequisites

The prerequisites for running the reference applications are as follows:

- A Qualcomm evaluation kit: Connected to a monitor through HDMI or DisplayPort
- An SSH connection with the Qualcomm evaluation kit; establish an SSH connection with the device. Once connected, the Qualcomm evaluation kit is accessible through its configured IP address. The IP address is needed further to push models and artifacts needed to run AI/ML applications.
- Model downloaded from AI Hub and the corresponding label files.
- Ensure that the device is powered on and the Wayland display is visible on the monitor.

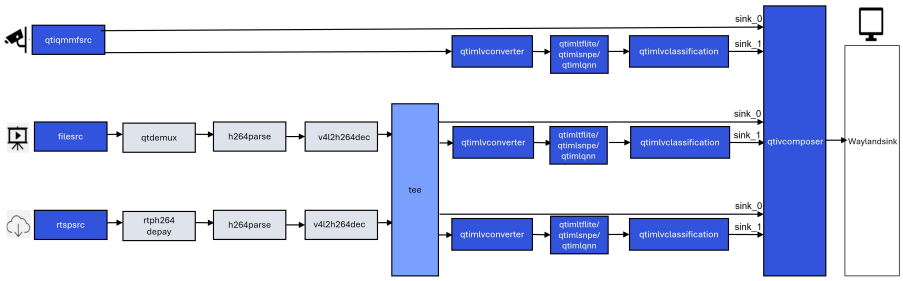
Setup eSDK for development

The eSDK (extensible SDK) needs to be setup to develop application/plugin code. See the Qualcomm Intelligent Multimedia Product SDK Quick Start Guide for instructions on setting up the eSDK and downloading and compiling the source code.

Classify objects using a default model

The *gst-ai-classification* sample application demonstrates the capability of hardware to perform classification on a video stream.

The pipeline receives the video stream from the camera, file source, or real-time streaming protocol (RTSP), performs preprocessing, conducts inference on AI hardware, and displays the results on the screen.



The *gst-ai-classification* application is part of the Qualcomm Intelligent Multimedia Product (QIMP) SDK and can be run directly after flashing the device. You need to push the model and label files to the device to run the application.

Download model and label files

To download the model and label files directly on the device, follow the steps below.

1. Enable Wi-Fi and SSH.

   The device needs an internet connection to download the artifacts required to run sample applications. If you have already enabled SSH and connected to the internet with Wi-Fi, skip this step.

2. Sign in to the target device using SSH:

   ```
   ssh root@<IP address of the target device>
   ```

3. Download the automated script used to download model and label files to the `/etc/models` and `/etc/labels` directories on the target device:

   ```
   curl -L -O https://raw.githubusercontent.com/quic/sample-apps-for-qualcomm-linux/refs/heads/main/download_artifacts.sh
   ```

Note: Run this script on the target device. Ensure that the target device has a valid Internet connection.

4. Set permissions for the script:

```
chmod +x download_artifacts.sh
```

5. Run the script with the required arguments to download model and label files to the target device:

```
./download_artifacts.sh -v GA1.4-rel -c <soc name>
```

   Replace `<soc name>` with QCS6490, QCS9075, or QCS8275.

6. The YOLOv8 and YOLO-NAS models aren't available by default.

   Download them using the provided script or export them with AI Hub APIs. If you are using the `gst-ai-multistream-batch-inference` application, generate a batch model.

Download them with a script

1. Create a Qualcomm AI Hub account.
2. Select the account name > go to **Settings** in the upper right corner > select the API key.
3. Run the following commands on the Linux host:

```
curl -L -O https://raw.githubusercontent.com/quic/sample-apps-for-qualcomm-linux/refs/heads/main/scripts/export_model.sh
```

```
chmod +x export_model.sh
```

   Replace `<API_KEY>` with the selected key:

```
./export_model.sh --api-key=<API_KEY>
```

   After executing the script, the models will be downloaded to the build directory.

4. Copy the models to the device in the `/etc/models/` directory.

```
scp <working-directory>/build/yolonas_quantized/yolonas_quantized.tflite root@<IP address of target device>:/etc/models/
```

```
scp <working-directory>/build/yolov8_det_quantized/yolov8_
det_quantized.tflite root@<IP address of target device>:/etc/
models/
```

Export them with AI Hub APIs

1.4 requires version 2.32 of the Qualcomm AI Runtime SDK. Use the following command to export the model. For example:

- To export the YoloV8 QNN model:

```
python -m qai_hub_models.models.yolov8_det.export --quantize
w8a8 --target-runtime=qnn --chipset="qualcomm-qcs6490-proxy"
--compile-options="--qairt_version 2.32" --profile-options "-
-qairt_version 2.32"
```

- To export the YoloV8 LiteRT model:

```
python -m qai_hub_models.models.yolov8_det.export --quantize
w8a8 --target-runtime=tflite --chipset="qualcomm-qcs6490-
proxy"
```

YOLOv8-Detection-Quantized

Yolo-NAS-Quantized

Generate a batch model

To change the batch size of the model, update <N> in the following export command:

```
python -m qai_hub_models.models.<Model_Name>.export --quantize
w8a8 --batch-size <N> --device "QCS6490 (Proxy)"
```

7. Update the q_offset and q_scale constants of the quantized LiteRT model in the JSON file. For instructions, see get model constants.
8. If a model isn't available after downloading with the script, download it from IoT-Qualcomm AI Hub and copy it to the target device by running the following command on the host computer:

```
scp <model filename> root@<IP addr of the target device>:/etc/
models
```

For example:

```
scp inception_v3_quantized.tflite root@<IP addr of the target
device>:/etc/models
```

Note: If you want to run sample applications from the UART shell, remount the file system with read/write permission using the following command on the target device:

```
mount -o remount,rw /usr
```

Run the sample application

Important: The rest of this document assumes the following convention for default file locations:

- `/etc/models/` for all model files
- `/etc/labels/` for all label files
- `/etc/media/` for all video files
- `/etc/configs/` for all configuration files

To run the reference application, run the following commands.

1. The `download_artifacts.sh` script downloads the sample `video.mp4` file to the `/etc/media` directory. If you wish to use your own video file, push the video file to the `/etc/media` directory using the command below and update the path in the configuration file.

   ```
   scp <video file> root@<IP address of target device>:/etc/media/
   ```

   Models for the LiteRT and Qualcomm AI Engine Direct runtimes are available from AI Hub.

   ```
   scp inception_v3_quantized.tflite root@<IP address of target
   device>:/etc/models/
   ```

   ```
   scp inception_v3_quantized.bin root@<IP address of target
   device>:/etc/models/
   ```

2. The sample application uses a JSON file to read input parameters. The following shows the format of the `/etc/configs/config_classification.json` file.

   ```
   {
   "file-path": "<input video path>",
   "ml-framework": "<snpe or LiteRT or qnn framework>",
   ```

```
"model": "<model path>",
"labels": "<label path>",
"threshold": <postprocesssing threshold, integer value from 1 to
100>,
"constants": "<Model constants for LiteRT/AI Engine Direct
runtime>",
"runtime": "<dsp, gpu, cpu runtime>"
}
```

The `ml-framework` field takes the following values:

- `snpe`: Use with Qualcomm Neural Processing Engine SDK models
- `tflite`: Use with LiteRT models
- `qnn`: Use with Qualcomm AI Engine Direct SDK models

The runtime field takes the following values:

- `cpu`: Run on CPU
- `gpu`: Run on GPU
- `dsp`: Run on HTP

The input source for the sample app can be selected by adding one of the following fields to the configuration file.

- `"camera"`:
  - `0`: Primary camera
  - `1`: Secondary camera

  Note: This is only applicable for QCS6490 devices.

- `"file-path"`: Path to video file
- `"rtsp-ip-port"`: Address of RTSP stream in the following format:

```
rtsp://<ip>:<port>/<stream>
```

Warning: Add only one input source in the configuration file.

3. Enable the Weston service for the display by running the following command:

```
export XDG_RUNTIME_DIR=/dev/socket/weston && export WAYLAND_
DISPLAY=wayland-1
```

4. Once the JSON file has been populated, run the sample app using the following command.

```
gst-ai-classification --config-file=/etc/configs/config_
classification.json
```

For more information about the fields in the configuration file or how to run the application use the help (`-h`) command.

```
gst-ai-classification -h
```

For example, the following configuration file executes the LiteRT model on video input using the DSP runtime.

```
{
"file-path": "/etc/media/video.mp4",
"ml-framework": "tflite",
"model": "/etc/models/inception_v3_quantized.tflite",
"labels": "/etc/labels/classification.labels",
"constants": "Mobilenet,q-offsets=<38.0>,q-scales=<0.
17039915919303894>;",
"threshold": 40,
"runtime": "dsp"
}
```

Notes

- To stop the use case, press `CTRL+C`.
- To display the available help options, run the following command:

```
gst-ai-classification -h
```

- The GStreamer debug output is controlled by the GST_DEBUG environment variable. Set the required level to enable logging. For example, to log all warnings, run the following command.

```
export GST_DEBUG=2
```

For more troubleshooting options, see Troubleshooting and FAQ.

Run a sample application with a custom trained model

It uses a custom-trained YoloV8 model as an example.

Prerequisites

1. Set up the extensible SDK (eSDK) to develop application/plugin code.

   See the Qualcomm Intelligent Multimedia Software Development Kit (IM SDK) Quick Start Guide for instructions on setting up the eSDK and downloading and compiling the source code.

2. See Develop your own AI/ML application for instructions on downloading the source code and compiling custom reference apps.

3. Download model and label files for Qualcomm Neural Processing SDK.

Use a custom trained YoloV8 LiteRT model

Qualcomm IM SDK reference applications use the YoloV8 model for object detection. This example explains how you can try a custom trained YoloV8 variant.

Use the following steps to run your own custom-trained YoloV8 model with the current reference application.

1. Replace the existing model with your new model in the reference app.
2. Modify the label files with custom labels.
3. Run the reference application with the modified model.

Modify the labels

When using Qualcomm IM SDK and its reference apps, the apps expect labels in a specific format. You need to update `<label_name>`, `<hex_value_for_label_id>`, and `<hex_value_for_color>` values for each label within this labels file.

The format of each label within the labels file should follow the template shown below.

```
(structure)"<label_name>,id=(guint)<hex_value_for_label_id>,
color=(guint)<hex_value_for_color>;"
```

For example:

```
(structure)"person,id=(guint)0x0,color=(guint)0x00FF00FF;"
(structure)"bicycle,id=(guint)0x1,color=(guint)0x00FF00FF;"
(structure)"car,id=(guint)0x2,color=(guint)0x0000FFFF;"
(structure)"motorcycle,id=(guint)0x3,color=(guint)0x00FF00FF;"
```

Run object detection using the custom model

To run object detection models using the LiteRT runtime, specify the custom trained model along with your custom label files in the command-line parameters and run the following commands.

1. Copy the model to the device.

```
scp yolov8_custom.tflite root@<IP address of the target device>:
/etc/models/
```

2. Copy the labels to the device.

```
scp yolov8_custom.labels root@<IP address of the target device>:
/etc/labels/
```

3. Sign in to the device.

```
ssh root@<IP address of the target device>
```

4. Modify the `/etc/configs/config_detection.json` file as shown.

```
{
"file-path": "/etc/media/video.mp4",
"ml-framework": "tflite",
"yolo-model-type": "yolov8",
"model": "/etc/models/yolov8_custom.tflite",
"labels": "/etc/labels/yolov8_custom.labels",
"constants": "<constants for custom YOLOv8 model>",
"threshold": 40,
"runtime": "dsp"
}
```

5. Enable the Weston service by running the following command:

```
export XDG_RUNTIME_DIR=/dev/socket/weston && export WAYLAND_
DISPLAY=wayland-1
```

6. Run the sample application.

```
gst-ai-object-detection --config-file=/etc/configs/config_
detection.json
```

Notes

- To display the available help options, run the following command:

```
gst-ai-object-detection -h
```

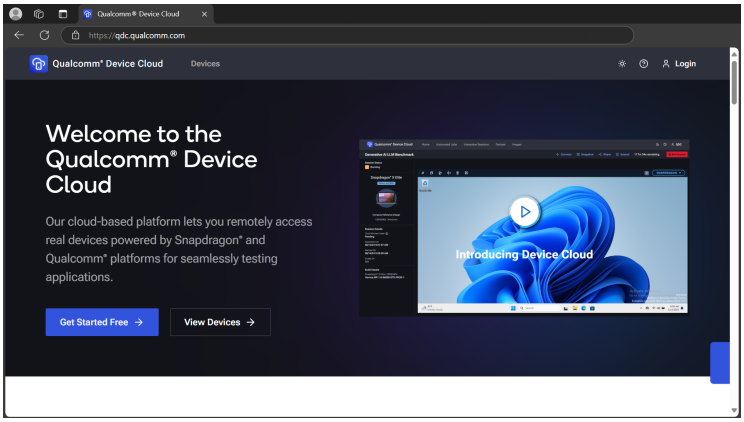
- To stop the use case, use `CTRL+C`.

4.2 Run a sample application using Qualcomm Device Cloud

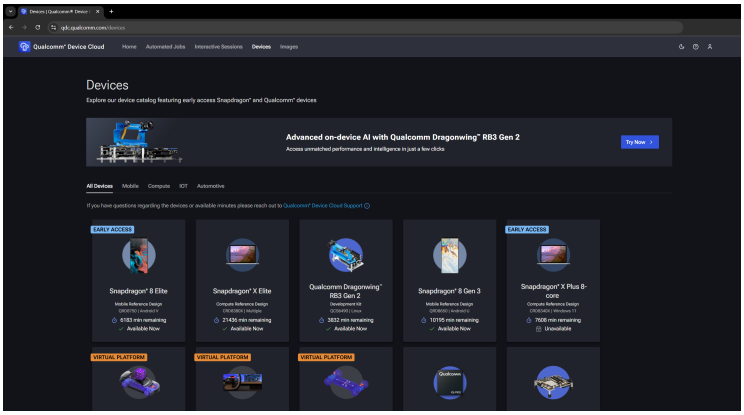
Qualcomm Device Cloud is a cloud-based platform that allows developers to remotely access devices powered by Snapdragon and Qualcomm platforms for seamlessly testing applications. Users can remotely access Qualcomm IoT platforms such as QCS6490 to get a feel for the device and experience some of its capabilities.

Start a device session

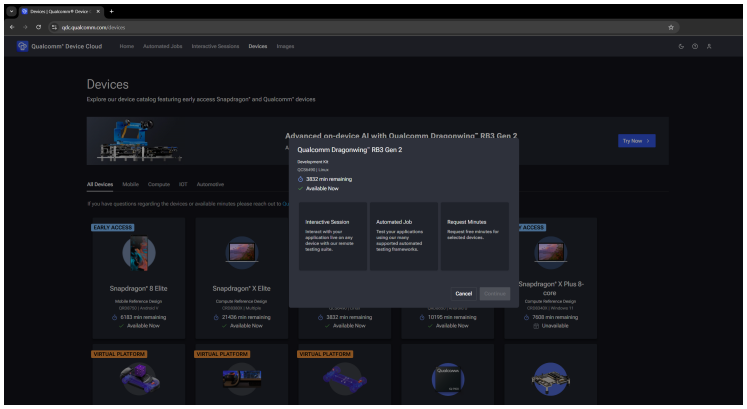
1. Head to Qualcomm Device Cloud and sign in using your Qualcomm ID.



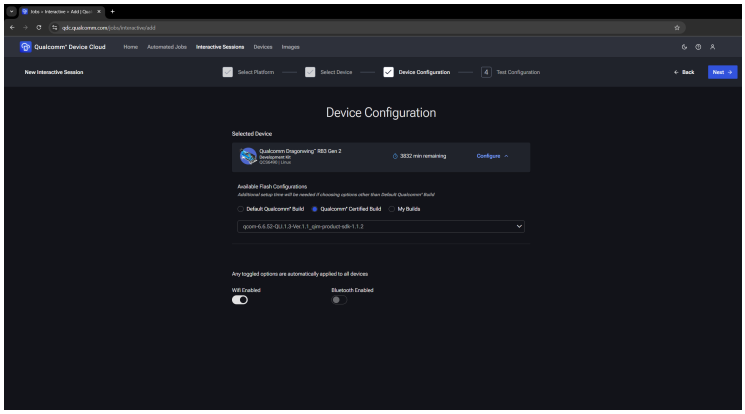
2. Select **Devices** and select **Qualcomm Dragonwing™ RB3 Gen 2**.



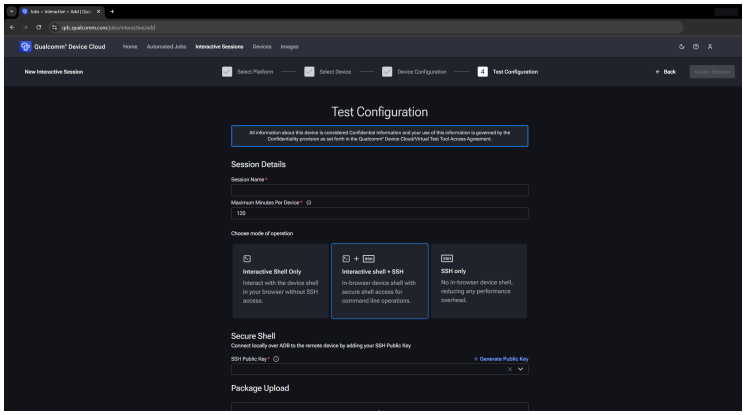
3. Select **Interactive session** and press **Continue**.



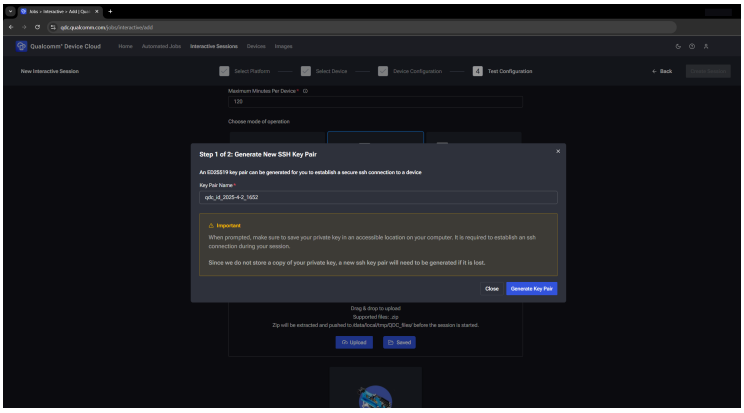
4. On the Device Configuration page, you can choose from Qualcomm Certified Builds or custom builds. You can also enable Wi-Fi and Bluetooth. Ensure that Wi-Fi is enabled before proceeding to the next step.



5. On the Test Configuration page, enter the **Session Name** and **Maximum Minutes Per Device**. Select **Interactive shell + SSH**, and generate a new Public Key for the session.



6. Select **Generate New Pair**.



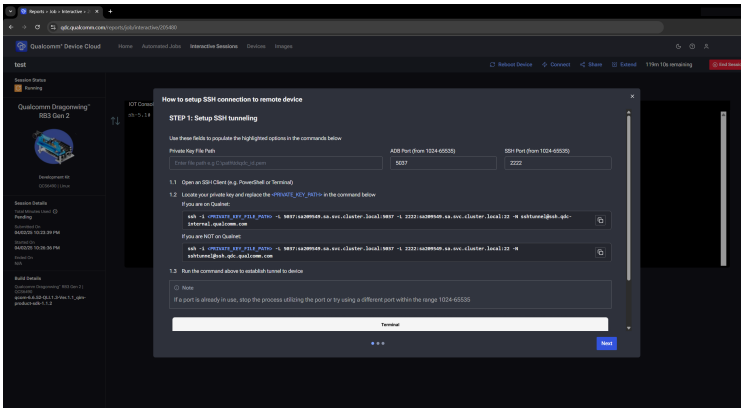
A `.pem` file is downloaded on your host computer.

7. Select **Create Session**.

   A new device session will be created in 10-15 minutes.

Connect using SSH

1. Once the session is created, select **Connect** on the top bar.
2. In the **Private Key File Path** textbox, enter the path to the `.pem` file downloaded during the *Generate new SSH key Pair* step.



Note: The command provided here doesn't create an SSH shell or connect you to the QDC device via SSH. It starts port forwarding to allow SSH access.

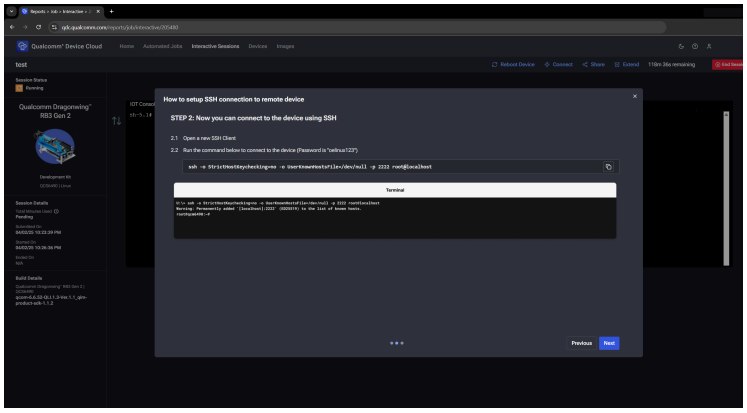
Run the command provided in this step in your local terminal and keep it running to allow port forwarding.

To obtain the SSH command, proceed to the next steps.

---

3. Press **Next** to obtain commands to connect to SSH shell

4. Copy the commands provided and run them on your host computer.

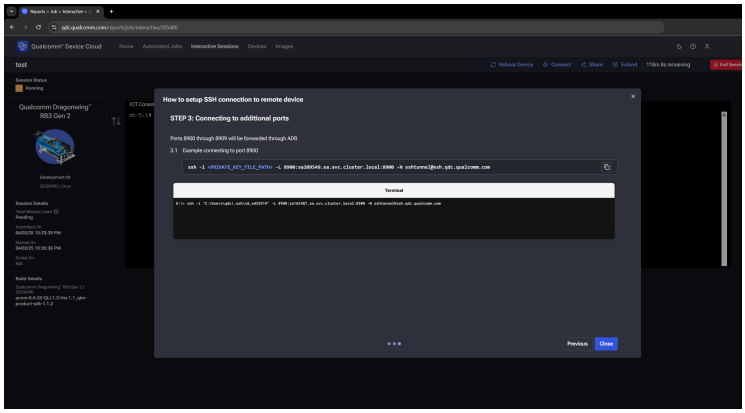


5. Copy the command and run it in the terminal to connect using SSH.

Note: If prompted for a password, enter the `oelinux123`

---

6. Press **Next** to obtain commands to enable port forwarding.



The above command forwards port 8900. If you wish to forward more ports between your host and the QDC device, edit the following command as follows:

```
ssh -i -L 8900:sa203658.sa.svc.cluster.local:8900 -L 8901:
sa203658.sa.svc.cluster.local:8901 ... -N sshtunnel@ssh.qdc.
qualcomm.com
```

Note: The above command is an example only.

The command provided in your device session uses a unique session ID:
`8900:sa<unique_session_id>.sa.svc.cluster.local:8900`

Edit the command that you get from your own device session as illustrated in the above

command.

Setup RTSP

1. Download the MediaMTX utility to setup an RTSP listening server.

   The following steps are verified for MediaMTX version v1.11.3_linux_arm64v8, which can be downloaded from mediamtx_v1.11.3_linux_arm64v8.tar.gz.

2. Unzip the folder and edit the `mediamtx.yml file` to allow RTSP streaming at port `8901`.

```
############################################

# Global settings -> RTSP server

# Enable publishing and reading streams with the RTSP protocol.

rtsp: yes

# List of enabled RTSP transport protocols.
# UDP is the most performant, but doesn't work when there's a
NAT/firewall between
# server and clients, and doesn't support encryption.
# UDP-multicast allows to save bandwidth when clients are all in
the same LAN.
# TCP is the most versatile, and does support encryption.
# The handshake is always performed with TCP.
rtspTransports: [udp, multicast, tcp]

# Encrypt handshakes and TCP streams with TLS (RTSPS).
# Available values are "no", "strict", "optional".
rtspEncryption: "no"

# Address of the TCP/RTSP listener. This is needed only when
encryption is "no" or "optional".
rtspAddress: :8901

# Address of the TCP/TLS/RTSPS listener. This is needed only
when encryption is "strict" or "optional".
rtspsAddress: :8322
```

3. Copy the `mediamtx.yml` file and mediamtx executable to the device:

```
scp -P 2222 -o StrictHostKeychecking=no -o UserKnownHostsFile=/
dev/null <mediamtx path>/mediamtx root@localhost:/opt
```

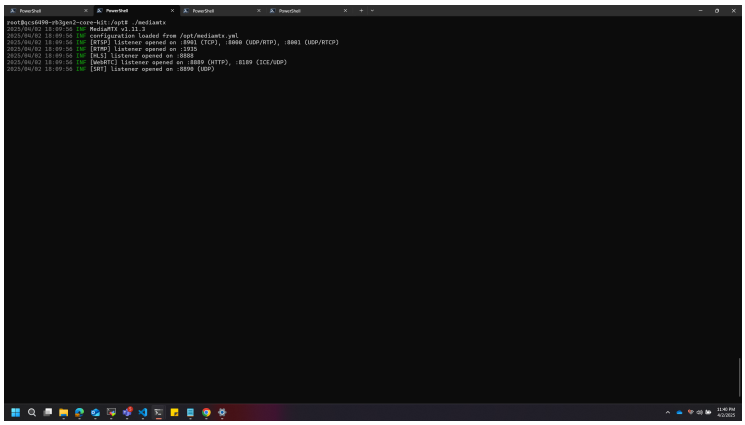
```
scp -P 2222 -o StrictHostKeychecking=no -o UserKnownHostsFile=/
dev/null <mediamtx path>/mediamtx.yml root@localhost:/opt
```

4. Start a new SSH shell and run the `mediamtx` utility

```
cd /opt
```

```
chmod +x mediamtx
```

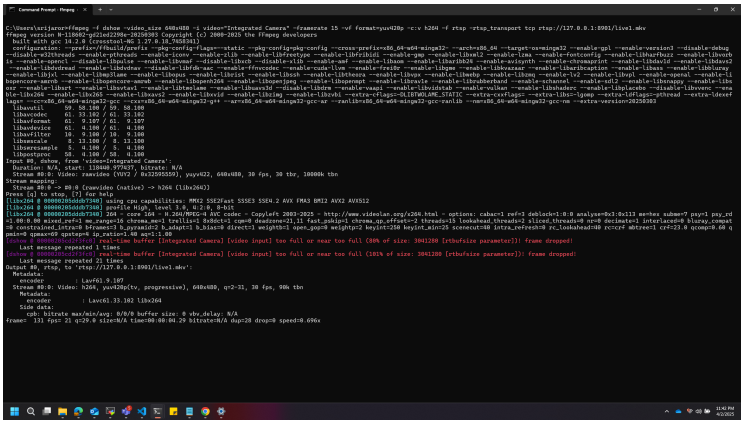
```
./mediamtx
```



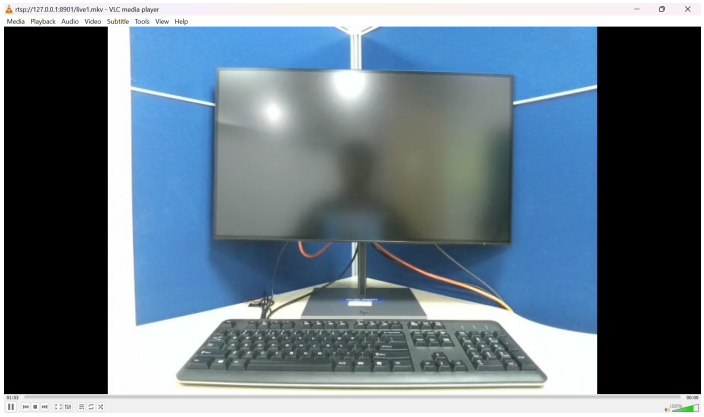
5. Start an RTSP stream on your host computer on localhost and port number `8901`.

   For example, the following command uses FFmpeg to take input from connected webcam and send it over an RTSP stream. If you don't have FFmpeg, you can download it from the FFmpeg website.

```
ffmpeg -f dshow -video_size 640x480 -i video="Integrated Camera"
-framerate 15 -vf format=yuv420p -c:v h264 -f rtsp -rtsp_
transport tcp rtsp://127.0.0.1:8901/live1.mkv
```



6. View the output of this RTSP stream, using VLC media player.
    a. Open VLC media player.
    b. Press `Control+N`.
    c. Enter the URL `rtsp://127.0.0.1:8901/live1.mkv` in the input bar.
    d. Press **Play**.



Run sample apps

The Qualcomm Dragonwing™ RB3 Gen 2 device comes with pre-installed GStreamer based AI sample applications. Some of these sample applications support taking input using RTSP (such as IP cameras) and output through RTSP stream which can be viewed on the host computer.

AI Hub workflow using multistream inference sample application

These instructions guide developers in replacing models from AI hub in sample apps and running them on QDC (Qualcomm Device Cloud).

The **gst-ai-multistream-inference** application demonstrates AI inference (object detection and classification) on up to 8 combined streams coming from a camera, file, or RTSP stream.

The output is displayed on an HDMI display, saved as an H.264 encoded MP4 file, or converted into an RTSP stream.

1. Download YOLOv8 and quantized InceptionV3 models.
2. Obtain constants for the downloaded models.
3. Copy the models and input video files to the device by running the following commands on your host computer terminal.

```
scp -P 2222 -o StrictHostKeychecking=no -o UserKnownHostsFile=/
dev/null <model file path> root@localhost:/etc/models/
```

```
scp -P 2222 -o StrictHostKeychecking=no -o UserKnownHostsFile=/
dev/null <video file path> root@localhost:/etc/media/
```

4. Download the labels file on the device by running the following commands on your QDC device shell:

```
wget https://github.com/quic/sample-apps-for-qualcomm-linux/
releases/download/GA1.4-rel/labels.zip
```

```
unzip -d /etc/ labels.zip
```

5. Create the `/etc/xdg/weston` directory on the device.

```
mkdir /etc/xdg/weston
```

6. Edit the `/etc/xdg/weston/weston.ini` file on the device as shown below:

```
# configuration file for Weston
[core]
idle-time=0
# backend=sdm-backend.so
repaint-window=10
gbm-format=xbgr8888
require-outputs=none

[output]
```

```
name=DSI-1
mode=on

[output]
name=DP-1
mode=on

[shell]
clock-format=seconds
```

7. In a new SSH shell, run the following command to enable Weston.

```
export GBM_BACKEND=msm && export XDG_RUNTIME_DIR=/dev/socket/
weston && mkdir -p $XDG_RUNTIME_DIR && weston --continue-
without-input --idle-time=0
```

8. Edit the `/etc/configs/config-multistream-inference.json` as shown below:

```
{
"input-rtsp-path": [
   "rtsp://127.0.0.1:8901/live1.mkv"
],
"model": "/etc/models/yolov8_det_quantized.tflite",
"labels": "/etc/labels/yolov8.labels",
"constants" : "<constants for LiteRT model>",
"output-display": 0,
"output-ip-address": "127.0.0.1",
"output-port-number": "8900",
"use-case": 0
}
```

Here, the `input-rtsp-path` field is a JSON array of input RTSP addresses from which the sample app takes input. The output is transmitted as an RTSP stream at `rtsp://<output-ip-address>:<output-port-number>/live`.

To demonstrate file input and file output, edit the `/etc/configs/config-multistream-inference.json` as shown below:

```
{
"input-file-path": [
   "/etc/media/video1.mp4",
   "/etc/media/video2.mp4",
   "/etc/media/video3.mp4",
   "/etc/media/video4.mp4",
   "/etc/media/video5.mp4",
```

```
    "/etc/media/video6.mp4"
],
"model": "/etc/models/yolov8_det_quantized.tflite",
"labels": "/etc/labels/yolov8.labels",
"input-type": "h264",
"output-display": 0,
"output-file-path": "<output file path>",
"use-case": 0
}
```

The `input-file-path` field is a JSON array of input files. The output of the sample app is stored in the file path specified in the `output-file-path` field

9. Run the sample app:

```
export XDG_RUNTIME_DIR=/dev/socket/weston && export WAYLAND_
DISPLAY=wayland-1
```

```
gst-ai-multistream-inference
```

For more information about the sample app and config file fields, run the following command on the device:

```
gst-ai-multistream-inference -h
```

10. View or copy the output.

File output

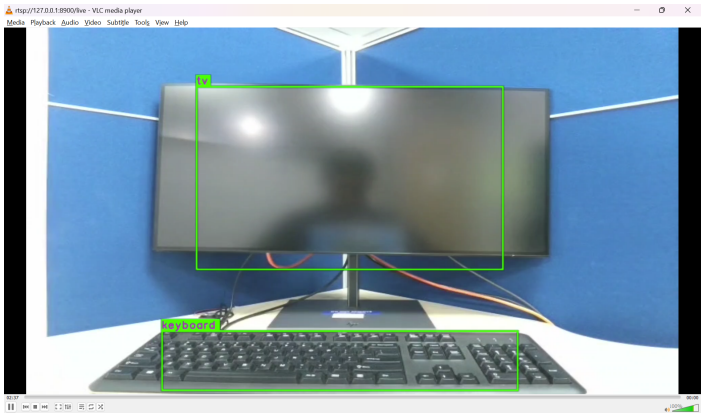
The output file can be copied to your host machine by executing the following command in your host terminal:

```
scp -P 2222 -o StrictHostKeychecking=no -o UserKnownHostsFile=/
dev/null root@localhost:<output file path> ./
```

RTSP output

View the output using VLC media player.

1. Open VLC media player.
2. Press `Control+N`.
3. Enter the URL `rtsp://127.0.0.1:8900/live` in the input bar.
4. Press **Play**



For more information, see multistream inference

Run gst-ai-multi-input-output-object-detection sample app on QDC

The `gst-ai-multi-input-output-object-detection` application allows you to perform objection detection on video streams from various sources such as a camera, a file, or over a network such as RTSP.

The output is displayed on an HDMI display, saved as an H.264 encoded MP4 file, or converted into an RTSP stream.

1. Generate the yolov5.tflite model and download the label files
2. Obtain constants for the downloaded models.
3. Copy the model and input video files to device using the following commands:

```
scp -P 2222 -o StrictHostKeychecking=no -o UserKnownHostsFile=/
dev/null <model file path> root@localhost:/etc/models/
```

```
scp -P 2222 -o StrictHostKeychecking=no -o UserKnownHostsFile=/
dev/null <video file path> root@localhost:/etc/media/
```

4. Download the labels file on the device by running the following commands:

```
wget https://github.com/quic/sample-apps-for-qualcomm-linux/
releases/download/GA1.4-rel/labels.zip
```

```
unzip -d /etc/ labels.zip
```

5. Unless already done, create the `/etc/xdg/weston` directory on the device.

```
mkdir /etc/xdg/weston
```

6. Unless already done, edit the `/etc/xdg/weston/weston.ini` file on the device as shown below:

```
# configuration file for Weston
[core]
idle-time=0
# backend=sdm-backend.so
repaint-window=10
gbm-format=xbgr8888
require-outputs=none

[output]
name=DSI-1
mode=on

[output]
name=DP-1
mode=on

[shell]
clock-format=seconds
```

7. Unless already done, in a new SSH shell, run the following command to enable Weston.

```
export GBM_BACKEND=msm && export XDG_RUNTIME_DIR=/dev/socket/
weston && mkdir -p $XDG_RUNTIME_DIR && weston --continue-
without-input --idle-time=0
```

8. Run the following command:

```
export XDG_RUNTIME_DIR=/dev/socket/weston && export WAYLAND_
DISPLAY=wayland-1
```

9. Run the sample application:

   - Run object detection on streams from file source and store the output in a file:

```
gst-ai-multi-input-output-object-detection --num-file=2 -f /
etc/media/output.mp4 --model=/etc/models/yolov5.tflite --
labels=/etc/labels/yolov5.labels
```

- Run object detection on streams coming from RTSP and output over RTSP stream:

```
gst-ai-multi-input-output-object-detection --num-rtsp=1 --
rtsp-ip-port=127.0.0.1:8901 --out-rtsp --model=/etc/models/
yolov5.tflite --labels=/etc/labels/yolov5.labels -i 127.0.0.1
-p 8900
```

10. View or copy the output.

File output

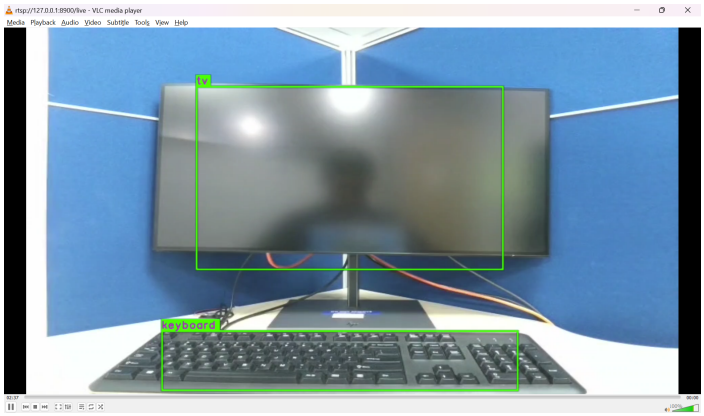
The output file can be copied to your host machine by running the following command in your host terminal:

```
scp -P 2222 -o StrictHostKeychecking=no -o UserKnownHostsFile=/
dev/null root@localhost:<output file path> ./
```

RTSP output

View the output using VLC media player.

1. Open VLC media player.
2. Press `Control+N`.
3. Enter the URL `rtsp://127.0.0.1:8900/live` in the input bar.
4. Press **Play**.



For more information, see multi-input inferencing

5 Develop your own AI/ML application

You can write AI/ML applications using one of the following methods.

 Develop your own AI/ML application with the Qualcomm Visual Studio Code Extension

 Develop your own AI/ML application with the Qualcomm IM SDK

 Develop your own application with the Qualcomm AI Runtime SDK APIs

5.1 Develop your own AI/ML application with the Qualcomm Visual Studio Code Extension

The recommended approach for creating applications is to use the Qualcomm Visual Studio Code Extension, a tool that integrates application development for QLI platforms.

Before you begin, ensure that both the Host and Target are configured for the Qualcomm Visual Studio Code Extension.

Follow the steps provided in Projects to configure and create applications using the Qualcomm Visual Studio Code Extension.

5.2 Develop your own AI/ML application with the Qualcomm IM SDK

You can leverage existing reference applications to deploy your models or extend capabilities of the Qualcomm IM SDK to add support for your own models and use cases.

Qualcomm IM SDK, is a unified SDK enabling seamless multimedia and AI/ML application deployment. The SDK uses GStreamer, an open-source multimedia framework, and exposes APIs

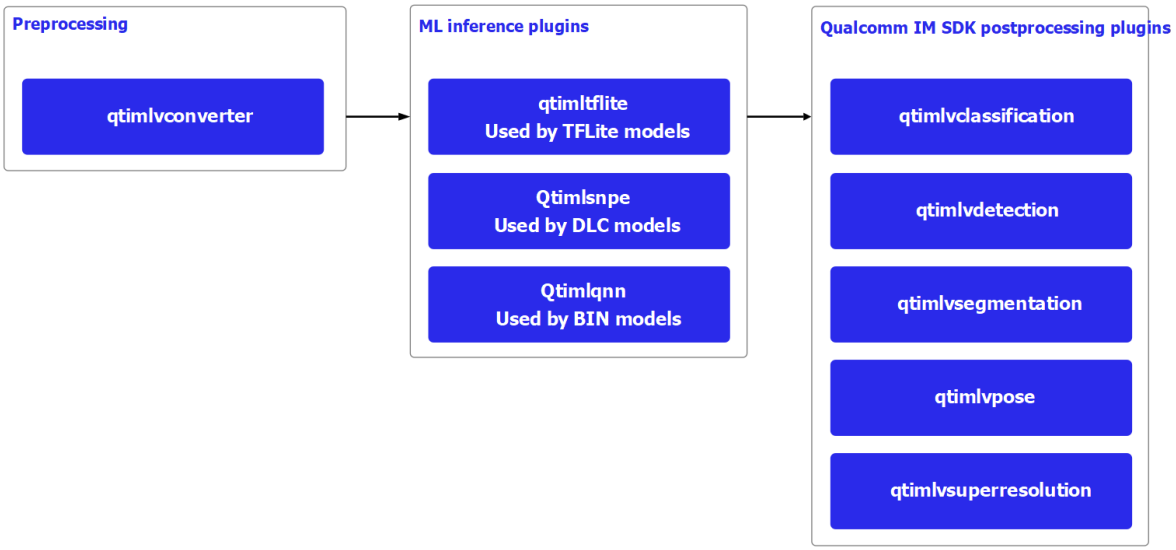
and plugins in both multimedia and ML domains.

For details, see the official SDK documentation.

The Qualcomm IM SDK implements the following plugins for AI/ML applications. A complete list of plugins can be found in the Qualcomm IM SDK plugin documentation.

Download source code for development

The eSDK (extensible SDK) needs to be setup to develop application/plugin code. See the Qualcomm Intelligent Multimedia Software Development Kit (IM SDK) Quick Start Guide for instructions on setting up the eSDK and downloading and compiling the source code.



The following pre-/postprocessing ML plugins are available with the Qualcomm IM SDK. Use these plugins to develop your own use case.

Plugin	Functionality
qtimlvconverter	Transforms incoming video buffers into neural-network tensors while performing required format conversion and resizing.
qtimlvclassification	Performs postprocessing of output tensors for classification use cases.
qtimlvdetection	Performs postprocessing of output tensors for detection use cases.
qtimlvsegmentation	Performs postprocessing of output tensors for pixel-level use cases, like image segmentation, depth-map, etc.
qtimlvpose	Performs postprocessing of output tensors for pose estimation use cases.
qtimlvsuperresolution	Performs postprocessing of the output tensors for video super resolution use cases.

The Qualcomm IM SDK, supports the following use cases and related models.

Use cases supported by Qualcomm IM SDK	**Supported Models**
Classification	Models like Mobilenet. Currently Qualcomm AI Hub has 11 classification models supported. New models will keep getting added to AI Hub.
Detection	Models like ssd-mobilenet, yolov5, yolo-nas, and yolov8
Segmentation	Models like deeplabv3_resnet and ffnet
Pose detection	Models like posenet_mobilenet
Super resolution	Models like QuickSRNet, XLSR, etc.

Note: A list of verified models from Qualcomm AI Hub is available in a Integrate an AI Hub model in an application.

You can use many other models with similar postprocessing requirements, however it's recommended to verify postprocessing support in the relevant Qualcomm IM SDK plugins before integrating your own model.

Integrate a custom AI model in an application

This section describes the paths available for you to deploy your custom model with Qualcomm SDKs.

To use a custom model in an AI application, you can use the Qualcomm Intelligent Multimedia SDK or Qualcomm AI Runtime SDK.

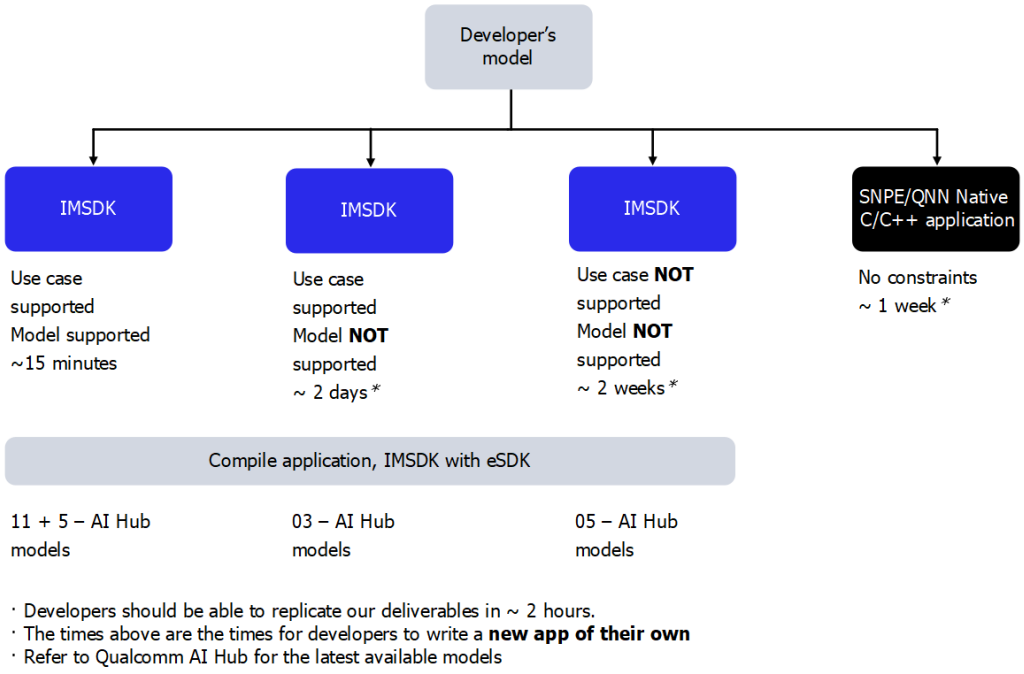


Figure1 Bring your own model: Time taken to deploy a model

SDK supported use case?	SDK supported model?	Number of AI Hub models available	Expected time to integrate the model in an application
Yes	Yes	16	15 minutes
Yes	No	3	2 days
No	No	5	~2 weeks This is an advanced use case.

SDK supported use case?	**SDK supported model?**	**Number of AI Hub models available**	**Expected time to integrate the model in an application**
No constraints	No constraints	0	~1 week This is an advanced use case.

Note: The moderate and advanced paths assume understanding of GStreamer fundamentals, understanding of GStreamer pipelines, expertise in C/C++, and a Yocto build environment.

Integrate an AI Hub model in an application

This section describes how to use an AI Hub model in a reference application. This example uses an image classification model from AI Hub.

Note: Preprocessing and postprocessing for most of the publicly available image classification models is similar and is likely supported by the `qtimlvclassification` plugin.

Qualcomm AI Hub provides optimized AI models for Qualcomm devices, allowing them to operate on CPU, GPU, or NPU using either LiteRT (previously Tensorflow Lite) or Qualcomm® AI Engine Direct.

This example explains how to use a mobilenet-v2-quantized model from AI Hub and integrate the model into the `gst-ai-classification` reference application.

AI Hub models verified with IMSDK

S.No	Reference Application	Verified AI Hub Models	Label file
1	gst-ai-classification	googlenet_quantized	Labels
2		inception_v3_quantized	
3		mobilenet-v2-quantized	
4		MobileNet-v3-Large-Quantized	
5		ResNet101Quantized	
6		SqueezeNet-1_1Quantized	
7		ResNet18Quantized	
8		resnext50_quantized	
9		wideresnet50_quantized	
10		shufflenet_v2_quantized	
11		resnext101_quantized	
12	gst-ai-segmentation	deeplabv3_plus_mobilenet_quantized	Labels
13		fcn_resnet50_quantized	
14		ffnet_40s_quantized	
15		ffnet_54s_quantized	
16		ffnet_78s_quantized	
17	gst-ai-object-detection	Yolo-NAS-Quantized	Labels
18		Yolo-v7-Quantized	
19		YOLOv8-Detection-Quantized	
20	gst-ai-superresolution	QuickSRNetLarge-Quantized	Not applicable
21		QuickSRNetMedium-Quantized	
22		QuickSRNetSmall-Quantized	
23		XLSR-Quantized	

Note: Some models only have LiteRT or Qualcomm AI Engine Direct SDK models available with AI Hub. However, AI Hub frequently updates its library to include more models and enhance

existing ones.

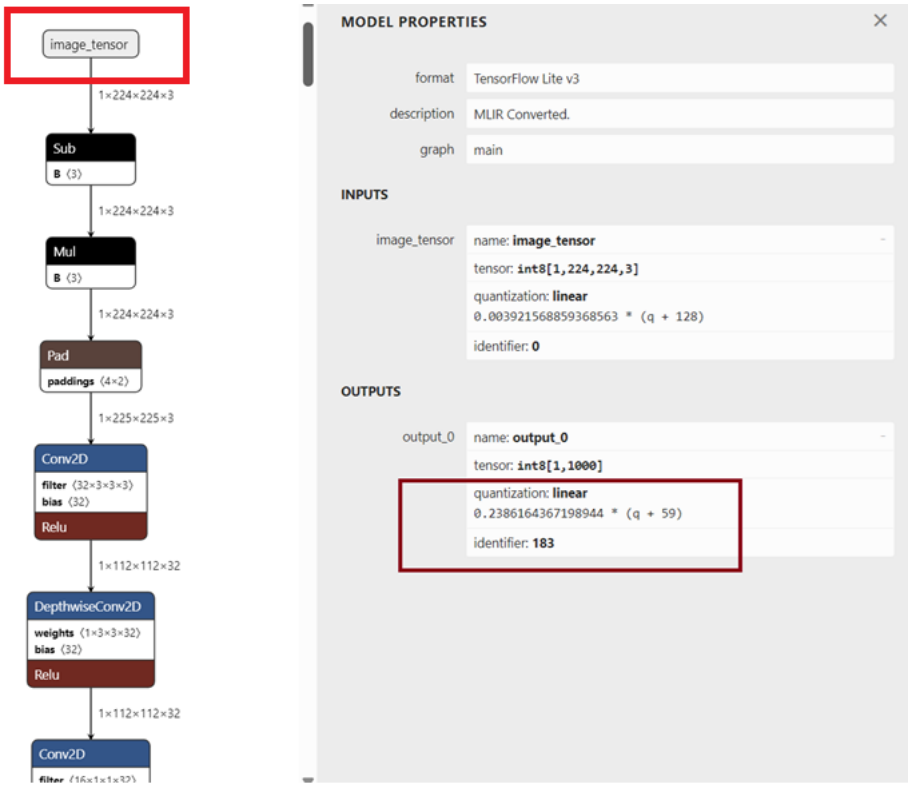
Prerequisites

1. Download AI Hub models.
2. Select the chipset for which you want to download the model.
   - For RB3 Gen 2 select *Qualcomm QCS6490*
   - For QCS9075 select *Qualcomm QCS9075*
   - For QCS8275 select *Qualcomm QCS8275*
3. Copy the label file that matches the model to the device.

   Label files are available for download at the provided links.

Get the model constants

1. Open the downloaded LiteRT model with a graph viewer tool like Netron.
2. Check postprocessing requirements for the specific model, you are going to include in the reference application.
   a. Select the input node of the model to see model properties.
   b. Copy the values from output node of the model.



   For example, in the above figure, `q_scale` is the value mentioned in quantization: linear, and `q_offset` is `-59` at output. You need to change the sign of `q_offset` from the model graph, this is needed based on the implementation of the plugin.

Note these values and use them when modifying the configuration.

Modify the configuration

1. Copy the model to the device.

```
scp mobilenet_v2_quantized.tflite root@<IP address of the target
device>:/etc/models/
```

2. Copy the labels to the device.

```
wget https://raw.githubusercontent.com/quic/ai-hub-models/refs/
heads/main/qai_hub_models/labels/imagenet_labels.txt
```

```
scp imagenet_labels.txt root@<IP addr of the target device>:/
etc/labels/
```

3. Update the `/etc/configs/config_classification.json` file as shown below.

```
{
"file-path": "/etc/media/video.mp4",
"ml-framework": "tflite",
"model": "/etc/models/mobilenet_v2_quantized.tflite",
"labels": "/etc/labels/imagenet_labels.txt",
"constants": "Mobilenet,q-offsets=<63.0>,q-scales=<0.
16931529343128204>;",
"threshold": 40,
"runtime": "dsp"
}
```

Note: Update `q-offset` and `q_scale` with the values noted in get the model constants.

Run the sample application

1. Enable the Weston service for the display by running the following command:

```
export XDG_RUNTIME_DIR=/dev/socket/weston && export WAYLAND_
DISPLAY=wayland-1
```

2. Run the reference application by giving the constants values in the command line parameters:

```
gst-ai-classification --config-file=/etc/configs/config_
classification.json
```

3. Use sample applications to run various AI Hub models on your devices. To run different AI Hub models:

   a. Download the model and label files that you want to run.

   b. Copy the model and label files to the device.

   c. Inspect the model file with the Netron application, fill in the constant parameters accordingly.

   d. Follow the steps in download models files for the YOLOv8 Quantized LiteRT model file (used for object detection).

   e. Update the `/etc/configs/config_detection.json` file as shown below.

```
{
"file-path": "/etc/media/video.mp4",
"ml-framework": "tflite",
"yolo-model-type": "yolov8",
"model": "/etc/models/YOLOv8-Detection-Quantized.tflite",
"labels": "/etc/labels/yolonas.labels",
"constants": "YOLOv8,q-offsets=<21.0, 0.0, 0.0>,q-scales=<3.
093529462814331, 0.00390625, 1.0>;",
"threshold": 40,
"runtime": "dsp"
}
```

   Note: Update the model name and model constants in the configuration file according to the model you obtained.

   f. Run the sample app.

   Note: Ensure that you have copied the model and label files to the device.

```
gst-ai-object-detection --config-file=/etc/configs/config_
detection.json
```

   g. Update the `/etc/configs/config_detection.json` file as follows:

   Take the `deeplabv3_plus_mobilenet_quantized.tflite` model file (used for segmentation) and the `deeplabv3_resnet50.labels` file as an example.

```
{
"file-path": "/etc/media/video.mp4",
"ml-framework": "tflite",
```

```
"model": "/etc/models/deeplabv3_plus_mobilenet_quantized.
tflite",
"labels": "/etc/labels/deeplabv3_resnet50.labels",
"constants": "deeplab,q-offsets=<0.0>,q-scales=<1.0>;",
"runtime": "dsp"
}
```

h. Run the sample app.

```
gst-ai-segmentation --config-file=/etc/configs/config_
segmentation.json
```

i. Take the `quicksrnetsmall_quantized.tflite` model file (used for superresolution) as an example to run:

```
gst-ai-superresolution --input-file=/etc/media/video.mp4 --
model=/etc/models/quicksrnetsmall_quantized.tflite --
constants="srnet,q-offsets=<0.0>,q-scales=<1.0>;"
```

Note: Ensure that the video used has a 128x128 resolution, which is the input to the model.

---

j. For additional details regarding the input parameters, see the help section of the application.

Note: In cases of segmentation, objects that aren't recognized may appear against a white background. This issue stems from the label file and will be addressed in future updates.

---

4. Recompile the reference application if you want to make any changes and run it. See the Qualcomm Intelligent Multimedia Software Development Kit (IM SDK) Quick Start Guide for instructions on how to compile reference applications with the eSDK.

Add postprocessing support for a custom model

This section guides you in adding model postprocessing support in a Qualcomm IM SDK plugin, particularly in cases where the current Qualcomm IM SDK plugin doesn't support postprocessing for the model.

This example explains the steps to add custom model postprocessing to the `qtimlvdetection` plugin.

Prerequisites

`qtimlvconverter` must support preprocessing for the model. See the README for more information.

Add YoloV8 module support to `qtimlvdetection`

1. Write a new module in the `qtimlvdetection` plugin to enable postprocessing support.

   See the following example code as a reference:
   gst-plugin-mlvdetection/modules/ml-vdetection-yolov8.c imsdk.lnx.2.0.0.r2-rel · CodeLinaro / le / platform / vendor / qcom-opensource / gst-plugins-qti-oss GitLab.

2. Write a new file for your custom model. In the model postprocessing module, the `gst_ml_module_process` function is responsible for executing all postprocessing operations.

   Recompile the plugin, using the instructions provided in compile and install Qualcomm IM SDK plugins.

3. Add your new files in `Cmakelists.txt`.

   See the following example as a reference:

   gst-plugin-mlvdetection/modules/CMakeLists.txt · imsdk.lnx.2.0.0.r2-rel · CodeLinaro / le / platform / vendor / qcom-opensource / gst-plugins-qti-oss GitLab

4. Recompile the reference application and run it.

   See the Qualcomm Intelligent Multimedia Software Development Kit (IM SDK) Quick Start Guide for instructions on how to compile reference applications with the eSDK.

   See the README, for more detailed instructions on how to write your own plugins.

Develop a custom GStreamer plugin

Before attempting to write your own GStreamer plugin, you should familiarize yourself with the core concepts and tutorials.

Figure2 High-level workflow for implementing a new GStreamer plugin

To implement a custom plugin, you will need to:

- Decide whether to inherit from an existing base template class or directly from the GstElement skeleton class.
- Specify pads, elements, and caps (capabilities).
- Implement key functions like init, set_caps, set_property, transform, etc.
- Register the plugin.

Key tutorials, guides, and resources

- Tutorials:

  https://gstreamer.freedesktop.org/documentation/tutorials/index.html?gi-language=c

- Plugin writer's guide:

  https://gstreamer.freedesktop.org/documentation/plugin-development/index.html?gi-language=c

- Additional documentation:

  https://gstreamer.freedesktop.org/documentation/additional/index.html?gi-language=c

- API references:

  https://gstreamer.freedesktop.org/documentation/libs.html?gi-language=c

- Caps features:

  https://gstreamer.freedesktop.org/documentation/gstreamer/gstcapsfeatures.html?gi-language=c

- Caps negotiations:

  https://gstreamer.freedesktop.org/documentation/plugin-development/advanced/negotiation.html?gi-language=c

Key points

Keep the following key points in mind when you develop a new plugin:

- Inheritance and base classes
  - Each GStreamer plugin is inherited either from a base template class or directly from the GstElement skeleton class.
  - The GstElement class serves as a minimal framework and developers are responsible for implementing event handling, queries, buffer management, and other essential functionality.
  - When no existing base template class suits the plugin's requirements, developers can use GstElement as a starting point.
- Base template classes
  - GStreamer provides base template classes that handle most events and queries; however, these base classes don't manage buffer handling.
- Buffer management
  - Depending on the chosen base class, developers must implement buffer management and override relevant virtual methods.
  - Some methods have default implementations, while others are pure virtual functions.
- Capabilities
  - Caps are exposed on GstPadTemplates to specify supported media types for a pad.
  - Pads exchange information about their supported formats using caps. The intersection of caps from connected pads determines the actual data format for communication.
- Advantages of base classes
  - Base classes offer virtual methods that developers can override with custom implementations.
  - By leveraging existing base class functionality, developers can focus on specific customizations required by their plugin design.

Notes

- These instructions are based on the `imsdk.lnx.2.0.0.r1-rel` release .
- This guidance is for experienced developers creating a Qualcomm IM SDK plugin to handle unsupported use cases, such as super resolution, by building upon existing plugins.
- Creating a custom GStreamer plugin is an advanced task and is intended only for experienced GStreamer developers.
- Extensive tutorials are available in the open-source domain and on the official GStreamer portal for this task.
- Use this guide as a template and use the open-source resources to learn about developing custom plugins.

Develop a superresolution plugin

This section explains how to develop a new super resolution (qtimlvsuperresolution) plugin.

Notes

- These instructions are based on the `imsdk.lnx.2.0.0.r1-rel` release .
- This guidance is for experienced developers creating a Qualcomm IM SDK plugin to handle unsupported use cases, such as super resolution, by building upon existing plugins.
- Creating a custom GStreamer plugin is an advanced task and is intended only for experienced GStreamer developers.
- Extensive tutorials are available in the open-source domain and on the official GStreamer portal for this task.
- Use this guide as a template and use the open-source resources to learn about developing custom plugins.

Prerequisites

- The eSDK needs to be setup to download and compile application/plugin code. See the Qualcomm Intelligent Multimedia Software Development Kit (IM SDK) Quick Start Guide for instructions on setting up the eSDK and downloading and compiling plugin code.
- Use the source code of the existing plugins as a reference throughout this process.

  CodeLinaro/ le / platform / vendor / qcom-opensource / gst-plugins-qti-oss · GitLab

Plugin directory structure

```
gst-plugins-qti-oss/gst-plugins-mlv<plugin-name>/
    CMakeLists.txt
    config.h.in
    mlv<plugin-name>.c
    mlv<plugin-name>.h
    modules/
        CMakeLists.txt
        ml-video-<plugin-name>-module.h
        ml-v<plugin-name>-<module-name>.c
```

Steps to create superresolution plugin

1. Create the plugin header (`gst-plugins-mlvsuperresolution/mlvsuperresolution.h`).

   Use the existing plugin as a reference:

   gst-plugin-mlvsegmentation/mlvsegmentation.h·imsdk.lnx.2.0.0.r2-rel · CodeLinaro / le / platform / vendor / qcom-opensource / gst-plugins-qti-oss · GitLab

   a. Import the required headers.

   b. Create a structure with the data variables required for processing.

      Example super resolution inheritance chain: GObject > GstObject > GstElement > GstBaseTransform

      GstBaseTransform is suitable for elements where the output size and capabilities are entirely determined by the input size and capabilities

```
struct _GstMLVideoSuperresolution {GstBaseTransform   parent;
                                   GstMLInfo
*mlinfo;
                                   GstVideoInfo       *vinfo;
                                   GstBufferPool
*outpool;
                                   GstMLModule
*module;
                                   gint
mdlenum;
                                   GstStructure
*mlconstants;
};

struct _GstMLVideoSuperresolutionClass {GstBaseTransformClass
```

```
parent;
};
```

Note: The labels variable isn't required for superresolution.

Remove the labels variable and its usage in code.

2. Create the plugin implementation file (`gst-plugins-mlvsuperresolution/mlvsuperresolution.c`).

   Use the existing plugin as a reference:

   gst-plugin-mlvsegmentation/mlvsegmentation.c imsdk.lnx.2.0.0.r2-rel · CodeLinaro / le / platform / vendor / qcom-opensource / gst-plugins-qti-oss · GitLab

   a. Import the `mlvsuperresolution.h` and other required headers.

   b. Define the caps

      - video formats

      ```
      #define GST_ML_VIDEO_SUPERRESOLUTION_VIDEO_FORMATS "{
      RGBA, BGRA, ARGB, ABGR, RGBx, BGRx, xRGB, xBGR, RGB, BGR }
      "
      ```

      - src caps

      ```
      #define GST_ML_VIDEO_SUPER_RESOLUTION_SRC_CAPS \
          "video/x-raw, " \
          "format = (string) " GST_ML_VIDEO_SUPER_RESOLUTION_
      VIDEO_FORMATS "; " \
          "video/x-raw(" GST_CAPS_FEATURE_MEMORY_GBM "), " \
          "format = (string) " GST_ML_VIDEO_SUPER_RESOLUTION_
      VIDEO_FORMATS
      ```

      - sink caps

      ```
      #define GST_ML_VIDEO_SUPER_RESOLUTION_SINK_CAPS \
             "neural-network/tensors"
      ```

   c. Implement following functions

      - Init

        i. Register a custom plugin. Once registered you can access plugin in reference apps with `gst_element_factory_make("plugin-name","plugin-name");`

```
static gboolean plugin_init(GstPlugin * plugin);
```

ii. Initialize superresolution with default parameters. This allows you to control output buffers. On receiving an input buffer, `gst_ml_video_super_resolution_prepare_output_buffer` is called to allocate an output buffer.

```
static void gst_ml_video_super_resolution_
init(GstMLVideoSuperResolution * super_resolution)
```

iii. Initialize callback functions to the function pointer, sets caps, and default properties.

```
static void gst_ml_video_super_resolution_class_
init(GstMLVideoSuperResolutionClass * klass)
```

- Buffer management

  i. Allocates either GBM (Graphics Buffer Manager) or ION Memory depending on the negotiated caps.

```
static GstBufferPool * gst_ml_video_super_resolution_
create_pool (GstMLVideoSuperResolution * super_
resolution,

          GstCaps * caps)
```

  ii. Ensures appropriate buffer pool configuration based on the caps.

```
gst_ml_video_super_resolution_decide_allocation
(GstBaseTransform * base,

GstQuery * query)
```

  iii. Ensures output buffers are allocated from the pool depending on caps.

```
static GstFlowReturn gst_ml_video_super_resolution_
prepare_output_buffer (GstBaseTransform * base,

                  GstBuffer * inbuffer,

                  GstBuffer ** outbuffer)
```

- Caps

  i. Validate the module and then collect the inputs and output capabilities from the

module caps.

```
static gboolean gst_ml_video_super_resolution_set_caps
(GstBaseTransform * base,

GstCaps * incaps,

GstCaps * outcaps)
```

ii. Fix negotiated caps.

```
static GstCaps * gst_ml_video_super_resolution_fixate_
caps (GstBaseTransform * base,

    GstPadDirection direction,

    GstCaps * incaps,

    GstCaps * outcaps)
```

iii. Check whether specified caps are valid or not like video/x-raw, video/x-text, format, height and width.

```
static gboolean caps_has_feature (const GstCaps * caps,
                                  const gchar *
feature)
```

iv. Convert static caps to regular caps.

```
static GstCaps * gst_ml_video_super_resolution_src_caps
(void)
```

v. Convert static caps to regular caps.

```
static GstCaps * gst_ml_video_super_resolution_sink_
caps (void)
```

- Transform

i. Transform sink caps to source caps and vice versa.

```
static GstCaps * gst_ml_video_super_resolution_
transform_caps (GstBaseTransform * base,

       GstPadDirection direction,
```

```
       GstCaps * caps,

       GstCaps * filter)
```

ii. After inference, the postprocessing function (`gst_ml_video_superresolution_module_execute`) will be called based on the module.

```
gst_ml_video_super_resolution_transform
(GstBaseTransform * base,
                                          GstBuffer *
inbuffer,
                                          GstBuffer *
outbuffer)
```

- Property

  i. Set plugin properties, like module and constant.

```
static void gst_ml_video_super_resolution_set_property
(GObject * object,

guint prop_id,

const GValue * value,

GParamSpec * pspec)
```

  ii. Read plugin properties, like module and constant.

```
static void gst_ml_video_super_resolution_get_property
(GObject * object,

guint prop_id,

GValue * value,

GParamSpec * pspec)
```

- De-init

  i. Free all allocated buffers.

```
static void gst_ml_video_super_resolution_finalize
(GObject * object)
```

- Display text/bounding boxes (based on use case only, not required for super

resolution)

i. Overlay result on output frame

```
static gboolean gst_ml_video_<plugin-name>_fill_video_
output(GstMLVideoDetection * detection,

    GArray * predictions,

    GstBuffer * buffer);
```

d. Update the plugin name in the `gst-plugins-mlvsuperresolution/CMakeLists.txt` file.

Use the existing plugin as a reference.

gst-plugin-mlvsegmentation/CMakeLists.txt imsdk.lnx.2.0.0.r2-rel · CodeLinaro / le / platform / vendor / qcom-opensource / gst-plugins-qti-oss · GitLab

Reuse the `config.h.in` file from the existing plugin:

gst-plugin-mlvsegmentation/config.h.in imsdk.lnx.2.0.0.r2-rel · CodeLinaro / le / platform / vendor / qcom-opensource / gst-plugins-qti-oss · GitLab

Overview of Module

- Each model has specific postprocessing steps, which need to be performed on inferenced buffers and update output buffers accordingly, like populating RGB pixels for super resolution and segmentation or calculating bounding boxes for detection.
- `<module-name>.c` defines these postprocessing steps. Module directory contains list of files like below structure

```
  modules/
     "gst-plugins-mlvsuperresolution/modules/ml-
vsuperresolution-<module-name1>.c"
     "gst-plugins-mlvsuperresolution/modules/ml-
vsuperresolution-<module-name2>.c"

Here <module-name> represents the model's name like srnet for
superresolution.
```

Steps to create the srnet module

1. Create the module header `gst-plugins-qti-oss/gst-plugin-base/gst/ml/ml-module-video-super-resolution.h`.

   Use the existing plugin as a reference:

   gst-plugin-base/gst/ml/ml-module-video-super-resolution.h imsdk.lnx.2.0.0.r2-rel · CodeLinaro / le / platform / vendor / qcom-opensource / gst-plugins-qti-oss · GitLab

   a. Import the required `gst` headers.

   b. Create the `execute` function.

   ```
   GST_API gboolean gst_ml_module_video_super_resolution_
   execute(GstMLModule   * module,

   GstMLFrame    * mlframe,

   GstVideoFrame * vframe);
   ```

   This function is required to call the postprocessing function (`gst_ml_module_process`) which is defined in the `ml-vsuperresolution-<module-name>.c` file.

2. Create module the `gst-plugins-mlvsuperresolution/modules/ml-vsuperresolution-srnet.c` file.

   Use the existing pluging as a reference:

   gst-plugin-mlvsegmentation/modules/ml-vsegmentation-deeplab-argmax.c · imsdk.lnx.2.0.0.r1-rel · CodeLinaro / le / platform / vendor / qcom-opensource / gst-plugins-qti-oss · GitLab

   a. Import the `modules/ml-video-superresolution-module.h` header.

   b. Define the model output caps.

   ```
   #define GST_ML_MODULE_CAPS \
       "neural-network/tensors, " \
       "type = (string) { INT8, UINT8, INT32, FLOAT32 }, " \
       "dimensions = (int) < <1, [32, 4096], [32, 4096]> >; " \
       "neural-network/tensors, " \
       "type = (string) { INT8, UINT8, INT32, FLOAT32 }, " \
       "dimensions =  (int) < <1, [32, 4096], [32, 4096], [1,
   3]> >"
   ```

   For Example

   i. Download QuickSRNetLarge-Quantized- Qualcomm AI Hub model from AI Hub

ii. Check the output dimensions of the LiteRT model (See Integrate an AI Hub model in an application).

- Format: `dtype[N x H X W X C]`
- In the above module output caps `<1, [32, 4096], [32, 4096], [1, 3]>`
  - `N = 1`: Model output batch size
  - `H = [32,4096]`: Model output height should be in this range
  - `W= [32,4096]`: Model output width should be in this range
  - `C = [1,3]`: Model output channels should be in this range
  - `dtype`: `int8` for quantized models and `float32` for non-quantized models

iii. For quantized models, check the output node for scale and offset.

Format: `scale*(q-offset)`

c. Implement following functions

- Init

```
gpointer gst_ml_module_open (void);
```

- De-init

```
void gst_ml_module_close (gpointer instance);
```

- Caps

```
GstCaps *gst_ml_module_caps (void);
```

- Configure

```
gboolean gst_ml_module_configure (gpointer instance,
                                  GstStructure *
settings);
```

Check for module dytpe, scale, and offset.

- De-quantize

```
static inline gdouble gst_ml_module_get_dequant_value
(void       *data,

GstMLType  mltype,
```

```
guint      idx,

gdouble    offset,

gdouble    scale);
```

Scale and offset values need to passed for quantized models

- Postprocess

```
gboolean gst_ml_module_process (gpointer    instance,
                                GstMLFrame  *mlframe,
                                gpointer     output);
```

- Inferenced buffer

```
indata = GST_ML_FRAME_BLOCK_DATA (mlframe, 0);
```

- Output buffer to be filled with postprocessed data

```
outdata = GST_VIDEO_FRAME_PLANE_DATA (vframe, 0);
```

- Data type of the model

```
mltype = GST_ML_FRAME_TYPE (mlframe);
```

- Postprocess (for super resolution)

```
for (row = 0; row < GST_VIDEO_FRAME_HEIGHT (vframe row++)
{
    for (column = 0; column < GST_VIDEO_FRAME_WIDTH
(vframe); column++) {
        // Calculate the destination index.
        idx = (((row * GST_VIDEO_FRAME_WIDTH (vframe)) +
column) * bpp) + (row * padding)
        outdata[idx] = gst_ml_module_get_dequant_
value(indata, mltype, idx, submodule->qoffsets[0],
submodule->qscales[0]) * 255;
        outdata[idx + 1] = gst_ml_module_get_dequant_
value(indata, mltype, idx + 1, submodule->qoffsets[0],
submodule->qscales[0]) * 255;
        outdata[idx + 2] = gst_ml_module_get_dequant_
value(indata, mltype, idx + 2, submodule->qoffsets[0],
submodule->qscales[0]) * 255;
        if (bpp == 4)
            outdata[idx + 3] = 0;
```

```
    }
}
```

d. Update the module name in the `gst-plugins-mlvsuperresolution/modules/CMakeLists.txt` file.

Use the existing plugin as a reference:

gst-plugin-mlvsegmentation/modules/CMakeLists.txt · imsdk.lnx.2.0.0.r1-rel · CodeLinaro / le / platform / vendor / qcom-opensource / gst-plugins-qti-oss · GitLab

Steps to update BitBake files for plugin compilation

1. Follow create your Qualcomm IM SDK plugin for instructions on compiling and installing the superresolution plugin.
2. Setup the Wayland display.

```
ssh root@[IP address of the target device]
```

Note: If prompted, enter `oelinux123` as the password for the ssh shell.

```
export XDG_RUNTIME_DIR=/dev/socket/weston && export WAYLAND_
DISPLAY=wayland-1
```

Reference GStreamer pipeline to use qtimlvsuperresolution plugin

```
gst-launch-1.0 -e --gst-debug=2,qtimltflite:7 filesrc location=/etc/
media/video.mp4 ! qtdemux ! queue ! h264parse ! \
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! queue ! tee name=split \
split. ! queue ! qtivcomposer name=mixer \
sink_0::position="<0, 0>" sink_0::dimensions="<960, 1080>" \
sink_1::position="<960, 0>" sink_1::dimensions="<960, 1080>" \
! queue ! waylandsink sync=false fullscreen=true \
split. ! qtimlvconverter ! queue ! \
qtimltflite delegate=external external-delegate-
path=libQnnTFLiteDelegate.so \
external-delegate-options="QNNExternalDelegate,backend_type=htp;"
model=/etc/models/quicksrnetlarge_quantized.tflite ! queue ! \
qtimlvsuperresolution module=srnet constants="qsrnetlarge,q-offsets=
<0.0>,q-scales=<1.0>;" ! \
video/x-raw,width=512,height=512,format=RGB ! queue ! mixer.
```

Note: Input video resolution must be 128x128x3. As the `quicksrnetlarge_quantized.tflite` model input dimensions are 128x128x3.

6 Use AI Hub models and labels with the GStreamer API

This section explains how to use AI Hub models and labels with GStreamer commands on Qualcomm evaluation kits.

Prerequisites

1. Download model files from IoT - Qualcomm AI Hub
2. Filter by **Chipset** and select **Qualcomm QCS6490** for RB3 Gen 2, **Qualcomm QCS9075** for QCS9075 and **Qualcomm QCS8275** for QCS8275.
3. If you haven't downloaded the label files from any of previous sections, download and unzip the labels by running the following commands on your device:

```
wget https://github.com/quic/sample-apps-for-qualcomm-linux/
releases/download/GA1.4-rel/labels.zip
unzip -d /etc/ labels.zip
```

4. Push the models to the `/etc/` directory on the target device .
   - `/etc/models/` for all model files
   - `/etc/labels/` for all label files
   - `/etc/media/` for all video files
   - `/etc/configs/` for all configuration files

Note: The constants for each model may change subject to AI Hub models being updated.

See Integrate an AI Hub model in an application for model constant information.

6.1 Classify images

Before running the pipeline command for a model, follow the required Prerequisites.

Run the following command in the SSH shell before running the classification commands.

```
export XDG_RUNTIME_DIR=/dev/socket/weston && export WAYLAND_
DISPLAY=wayland-1
```

GoogLeNetQuantized

GoogLeNet is a machine learning model that can classify images from the Imagenet dataset. It can also be used to build more complex models for specific use cases.

The AI Hub model is based on this implementation of GoogLeNetQuantized.

- Model: googlenet_quantized.tflite
- Label: imagenet_labels.txt

Note: Classification labels may not appear when using this model for inference

```
gst-launch-1.0 -e --gst-debug=2 \
filesrc location=/etc/media/video.mp4 ! qtdemux ! queue ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! queue ! tee name=split \
split. ! queue ! qtivcomposer name=mixer sink_1::position="<30, 30>"
sink_1::dimensions="<640, 360>" ! queue ! waylandsink sync=true
fullscreen=true \
split. ! queue ! qtimlvconverter ! queue ! qtimltflite
delegate=external external-delegate-path=libQnnTFLiteDelegate.so \
external-delegate-options="QNNExternalDelegate,backend_type=htp;"
model=/etc/models/googlenet_quantized.tflite ! queue ! \
qtimlvclassification threshold=51.0 results=5 module=mobilenet
labels=/etc/labels/imagenet_labels.txt \
extra-operation=softmax constants="Mobilenet,q-offsets=<53.0>,q-
scales=<0.08174873143434525>;" ! video/x-raw,format=BGRA,width=640,
height=360 ! queue ! mixer.
```

Inception-V3-Quantized

InceptionNetV3 is a machine learning model that can classify images from the Imagenet dataset. It can also be used to build more complex models for specific use cases.

This model is post-training quantized to int8 using samples from Google's open images dataset.

The AI Hub model is based on this implementation of Inception-v3-Quantized.

- Model: inception_v3_quantized.tflite
- Label: imagenet_labels.txt

```
gst-launch-1.0 -e --gst-debug=2 \
filesrc location=/etc/media/video.mp4 ! qtdemux ! queue ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! queue ! tee name=split \
split. ! queue ! qtivcomposer name=mixer sink_1::position="<30, 30>"
sink_1::dimensions="<640, 360>" ! queue ! waylandsink sync=true
fullscreen=true \
split. ! queue ! qtimlvconverter ! queue ! qtimltflite
delegate=external external-delegate-path=libQnnTFLiteDelegate.so \
external-delegate-options="QNNExternalDelegate,backend_type=htp;"
model=/etc/models/inception_v3_quantized.tflite ! queue ! \
qtimlvclassification threshold=51.0 results=5 module=mobilenet
labels=/etc/labels/imagenet_labels.txt \
extra-operation=softmax constants="Inception,q-offsets=<33.0>,q-
scales=<0.18740029633045197>;" ! video/x-raw,format=BGRA,width=640,
height=360 ! queue ! mixer.
```

MobileNet-v2-Quantized

MobileNetV2 is a machine learning model that can classify images from the Imagenet dataset. It can also be used to build more complex models for specific use cases.

The AI Hub model is based on this implementation of MobileNet-v2-Quantized.

- Model: mobilenet_v2_quantized.tflite
- Label: imagenet_labels.txt

```
gst-launch-1.0 -e --gst-debug=2 \
filesrc location=/etc/media/video.mp4 ! qtdemux ! queue ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! queue ! tee name=split \
split. ! queue ! qtivcomposer name=mixer sink_1::position="<30, 30>"
sink_1::dimensions="<640, 360>" ! queue ! waylandsink sync=true
fullscreen=true \
```

```
split. ! queue ! qtimlvconverter ! queue ! qtimltflite
delegate=external external-delegate-path=libQnnTFLiteDelegate.so \
external-delegate-options="QNNExternalDelegate,backend_type=htp;"
model=/etc/models/mobilenet_v2_quantized.tflite ! queue ! \
qtimlvclassification threshold=51.0 results=5 module=mobilenet
labels=/etc/labels/imagenet_labels.txt \
extra-operation=softmax constants="Mobilenet,q-offsets=<69.0>,q-
scales=<0.2386164367198944>;" ! video/x-raw,format=BGRA,width=640,
height=360 ! queue ! mixer.
```

MobileNet-v3-Large-Quantized

MobileNet-v3-Large is a machine learning model that can classify images from the Imagenet dataset. It can also be used to build more complex models for specific use cases.

The AI Hub model is based on this implementation of MobileNet-v3-Large-Quantized.

- Model: mobilenet_v3_large_quantized.tflite
- Label: imagenet_labels.txt

```
gst-launch-1.0 -e --gst-debug=2 \
filesrc location=/etc/media/video.mp4 ! qtdemux ! queue ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! queue ! tee name=split \
split. ! queue ! qtivcomposer name=mixer sink_1::position="<30, 30>"
sink_1::dimensions="<640, 360>" ! queue ! waylandsink sync=true
fullscreen=true \
split. ! queue ! qtimlvconverter ! queue ! qtimltflite
delegate=external external-delegate-path=libQnnTFLiteDelegate.so \
external-delegate-options="QNNExternalDelegate,backend_type=htp;"
model=/etc/models/mobilenet_v3_large_quantized.tflite ! queue ! \
qtimlvclassification threshold=51.0 results=5 module=mobilenet
labels=/etc/labels/imagenet_labels.txt \
extra-operation=softmax constants="Mobilenet,q-offsets=<99.0>,q-
scales=<0.18705224990844727>;" ! video/x-raw,format=BGRA,width=640,
height=360 ! queue ! mixer.
```

ResNet18-Quantized

ResNet18 is a machine learning model that can classify images from the Imagenet dataset. It can also be used to build more complex models for specific use cases.

The AI Hub model is based on this implementation of ResNet18Quantized.

- Model: resnet18_quantized.tflite
- Label: imagenet_labels.txt

```
gst-launch-1.0 -e --gst-debug=2 \
filesrc location=/etc/media/video.mp4 ! qtdemux ! queue ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! queue ! tee name=split \
split. ! queue ! qtivcomposer name=mixer sink_1::position="<30, 30>"
sink_1::dimensions="<640, 360>" ! queue ! waylandsink sync=true
fullscreen=true \
split. ! queue ! qtimlvconverter ! queue ! qtimltflite
delegate=external external-delegate-path=libQnnTFLiteDelegate.so \
external-delegate-options="QNNExternalDelegate,backend_type=htp;"
model=/etc/models/resnet18_quantized.tflite ! queue ! \
qtimlvclassification threshold=30.0 results=5 module=mobilenet
labels=/etc/labels/imagenet_labels.txt \
extra-operation=softmax constants="Resnetnet,q-offsets=<68.0>,q-
scales=<0.14944985508918762>;" ! video/x-raw,format=BGRA,width=640,
height=360 ! queue ! mixer.
```

ResNet101-Quantized

ResNet101 is a machine learning model that can classify images from the Imagenet dataset. It can also be used to build more complex models for specific use cases.

The AI Hub model is based on this implementation of ResNet101Quantized.

- Model: resnet101_quantized.tflite
- Label: imagenet_labels.txt

```
gst-launch-1.0 -e --gst-debug=2 \
filesrc location=/etc/media/video.mp4 ! qtdemux ! queue ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! queue ! tee name=split \
split. ! queue ! qtivcomposer name=mixer sink_1::position="<30, 30>"
sink_1::dimensions="<640, 360>" ! queue ! waylandsink sync=true
fullscreen=true \
split. ! queue ! qtimlvconverter ! queue ! qtimltflite
```

```
delegate=external external-delegate-path=libQnnTFLiteDelegate.so \
external-delegate-options="QNNExternalDelegate,backend_type=htp;"
model=/etc/models/resnet101_quantized.tflite ! queue ! \
qtimlvclassification threshold=51.0 results=5 module=mobilenet
labels=/etc/labels/imagenet_labels.txt \
extra-operation=softmax constants="Resnet,q-offsets=<46.0>,q-scales=
<0.2186901867389679 >;" ! video/x-raw,format=BGRA,width=640,
height=360 ! queue ! mixer.
```

ResNeXt50-Quantized

ResNeXt50 is a machine learning model that can classify images from the Imagenet dataset. It can also be used to build more complex models for specific use cases.

The AI Hub model is based on this implementation of ResNeXt50Quantized.

- Model: resnext50_quantized.tflite
- Label: imagenet_labels.txt

```
gst-launch-1.0 -e --gst-debug=2 \
filesrc location=/etc/media/video.mp4 ! qtdemux ! queue ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! queue ! tee name=split \
split. ! queue ! qtivcomposer name=mixer sink_1::position="<30, 30>"
sink_1::dimensions="<640, 360>" ! queue ! waylandsink sync=true
fullscreen=true \
split. ! queue ! qtimlvconverter ! queue ! qtimltflite
delegate=external external-delegate-path=libQnnTFLiteDelegate.so \
external-delegate-options="QNNExternalDelegate,backend_type=htp;"
model=/etc/models/resnext50_quantized.tflite ! queue ! \
qtimlvclassification threshold=35.0 results=5 module=mobilenet
labels=/etc/labels/imagenet_labels.txt \
extra-operation=softmax constants="Resnetnet,q-offsets=<30.0>,q-
scales=<0.06314703077077866>;" ! video/x-raw,format=BGRA,width=640,
height=360 ! queue ! mixer.
```

ResNeXt101-Quantized

ResNeXt101 is a machine learning model that can classify images from the Imagenet dataset. It can also be used to build more complex models for specific use cases.

The AI Hub model is based on this implementation of ResNeXt101Quantized.

- Model: resnext101_quantized.tflite
- Label: imagenet_labels.txt

```
gst-launch-1.0 -e --gst-debug=2 \
filesrc location=/etc/media/video.mp4 ! qtdemux ! queue ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! queue ! tee name=split \
split. ! queue ! qtivcomposer name=mixer sink_1::position="<30, 30>"
sink_1::dimensions="<640, 360>" ! queue ! waylandsink sync=true
fullscreen=true \
split. ! queue ! qtimlvconverter ! queue ! qtimltflite
delegate=external external-delegate-path=libQnnTFLiteDelegate.so \
external-delegate-options="QNNExternalDelegate,backend_type=htp;"
model=/etc/models/resnext101_quantized.tflite ! queue ! \
qtimlvclassification threshold=35.0 results=5 module=mobilenet
labels=/etc/labels/imagenet_labels.txt \
extra-operation=softmax constants="Resnetnet,q-offsets=<37.0>,q-
scales=<0.1848793774843216>;" ! video/x-raw,format=BGRA,width=640,
height=360 ! queue ! mixer.
```

Shufflenet-v2-Quantized

ShufflenetV2 is a machine learning model that can classify images from the Imagenet dataset. It can also be used to build more complex models for specific use cases.

The AI Hub model is based on this implementation of Shufflenet-v2Quantized.

- Model: shufflenet_v2_quantized.tflite
- Label: imagenet_labels.txt

Note: Classification labels may not appear when using this model for inference

```
gst-launch-1.0 -e --gst-debug=2 \
filesrc location=/etc/media/video.mp4 ! qtdemux ! queue ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! queue ! tee name=split \
split. ! queue ! qtivcomposer name=mixer sink_1::position="<30, 30>"
```

```
sink_1::dimensions="<640, 360>" ! queue ! waylandsink sync=true
fullscreen=true \
split. ! queue ! qtimlvconverter ! queue ! qtimltflite
delegate=external external-delegate-path=libQnnTFLiteDelegate.so \
external-delegate-options="QNNExternalDelegate,backend_type=htp;"
model=/etc/models/shufflenet_v2_quantized.tflite ! queue ! \
qtimlvclassification threshold=35.0 results=5 module=mobilenet
labels=/etc/labels/imagenet_labels.txt \
extra-operation=softmax constants="Resnetnet,q-offsets=<69.0>,q-
scales=<0.14428946375846863>;" ! video/x-raw,format=BGRA,width=640,
height=360 ! queue ! mixer.
```

SqueezeNet-1_1-Quantized

SqueezeNet is a machine learning model that can classify images from the Imagenet dataset. It can also be used to build more complex models for specific use cases.

The AI Hub model is based on this implementation of SqueezeNet-1_1Quantized.

- Model: squeezenet1_1_quantized.tflite
- Label: imagenet_labels.txt

Note: Classification labels may not appear when using this model for inference

```
gst-launch-1.0 -e --gst-debug=2 \
filesrc location=/etc/media/video.mp4 ! qtdemux ! queue ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! queue ! tee name=split \
split. ! queue ! qtivcomposer name=mixer sink_1::position="<30, 30>"
sink_1::dimensions="<640, 360>" ! queue ! waylandsink sync=true
fullscreen=true \
split. ! queue ! qtimlvconverter ! queue ! qtimltflite
delegate=external external-delegate-path=libQnnTFLiteDelegate.so \
external-delegate-options="QNNExternalDelegate,backend_type=htp;"
model=/etc/models/squeezenet1_1_quantized.tflite ! queue ! \
qtimlvclassification threshold=25.0 results=5 module=mobilenet
labels=/etc/labels/imagenet_labels.txt \
extra-operation=softmax constants="Resnetnet,q-offsets=<0.0>,q-
scales=<0.16435524821281433>;" ! video/x-raw,format=BGRA,width=640,
height=360 ! queue ! mixer.
```

WideResNet50-Quantized

WideResNet50 is a machine learning model that can classify images from the Imagenet dataset. It can also be used to build more complex models for specific use cases.

The AI Hub model is based on this implementation of WideResNet50-Quantized.

- Model: wideresnet50_quantized.tflite
- Label: imagenet_labels.txt

```
gst-launch-1.0 -e --gst-debug=2 \
filesrc location=/etc/media/video.mp4 ! qtdemux ! queue ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! queue ! tee name=split \
split. ! queue ! qtivcomposer name=mixer sink_1::position="<30, 30>"
sink_1::dimensions="<640, 360>" ! queue ! waylandsink sync=true
fullscreen=true \
split. ! queue ! qtimlvconverter ! queue ! qtimltflite
delegate=external external-delegate-path=libQnnTFLiteDelegate.so \
external-delegate-options="QNNExternalDelegate,backend_type=htp;"
model=/etc/models/wideresnet50_quantized.tflite ! queue ! \
qtimlvclassification threshold=35.0 results=5 module=mobilenet
labels=/etc/labels/imagenet_labels.txt \
extra-operation=softmax constants="Resnet,q-offsets=<44.0>,q-scales=
<0.1439792960882187>;" ! video/x-raw,format=BGRA,width=640,height=360
! queue ! mixer.
```

6.2 Detect objects

Before running the pipeline command for a model, follow the required Prerequisites.

Run the following command to ensure your result is displayed on the connected display.

```
export XDG_RUNTIME_DIR=/dev/socket/weston && export WAYLAND_
DISPLAY=wayland-1
```

Yolo-V7 Quantized

YoloV7 is a machine learning model that predicts bounding boxes and classes of objects in an image.

This model is post-training quantized to int8 using samples from the COCO dataset.

The AI Hub model is based on this implementation of Yolo-v7-Quantized.

- Model: Yolo-v7-Quantized.tflite
- Label: coco_labels.txt

```
gst-launch-1.0 -e --gst-debug=2 \
filesrc location=/etc/media/video.mp4 ! qtdemux ! queue ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! queue ! tee name=split \
split. ! queue ! qtivcomposer name=mixer ! queue ! waylandsink
fullscreen=true \
split. ! queue ! qtimlvconverter ! queue ! qtimltflite
delegate=external external-delegate-path=libQnnTFLiteDelegate.so \
external-delegate-options="QNNExternalDelegate,backend_type=htp;"
model=/etc/models/Yolo-v7-Quantized.tflite ! queue ! \
qtimlvdetection threshold=50.0 results=10 module=yolov8 labels=/etc/
labels/coco_labels.txt constants="YOLOv7,q-offsets=<35.0, 0.0, 0.0>,
q-scales=<3.4220554, 0.0023370725102722645, 1.0>;" ! \
video/x-raw,format=BGRA,width=640,height=360 ! queue ! mixer.
```

Yolov8-Detection-Quantized

Ultralytics Yolov8 is a machine learning model that predicts bounding boxes and classes of objects in an image.

This model is post-training quantized to int8 using samples from the COCO dataset.

The AI Hub model is based on this implementation of YOLOv8-Detection-Quantized.

- Model: YOLOv8-Detection-Quantized.tflite
- Label: coco_labels.txt

```
gst-launch-1.0 -e --gst-debug=2 \
filesrc location=/etc/media/video.mp4 ! qtdemux ! queue ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! queue ! tee name=split \
split. ! queue ! qtivcomposer name=mixer ! queue ! waylandsink
fullscreen=true \
split. ! queue ! qtimlvconverter ! queue ! qtimltflite
```

```
delegate=external external-delegate-path=libQnnTFLiteDelegate.so \
external-delegate-options="QNNExternalDelegate,backend_type=htp,htp_
device_id=(string)0,htp_performance_mode=(string)2,htp_
precision=(string)1;" model=/etc/models/YOLOv8-Detection-Quantized.
tflite ! queue ! \
qtimlvdetection threshold=50.0 results=10 module=yolov8 labels=/etc/
labels/coco_labels.txt constants="YOLOv8,q-offsets=<21.0, 0.0, 0.0>,
q-scales=<3.093529462814331, 0.00390625, 1.0>;" ! \
video/x-raw,format=BGRA,width=640,height=360 ! queue ! mixer.
```

Note: Adding `htp_device_id=(string)0,htp_performance_mode=(string)2,htp_precision=(string)1` to the `external-delegate-options` in the above command allows the model to be run in High Performance mode on HTP.

High Performance mode can be enabled for all other pipelines in the same way.

Yolo-nas-Quantized

YoloNAS is a machine learning model that predicts bounding boxes and classes of objects in an image.

This model is post-training quantized to int8 using samples from the COCO dataset.

The AI Hub model is based on this implementation of Yolo-nas-Quantized.

- Model: Yolo-NAS-Quantized.tflite
- Label: coco_labels.txt

```
gst-launch-1.0 -e --gst-debug=2 \
filesrc location=/etc/media/video.mp4 ! qtdemux ! queue ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! queue ! tee name=split \
split. ! queue ! qtivcomposer name=mixer ! queue ! waylandsink
fullscreen=true \
split. ! queue ! qtimlvconverter ! queue ! qtimltflite
delegate=external external-delegate-path=libQnnTFLiteDelegate.so \
external-delegate-options="QNNExternalDelegate,backend_type=htp;"
model=/etc/models/Yolo-NAS-Quantized.tflite ! queue ! \
qtimlvdetection threshold=50.0 results=10 module=yolov8 labels=/etc/
labels/coco_labels.txt constants="yolo-nas,q-offsets=<37.0,0.0, 0.0>,
q-scales=<3.416602611541748, 0.00390625, 1.0>;" ! \
video/x-raw,format=BGRA,width=640,height=360 ! queue ! mixer.
```

6.3 Apply semantic segmentation to frames of a video

Before running the pipeline command for a model, follow the required Prerequisites.

Run the following command to ensure result is displayed on connected display:

```
export XDG_RUNTIME_DIR=/dev/socket/weston && export WAYLAND_
DISPLAY=wayland-1
```

DeepLabV3-Plus_MobileNet-Quantized

DeepLabV3 Quantized is designed for semantic segmentation at multiple scales and trained on various datasets.

The AI Hub model is based on this implementation of DeepLabV3-Plus-MobileNet-Quantized.

- Model: deeplabv3_plus_mobilenet_quantized.tflite
- Label: voc_labels.txt

```
gst-launch-1.0 -e --gst-debug=2 \
filesrc location=/etc/media/video.mp4 ! qtdemux ! queue ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! queue ! tee name=split \
split. ! queue ! qtivcomposer name=mixer sink_1::alpha=0.5 ! queue !
waylandsink sync=true fullscreen=true \
split. ! queue ! qtimlvconverter ! queue ! qtimltflite
delegate=external external-delegate-path=libQnnTFLiteDelegate.so \
external-delegate-options="QNNExternalDelegate,backend_type=htp;"
model=/etc/models/deeplabv3_plus_mobilenet_quantized.tflite ! queue !
\
qtimlvsegmentation module=deeplab-argmax labels=/etc/labels/voc_
labels.txt \
constants="deeplab,q-offsets=<92.0>,q-scales=<0.04518842324614525>;"
! video/x-raw,format=BGRA,width=256,height=144 ! queue ! mixer.
```

FCN-Resnet50-Quantized

FCN_ResNet50 is a quantized machine learning model that can segment images from the COCO dataset.

The AI Hub model is based on this implementation of FCN-ResNet50-Quantized.

- Model: FCN-ResNet50-Quantized.tflite
- Label: voc_labels.txt

Note: This pipeline is currently not supported on QCS6490.

```
gst-launch-1.0 -e --gst-debug=2 \
filesrc location=/etc/media/video.mp4 ! qtdemux ! queue ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! queue ! tee name=split \
split. ! queue ! qtivcomposer name=mixer sink_1::alpha=0.5 ! queue !
waylandsink sync=true fullscreen=true \
split. ! queue ! qtimlvconverter ! queue ! qtimltflite
delegate=external external-delegate-path=libQnnTFLiteDelegate.so \
external-delegate-options="QNNExternalDelegate,backend_type=htp;"
model=/etc/models/fcn_resnet50_quantized.tflite ! queue ! \
qtimlvsegmentation module=deeplab-argmax labels=/etc/labels/voc_
labels.txt \
constants="deeplab,q-offsets=<0.0>,q-scales=<1.0>;" ! video/x-raw,
format=BGRA,width=256,height=144 ! queue ! mixer.
```

FFNet-40S-Quantized

FFNet-40S-Quantized is a “fuss-free network” that segments street scene images with per-pixel classes like road, sidewalk, and pedestrian.

It’s trained on the cityscapes dataset.

The AI Hub model is based on this implementation of FFNet-40S-Quantized

- Model: ffnet_40s_quantized.tflite
- Label: voc_labels.txt

```
gst-launch-1.0 -e --gst-debug=2 \
filesrc location=/etc/media/video.mp4 ! qtdemux ! queue ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! queue ! tee name=split \
split. ! queue ! qtivcomposer name=mixer sink_1::alpha=0.5 ! queue !
```

```
waylandsink sync=false fullscreen=true \
split. ! queue ! qtimlvconverter ! queue ! qtimltflite
delegate=external external-delegate-path=libQnnTFLiteDelegate.so \
external-delegate-options="QNNExternalDelegate,backend_type=htp;"
model=/etc/models/ffnet_40s_quantized.tflite ! queue ! \
qtimlvsegmentation module=deeplab-argmax labels=/etc/labels/voc_
labels.txt constants="ffnet,q-offsets=<178.0>,q-scales=<0.
31378185749053955>;" ! \
video/x-raw,format=BGRA,width=256,height=144 ! queue ! mixer.
```

FFNet-54S-Quantized

FFNet-54S-Quantized is a “fuss-free network” that segments street scene images with per-pixel classes like road, sidewalk, and pedestrian.

It’s trained on the cityscapes dataset.

The AI Hub model is based on this implementation of FFNet-54S-Quantized.

- Model: ffnet_54s_quantized.tflite
- Label: voc_labels.txt

```
gst-launch-1.0 -e --gst-debug=2 \
filesrc location=/etc/media/video.mp4 ! qtdemux ! queue ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! queue ! tee name=split \
split. ! queue ! qtivcomposer name=mixer sink_1::alpha=0.5 ! queue !
waylandsink sync=false fullscreen=true \
split. ! queue ! qtimlvconverter ! queue ! qtimltflite
delegate=external external-delegate-path=libQnnTFLiteDelegate.so \
external-delegate-options="QNNExternalDelegate,backend_type=htp;"
model=/etc/models/ffnet_54s_quantized.tflite ! queue ! \
qtimlvsegmentation module=deeplab-argmax labels=/etc/labels/voc_
labels.txt constants="ffnet,q-offsets=<178.0>,q-scales=<0.
2929433584213257>;" ! \
video/x-raw,format=BGRA,width=256,height=144 ! queue ! mixer.
```

FFNet-78S-Quantized

FFNet-78S-Quantized is a “fuss-free network” that segments street scene images with per-pixel classes like road, sidewalk, and pedestrian.

It’s trained on the cityscapes dataset.

The AI Hub model is based on this implementation of FFNet-78S-Quantized.

- Model: ffnet_78s_quantized.tflite
- Label: voc_labels.txt

```
gst-launch-1.0 -e --gst-debug=2 \
filesrc location=/etc/media/video.mp4 ! qtdemux ! queue ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! queue ! tee name=split \
split. ! queue ! qtivcomposer name=mixer sink_1::alpha=0.5 ! queue !
waylandsink sync=false fullscreen=true \
split. ! queue ! qtimlvconverter ! queue ! qtimltflite
delegate=external external-delegate-path=libQnnTFLiteDelegate.so \
external-delegate-options="QNNExternalDelegate,backend_type=htp;"
model=/etc/models/ffnet_78s_quantized.tflite ! queue ! \
qtimlvsegmentation module=deeplab-argmax labels=/etc/labels/voc_
labels.txt constants="ffnet,q-offsets=<171.0>,q-scales=<0.
3849360942840576>;" ! \
video/x-raw,format=BGRA,width=256,height=144 ! queue ! mixer.
```

6.4 Upscale images with super resolution

Before running the pipeline command for a model, follow the required Prerequisites

Note: The input video needs dimensions of 128x128 for optimal results.

Run the following command to ensure your result is displayed on the connected display.

```
export XDG_RUNTIME_DIR=/dev/socket/weston && export WAYLAND_
DISPLAY=wayland-1
```

QuickSRNetLarge-Quantized

QuickSRNetLarge is designed for upscaling images on mobile platforms to sharpen them in real-time.

The AI Hub model is based on this implementation of QuickSRNetLarge-Quantized.

- Model: quicksrnetlarge_quantized.tflite

```
gst-launch-1.0 -e --gst-debug=2 \
filesrc location=/etc/media/video.mp4 ! qtdemux ! queue ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! queue ! tee name=split \
split. ! queue ! qtivcomposer name=mixer sink_0::position="<0, 0>"
sink_0::dimensions="<960, 1080>" sink_1::position="<960, 0>" sink_1::
dimensions="<960, 1080>" ! \
queue ! waylandsink sync=true fullscreen=true \
split. ! qtimlvconverter ! queue ! qtimltflite delegate=external
external-delegate-path=libQnnTFLiteDelegate.so \
external-delegate-options="QNNExternalDelegate,backend_type=htp;"
model=/etc/models/quicksrnetlarge_quantized.tflite ! queue ! \
qtimlvsuperresolution module=srnet constants="qsrnetlarge,q-offsets=
<0.0>,q-scales=<1.0>;" ! video/x-raw,format=RGB ! queue ! mixer.
```

QuickSRNetMedium-Quantized

QuickSRNetMedium is designed for upscaling images on mobile platforms to sharpen them in real-time.

The AI Hub model is based on this implementation of QuickSRNetMedium-Quantized.

- Model: quicksrnetmedium_quantized.tflite

```
gst-launch-1.0 -e --gst-debug=2 \
filesrc location=/etc/media/video.mp4 ! qtdemux ! queue ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! queue ! tee name=split \
split. ! queue ! qtivcomposer name=mixer sink_0::position="<0, 0>"
sink_0::dimensions="<960, 1080>" sink_1::position="<960, 0>" sink_1::
dimensions="<960, 1080>" ! \
queue ! waylandsink sync=true fullscreen=true \
split. ! qtimlvconverter ! queue ! qtimltflite delegate=external
external-delegate-path=libQnnTFLiteDelegate.so \
external-delegate-options="QNNExternalDelegate,backend_type=htp;"
model=/etc/models/quicksrnetmedium_quantized.tflite ! queue ! \
qtimlvsuperresolution module=srnet constants="qsrnetlarge,q-offsets=
```

```
<0.0>,q-scales=<1.0>;" ! video/x-raw,format=RGB ! queue ! mixer.
```

QuickSRNetSmall-Quantized

QuickSRNetSmall is designed for upscaling images on mobile platforms to sharpen them in real-time.

The AI Hub model is based on this implementation of QuickSRNetSmall-Quantized.

- Model: quicksrnetsmall_quantized.tflite

```
gst-launch-1.0 -e --gst-debug=2 \
filesrc location=/etc/media/video.mp4 ! qtdemux ! queue ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! queue ! tee name=split \
split. ! queue ! qtivcomposer name=mixer sink_0::position="<0, 0>"
sink_0::dimensions="<960, 1080>" sink_1::position="<960, 0>" sink_1::
dimensions="<960, 1080>" ! \
queue ! waylandsink sync=true fullscreen=true \
split. ! qtimlvconverter ! queue ! qtimltflite delegate=external
external-delegate-path=libQnnTFLiteDelegate.so \
external-delegate-options="QNNExternalDelegate,backend_type=htp;"
model=/etc/models/quicksrnetsmall_quantized.tflite ! queue ! \
qtimlvsuperresolution module=srnet constants="qsrnetlarge,q-offsets=
<0.0>,q-scales=<1.0>;" ! video/x-raw,format=RGB ! queue ! mixer.
```

XLSR-Quantized

XLSR is designed for lightweight real-time upscaling of images.

The AI Hub model is based on this implementation of XLSR-Quantized.

- Model: xlsr_quantized.tflite

```
gst-launch-1.0 -e --gst-debug=2 \
filesrc location=/etc/media/video.mp4 ! qtdemux ! queue ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! queue ! tee name=split \
split. ! queue ! qtivcomposer name=mixer sink_0::position="<0, 0>"
sink_0::dimensions="<960, 1080>" sink_1::position="<960, 0>" sink_1::
dimensions="<960, 1080>" ! \
queue ! waylandsink sync=true fullscreen=true \
split. ! qtimlvconverter ! queue ! qtimltflite delegate=external
external-delegate-path=libQnnTFLiteDelegate.so \
external-delegate-options="QNNExternalDelegate,backend_type=htp;"
```

```
model=/etc/models/xlsr_quantized.tflite ! queue ! \
qtimlvsuperresolution module=srnet constants="qsrnetlarge,q-offsets=
<0.0>,q-scales=<1.0>;" ! video/x-raw,format=RGB ! queue ! mixer.
```

7 Troubleshooting and FAQ

1. By default, when an EVK is connected to an external display, a flower screen is displayed.

   If the flower screen doesn't display, run the following command in a new SSH shell:

   ```
   export GBM_BACKEND=msm && export XDG_RUNTIME_DIR=/dev/socket/
   weston && mkdir -p $XDG_RUNTIME_DIR && weston --continue-
   without-input --idle-time=0
   ```

2. How to debug the sample application using debug logs?

   To troubleshoot execution failures when running AI sample applications or gst-launch-1.0, enable debug logs.

   To enable debug logs in a GStreamer sample application, use the `GST_DEBUG` environment variable to set the debug level.

   The `GST_DEBUG` environment variable controls the verbosity of the debug output. You can set it to different levels, such as:

   - 0: None (no debug information)
   - 1: ERROR (logs all fatal errors)
   - 2: WARNING (logs all warnings)
   - 3: FIXME (logs incomplete code paths)
   - 4: INFO (logs informational messages)
   - 5: DEBUG (logs general debug messages)
   - 6: LOG (logs all log messages)
   - 7: TRACE (logs trace messages)
   - 9: MEMDUMP (logs memory dumps)

   For example, to set the debug level to ERROR, you can use the following command in your terminal:

   ```
   export GST_DEBUG=2
   ```

   If you want to filter debug logs for specific categories, you can specify them in the `GST_`

`DEBUG` variable. For example, to enable debug logs for ML inference plugin and FPS, you can use:

```
export GST_DEBUG=qtiml*:fps*:5
```

3. What common issues prevent a quick out-of-the-box experience for AI sample applications?

   The out-of-the-box experience for AI sample applications is designed to be very quick. However, the following issues are the most common issues that prevent a quick out-of-the-box experience.

   - Failure to load model file

     ```
     0:00:00.042355885 4275 0x5579b9f760 ERROR ml-tflite-engine
     ml-tflite-engine-c-api.cc:578:gst_ml_tflite_engine_new:
     Failed to load model file '/etc/models/googlenet_quantized.
     tflite'!
     ```

     This issue arises when the model file is either missing or not in the correct format.

     Copy the model file to the correct path and ensure you are using the same SDK version for model conversion/quantization and on the target device.

   - Failure to deserialize labels

     ```
     0:00:00.543394063 4676 0x55aca865e0 ERROR mlmodule
     gstmlmodule.c:301:gst_ml_parse_labels: Failed to deserialize
     labels!
     ```

     This occurs when the label file specified at the path isn't present. Ensure that the label file is copied to the device and the specified path is correct.

   - Failure to set module options

     ```
     0:00:00.55129740 4958 0x5561019180 WARN  qtimlvclassification
     mlvclassification.c:986:gst_ml_video_classification_set_caps:
     <mlvideoclassification0> error: Failed to set module options!
     ```

     This occurs when setting constants for LiteRT/Qualcomm AI Engine direct use cases. See Integrate an AI Hub model in an application for steps to read and set model constants correctly.

4. How to measure AI sample app profiling?

   - Preprocessing time

     Before feeding input to the AI inferencing plugin, the input must be preprocessed (including normalization, rescaling, and color inversion). This task is managed by the preprocessing plugin, `qtimlvconverter`. You can measure the preprocessing time

by executing the following command:

```
export GST_DEBUG=qtimlvconverter:6
```

The preprocessing time is displayed in the logs.

```
LOG   qtimlvconverter mlconverter.c:1830:gst_ml_video_
converter_transform:<qtimlvconverter> Conversion took 2.743
ms
```

- Model inference time

  The Qualcomm Intelligent Multimedia SDK supports three AI inferencing plugins that utilize the Qualcomm Neural Processing SDK, LiteRT, and Qualcomm AI Engine Direct frameworks, respectively.

  – qtimlsnpe

  – qtimltflite

  – qtimlqnn

  These plugins can run inference on various hardware, such as CPU, GPU, and HTP. To determine the AI inference time on the hardware, use the following command.

```
export GST_DEBUG=qtiml*:6
```

```
LOG qtimlvtflite mltflite.c:561:gst_ml_tflite_transform:
<qtimltflite> Execute took 3.445 ms
LOG qtimlvtflite mltflite.c:561:gst_ml_tflite_transform:
<qtimltflite> Execute took 3.555 ms
```

- Postprocessing time

  The outputs from AI inferencing plugins are handled by postprocessing plugins. These plugins take the AI model's results and produce elements that can be overlaid on the input stream or used for further computation. For example, text boxes for `qtimlvclassification`, segmentation masks for `qtimlvsegmentation`, etc.

  To measure the processing time for these plugins, use the following command. .. code-block:

```
export GST_DEBUG=qtimlv*:6
```

The postprocessing time is shown in the logs. The following uses the `qtimlvclassification` plugins as an example.

```
LOG    qtimlvclassification mlvclassification.c:1068:gst_ml_
video_classification_transform:<qtimlvclassification>
Categorization tookExecute took 1.962 ms
```

5. How to measure end-to-end FPS of a use case?

   To measure the frames per second (FPS) in a GStreamer pipeline, use the `fpsdisplaysink` element. This element can display the current and average framerate either as an overlay on the video or by printing it to the console.

   Sample apps use the `fpsdisplaysink` plugin to display the FPS of the pipeline directly on the HDMI monitor.

6. What's the easiest way replace an existing model with a custom model in a reference application?

   Ensure you are using the correct model and label file for the sample application.

   Provide the `--model_path` and `--label_path` options with the respective file paths.

   The sample app will use the model and label files from the specified locations.

   See Integrate an AI Hub model in an application for more information.

7. User replaced another supported model (IMSDK supported) in reference application. How to debug performance and accuracy issues?

   To measure performance and accuracy issues of the model, please use the AI SDK tools, you can follow these steps:

   - Performance measurement:

     Use the SDK benchmarking tool. For example, If you are using Qualcomm Neural Processing Engine SDK, use https://docs.qualcomm.com/bundle/publicresource/topics/80-63442-2/tools.html#snpe-bench-py

   - Accuracy debugger:

     – AI Hub model:

       a. Ensure you are using the latest model from AI Hub.

       b. Correctly populate the constants for your selected model. See Integrate an AI Hub model in an application for more information.

       c. For further support on AI Hub model accuracy issues, report your issue on Qualcomm AI Hub slack.

- Custom model

    a. Model quantization is a common cause of accuracy drops. Ensure you are using the correct dataset for model quantization. Users are expected to use a portion of dataset which is an approximation of actual deployment environment for Post Training Quantization (PTQ). For PTQ to give good results, users need to feed decent amount of data to quantize the model, for example, approximately 25-30 RAW images.

    b. Experiment with different model precisions, such as W8A16 and W16A16, to see if there is an improvement in model accuracy.

    c. Use AIMET for Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT) techniques for advanced quantization. See the AIMET documentation for more details.

8. What are the best practices for model quantization using AI SDK?

    - Prepare Calibration Data

      Use representative calibration data that closely matches the data the model will encounter in production. This helps accurately determine the scaling factors and zero points for quantization

    - Choose the Right Quantization Method

      - Post-Training Quantization (PTQ): This method is simpler and faster. Suitable for models where slight accuracy loss is acceptable. It converts a pretrained floating-point model to a quantized model without retraining.
      - Quantization-Aware Training (QAT): This method involves training the model with quantization in mind, which can help maintain higher accuracy. It's more complex but beneficial for models where accuracy is critical.

      See the AIMET documentation for detailed steps.

9. AI SDK provides tools to convert LiteRT to DLC. Do users have to convert LiteRT models to DLC or can LiteRT models accelerate directly on NPU ?

    You don't necessarily have to convert LiteRT models to DLC to accelerate them on the NPU. Qualcomm's AI SDK supports running LiteRT models directly on the NPU using the LiteRT delegate. This means you can leverage NPU capabilities without converting your LiteRT models to DLC format.

10. AI SDK provides tools to convert PyTorch, Onnx, and Tensorflow models to DLC. Which of these is the quickest path for deployment ?

    Converting your models to ONNX first and then to DLC is often the quickest and most flexible approach for deployment.

11. How does a user know, if the model is running on the NPU?

Use the following below tools from Qualcomm:

a. Qualcomm profiler

b. Sysmon (Link TBD)

12. How do users get the benefit of heterogeneous AI engines of Snapdragon platform? How can users run different AI models on different Hardware cores? (GPU, NPU, etc.)

    AI SDK tools and APIs provide options to select the runtime (CPU, GPU or DSP). You can select the appropriate runtime as a command line parameter or use specific C/C++ APIs. See https://docs.qualcomm.com/bundle/publicresource/topics/80-63442-2/tools.html#snpe-net-run for more details.

    The AI sample applications, by default, use the DSP runtime. You can change the runtime using sample application configuration.

    For example, in the `gst-ai-object-detection` sample app modify the `runtime` parameter in the `config_detection.json` file.

    Runtime:

    - `"cpu"`
    - `"gpu"`
    - `"dsp"`

13. How to proceed if model conversion fails with AI SDK?

    Contact Qualcomm Support Forums for support.

14. A custom floating point model provides good accuracy on CPU, but quantized model accuracy is bad on both CPU and NPU. How to debug this?

    There could be problems related to model quantization, see user replaced model FAQ for guidance on model quantization

15. A custom quantized model is running as expected on CPU, but the same model isn't running accurately on NPU. How to debug this?

    TBD

16. When running a sample app, there is no output on the HDMI screen. How to debug this?

    Follow display debugging to debug HDMI display issues.

17. Is it possible to use a HDMI TV instead of HDMI monitor to run sample applications?

    Mostly it will work, if it doesn't work please use HDMI monitor.

18. How to check hardware runtime for AI sample apps?

    Many GStreamer sample applications support inference on various runtimes (CPU, GPU,

and DSP). To determine which runtime the app will use for inference, observe the logs.

- CPU

```
Running app with model: /usr/models/inception_v3_quantized.
tflite and labels: /usr/labels/classification.labels
Using CPU Delegate
Adding all elements to the pipeline...
```

- GPU

```
Running app with model: /usr/models/inception_v3_quantized.
tflite and labels: /usr/labels/classification.labels
Using GPU Delegate
Adding all elements to the pipeline...
```

- DSP

```
Running app with model: /usr/models/inception_v3_quantized.
tflite and labels: /usr/labels/classification.labels
Using DSP Delegate
Adding all elements to the pipeline...
```

19. What are devtool sanity check errors? How to debug them?

    Occasionally, you may see devtool showing sanity check errors.

    Ensure you have sudo access for the host computer. If the error persists, do the following workaround:

    a. Update permissions.

    ```
    umask a+rx
    ```

    b. Disable BitBake sanity checking in the `$ESDK_ROOT/layers/poky/meta/conf/sanity.conf` file.

    ```
    BB_MIN_VERSION = "1.53.1"
    SANITY_ABIFILE = "${TMPDIR}/abi_version"
    SANITY_VERSION ?= "1"
    LOCALCONF_VERSION ?= "2"
    LAYER_CONF_VERSION ?= "7"
    ```

```
SITE_CONF_VERSION ?= "1"

#INHERIT += "sanity"
```

7.1 Further support

Ask your question on the Qualcomm support forum.

8 Model Porting Best Practices

8.1 Export YoloV8 model using Neural Processing Engine SDK

Prerequisites

Install the Qualcomm AI Runtime SDK on a host computer with `Python >= 3.10` and `PyTorch >=1.8`.

For more details, follow Install Qualcomm AI Runtime SDK

Run the following commands on the host computer.

1. Activate your virtual environment.

```
source <venv_path>/bin/activate
```

2. Install the Ultralytics package and export the ONNX model.

```
pip install ultralytics
```

```
yolo export model=yolov8s.pt imgsz=320 format=onnx opset=11
optimize=True simplify=true
```

3. Convert the ONNX model to DLC.

```
snpe-onnx-to-dlc -i yolov8s.onnx
```

4. Convert the DLC to quantized DLC.

   a. Create raw files with the same input dimensions as the exported model and specify the export path in the `input.txt` file.

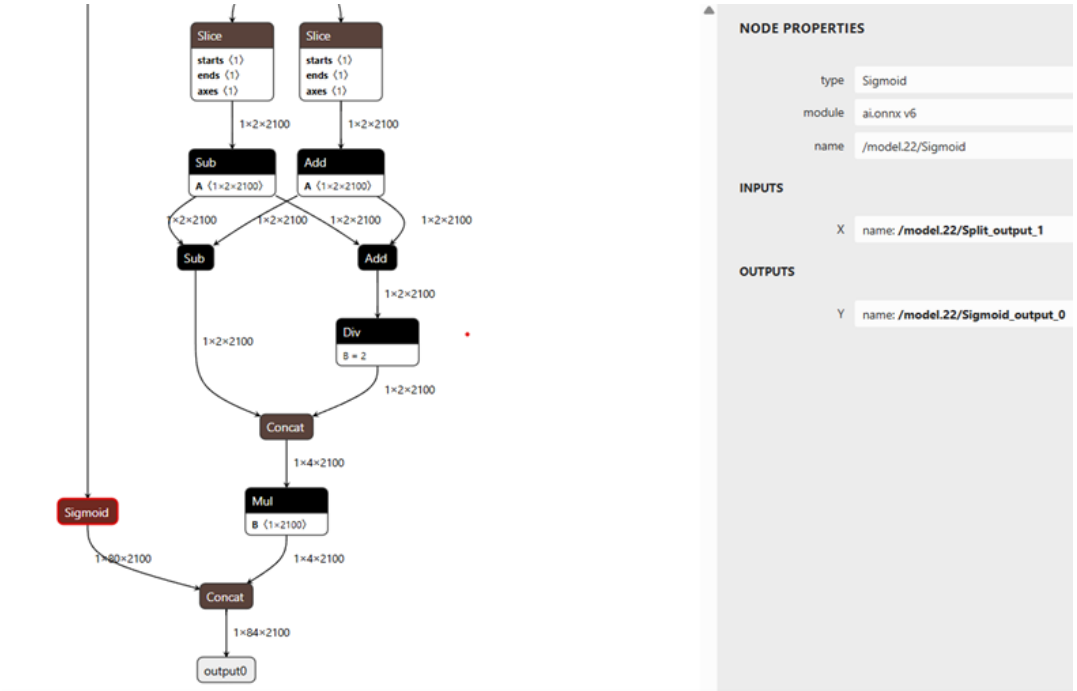
   b. Run the preprocess.py script to generate raw files for YOLOv8 model.

   c. Use the `snpe-dlc-quantize` tool to convert to quantized DLC.

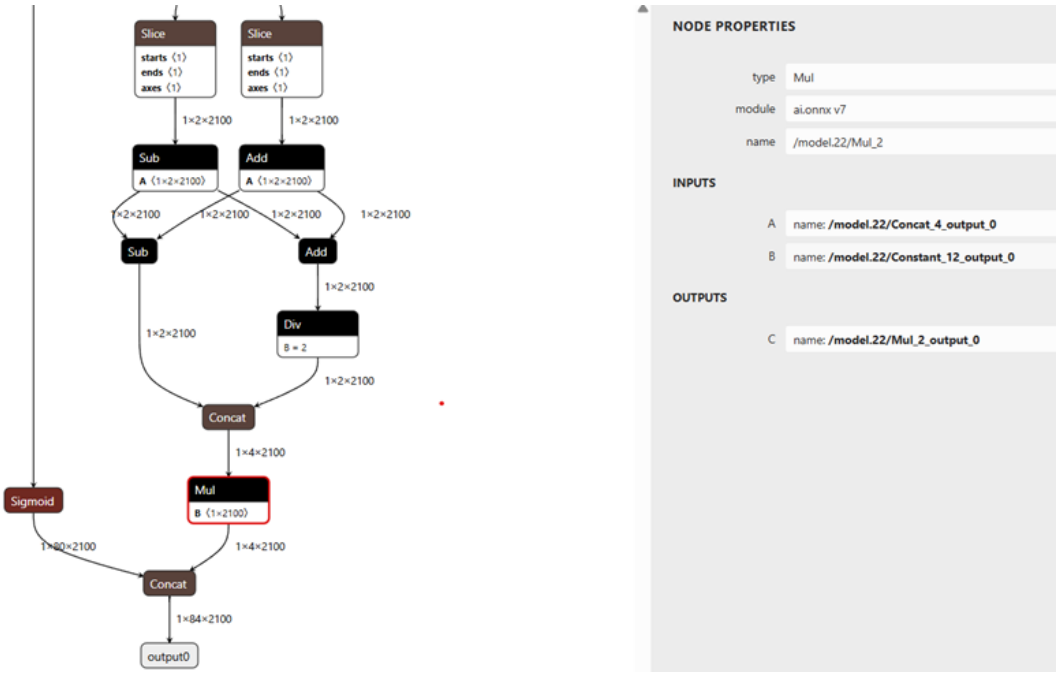
```
snpe-dlc-quantize --input_dlc yolov8s.dlc --input_list input.
txt
```

5. Get the output node of quantized DLC model using Netron.

   a. Select **Sigmoid** as shown below to get the name of the node: `/model.22/Sigmoid`



   b. Select **Mul** as shown below to get its node name: `/model.22/Mul_2`



8.2 Run the demo

1. Download the labels file. See Download model files for Qualcomm Neural Processing SDK.
2. Push the test video file `/etc/media` on the device.

```
scp <video file> root@<IP address of target device>:/etc/media/
```

3. Push the quantized YoloV8 model to the device.

```
scp <model file> root@<IP address of target device>:/etc/models/
```

4. Create a new shell and enable the display.

```
ssh root@<IP address of of device>
```

```
export GBM_BACKEND=msm && export XDG_RUNTIME_DIR=/dev/socket/
weston && mkdir -p $XDG_RUNTIME_DIR && weston --continue-
without-input
```

5. In a new shell, run the following commands:

```
export XDG_RUNTIME_DIR=/dev/socket/weston && export WAYLAND_
DISPLAY=wayland-1
```

```
gst-launch-1.0 qtivcomposer name=mixer ! queue ! waylandsink
sync=false fullscreen=true \
filesrc location=/etc/media/video.mp4 ! qtdemux ! queue !
h264parse ! v4l2h264dec capture-io-mode=4 output-io-mode=4 ! \
video/x-raw,format=NV12 ! queue ! tee name=t ! queue ! mixer. \
t. ! queue ! qtimlvconverter ! queue ! qtimlsnpe delegate=dsp
model=/etc/models/yolov8s_quantized.dlc layers="</model.22/Mul_
2, /model.22/Sigmoid>" ! queue ! \
qtimlvdetection threshold=51.0 results=10 module= yolov8
labels=/etc/labels/yolov8.labels  ! \
video/x-raw,width=640,height=360 ! queue ! mixer.
```

LEGAL INFORMATION

Your access to and use of this material, along with any documents, software, specifications, reference board files, drawings, diagnostics and other information contained herein (collectively this "Material"), is subject to your (including the corporation or other legal entity you represent, collectively "You" or "Your") acceptance of the terms and conditions ("Terms of Use") set forth below. If You do not agree to these Terms of Use, you may not use this Material and shall immediately destroy any copy thereof.

1) Legal Notice.

This Material is being made available to You solely for Your internal use with those products and service offerings of Qualcomm Technologies, Inc. ("**Qualcomm Technologies**"), its affiliates and/or licensors described in this Material, and shall not be used for any other purposes. If this Material is marked as "**Qualcomm Internal Use Only**", no license is granted to You herein, and You must immediately (a) destroy or return this Material to Qualcomm Technologies, and (b) report Your receipt of this Material to qualcomm.support@qti.qualcomm.com. This Material may not be altered, edited, or modified in any way without Qualcomm Technologies' prior written approval, nor may it be used for any machine learning or artificial intelligence development purpose which results, whether directly or indirectly, in the creation or development of an automated device, program, tool, algorithm, process, methodology, product and/or other output. Unauthorized use or disclosure of this Material or the information contained herein is strictly prohibited, and You agree to indemnify Qualcomm Technologies, its affiliates and licensors for any damages or losses suffered by Qualcomm Technologies, its affiliates and/or licensors for any such unauthorized uses or disclosures of this Material, in whole or part.

Qualcomm Technologies, its affiliates and/or licensors retain all rights and ownership in and to this Material. No license to any trademark, patent, copyright, mask work protection right or any other intellectual property right is either granted or implied by this Material or any information disclosed herein, including, but not limited to, any license to make, use, import or sell any product, service or technology offering embodying any of the information in this Material.

THIS MATERIAL IS BEING PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, WHETHER EXPRESSED, IMPLIED, STATUTORY OR OTHERWISE. TO THE MAXIMUM EXTENT PERMITTED BY LAW, QUALCOMM TECHNOLOGIES, ITS AFFILIATES AND/OR LICENSORS SPECIFICALLY DISCLAIM ALL WARRANTIES OF TITLE, MERCHANTABILITY, NON-INFRINGEMENT, FITNESS FOR A PARTICULAR PURPOSE, SATISFACTORY QUALITY, COMPLETENESS OR ACCURACY, AND ALL WARRANTIES ARISING OUT OF TRADE USAGE OR OUT OF A COURSE OF DEALING OR COURSE OF PERFORMANCE. MOREOVER, NEITHER QUALCOMM TECHNOLOGIES, NOR ANY OF ITS AFFILIATES AND/OR LICENSORS, SHALL BE LIABLE TO YOU OR ANY THIRD PARTY FOR ANY EXPENSES, LOSSES, USE, OR ACTIONS HOWSOEVER INCURRED OR UNDERTAKEN BY YOU IN RELIANCE ON THIS MATERIAL.

Certain product kits, tools and other items referenced in this Material may require You to accept additional terms and conditions before accessing or using those items.

Technical data specified in this Material may be subject to U.S. and other applicable export control laws. Transmission contrary to U.S. and any other applicable law is strictly prohibited.

Nothing in this Material is an offer to sell any of the components or devices referenced herein.

This Material is subject to change without further notification.

In the event of a conflict between these Terms of Use and the *Website Terms of Use* on www.qualcomm.com, the *Qualcomm Privacy Policy* referenced on www.qualcomm.com, or other legal statements or notices found on prior pages of the Material, these Terms of Use will control. In the event of a conflict between these Terms of Use and any other agreement (written or click-through, including, without limitation any non-disclosure agreement) executed by You and Qualcomm Technologies or a Qualcomm Technologies affiliate and/or licensor with respect to Your access to and use of this Material, the other agreement will control.

These Terms of Use shall be governed by and construed and enforced in accordance with the laws of the State of California, excluding the U.N. Convention on International Sale of Goods, without regard to conflict of laws principles. Any dispute, claim or controversy arising out of or relating to these Terms of Use, or the breach or validity hereof, shall be adjudicated only by a court of competent jurisdiction in the county of San Diego, State of California, and You hereby consent to the personal jurisdiction of such courts for that purpose.

2) Trademark and Product Attribution Statements.

Qualcomm is a trademark or registered trademark of Qualcomm Incorporated. Arm is a registered trademark of Arm Limited (or its subsidiaries) in the U.S. and/or elsewhere. The Bluetooth® word mark is a registered trademark owned by Bluetooth SIG, Inc. Other product and brand names referenced in this Material may be trademarks or registered trademarks of their respective owners.

Snapdragon and Qualcomm branded products referenced in this Material are products of Qualcomm Technologies, Inc. and/or its subsidiaries. Qualcomm patented technologies are licensed by Qualcomm Incorporated.