CNN to QNN for Linux Host on Linux Target

Updated: Jan 06, 2026 80-63442-10 Rev: AE

Note
This is **Part 2** of the CNN to QNN tutorial for Linux host machines. If you have not completed Part 1, please do so here.

Step 3. Model Build on Linux Host for Linux Target

Once the CNN model has been converted into QNN format, the next step is to build it so it can run on the target device's operating system with `qnn-model-lib-generator`.

Based on the operating system and architecture of your target device, choose one of the following build instructions.

Warning
For cases where the "host machine" and "target device" are the same, you will need to adapt the steps to handle files locally instead of transferring them to a remote device.

Note
Please continue to use the same terminal you were using on your host machine from part 1.

1. Create a directory on your host machine where your newly built files will live by running:

```
mkdir -p /tmp/qnn_tmp
```

2. Navigate to the new directory:

```
cd /tmp/qnn_tmp
```

3. Copy over the QNN `.cpp` and `.bin` model files to `/tmp/qnn_tmp/`:

```
cp "${QNN_SDK_ROOT}/examples/Models/InceptionV3/model/Inception_v3.cpp" "${Q
```

4. Choose the most relevant supported target architecture from the following list:

> ⚠ **Warning**
> If you don't know which one to choose, you can run the following commands on your target device to get more information: `uname -a`, `cat /etc/os-release`, and `gcc --version`.

Target Architecture	Build Instruction
64-bit Linux targets	`x86_64-linux-clang`
64-bit Android devices	`aarch64-android`
Qualcomm's QNX OS	`aarch64-qnx` - Note: This architecture is not supported by default in the
OpenEmbedded Linux (GCC 11.2)	`aarch64-oe-linux-gcc11.2`
OpenEmbedded Linux (GCC 9.3)	`aarch64-oe-linux-gcc9.3`
OpenEmbedded Linux (GCC 8.2)	`aarch64-oe-linux-gcc8.2`
Ubuntu Linux (GCC 9.4)	`aarch64-ubuntu-gcc9.4`
Ubuntu Linux (GCC 7.5)	`aarch64-ubuntu-gcc7.5`

5. On your host machine, set the target architecture of your target device by setting `QNN_TARGET_ARCH` to your device's target architecture:

```
export QNN_TARGET_ARCH="your-target-architecture-from-above"
```

For example:

```
export QNN_TARGET_ARCH="x86_64-linux-clang"
```

6. Run the following command on your host machine to generate the model library:

```
python3 "${QNN_SDK_ROOT}/bin/x86_64-linux-clang/qnn-model-lib-generator" \
    -c "Inception_v3.cpp" \
    -b "Inception_v3.bin" \
    -o model_libs \
    -t ${QNN_TARGET_ARCH}
```

   1. `c` - This indicates the path to the `.cpp` QNN model file.
   2. `b` - This indicates the path to the `.bin` QNN model file. (`b` is optional, but at runtime, the `.cpp` file could fail if it needs the `.bin` file, so it is recommended).
   3. `o` - The path to the output folder.
   4. `t` - Indicate which architecture to build for.
7. Run `ls /tmp/qnn_tmp/model_libs/${QNN_TARGET_ARCH}` and verify that the output file `libInception_v3.so` is inside. - You will use the `libInception_v3.so` file on the target device to execute inferences. - The output `.so` file will be located in the `model_libs` directory, named according to the target architecture.
   - For example: `model_libs/x64/Inception_v3.so` or `model_libs/aarch64/Inception_v3.so`.

Step 4. Use the built model on specific processors

Now that you have an executable version of your model, the next step is to transfer the built model and all necessary files to the target processor, then to run inferences on it.

1. Install all necessary dependencies from Setup.
2. Follow the below SSH setup instructions.
3. Follow the instructions for each specific processor you want to run your model on.

Sub-Step 1: Ensure that you follow the processor-specific Setup instructions for your host machine :doc:`here </general/setup/setup>`.

Sub-Step 2: Set up SSH on the target device.

Here we use `OpenSSH` to copy files with `scp` later on and run scripts on the target device via `ssh`. If that does not work for your target device, feel free to use any other method of transferring the files over. (Ex. `adb` for Android debugging or USB with manual terminal commands on the target device)

1. **Ensure that both the host device and the target device are on the same network for this setup.**
   - Otherwise, `OpenSSH` requires port-forwarding to connect.
2. **On your target device, install `OpenSSH Server` if it is not already installed.**
   - Example for an Ubuntu device:

```
# Update package lists
sudo apt update

# Install OpenSSH server
sudo apt install openssh-server

# Check SSH service status
sudo systemctl status ssh

# Start SSH service if it's not running
sudo systemctl start ssh

# Enable SSH service to start on boot
```

```
sudo systemctl enable ssh
```

> **Note**
> You can turn off the OpenSSH Server service later by running `sudo systemctl stop ssh` if you want to.

3. On your target device, run `ifconfig` to get the IP address of your target device.
4. On your host machine, set a console variable for your target device's `inet addr` from above to see its `ipv4` address (replacing `127.0.0.1` below).

```
export TARGET_IP="127.0.0.1"
```

5. Also set the username of the desired account on your target device (you can find it by running `whoami` on your target device if you are logged into the desired account).

```
export TARGET_USER="your-linux-account-username"
```

6. On your host machine, install `OpenSSH Client` by running the following commands:

```
sudo apt update
sudo apt install openssh-client
sudo systemctl status ssh
sudo systemctl start ssh   # Only if the service is not running
sudo systemctl enable ssh
```

> **Note**
> From this point on you should be able to connect to `ssh` from the terminal. You may need to open another terminal to do so though.

Sub-Step 3: Follow the steps below for whichever processor you would like to run your model on.

CPU

Transferring over all relevant files

1. On the target device, open a terminal and make a destination folder by running:

   ```
   mount -o remount,rw /
   mkdir -p /data/local/tmp
   cd /data/local/tmp
   ln -s /etc/ /data/local/tmp
   chmod -R 777 /data/local/tmp
   mkdir -p /data/local/tmp/inception_v3
   ```

2. On the host device, use `scp` to transfer `libQnnCpu.so` from your host machine to `/data/local/tmp/inception_v3` on the target device.

```
scp "${QNN_SDK_ROOT}/lib/${QNN_TARGET_ARCH}/libQnnCpu.so" "${TARGET_USER}@${
```

3. Use `scp` to transfer the example built model. 1. Update the `x64` folder below to the proper folder for your built model. The folder name depends on your host machine's architecture.

```
scp "/tmp/qnn_tmp/model_libs/x64/libInception_v3.so" "${TARGET_USER}@${TARGE
```

4. Transfer the input data, input list, and script from the QNN SDK examples folder into `/data/local/tmp/inception_v3` on the target device using `scp` in a similar way:

```
scp -r "${QNN_SDK_ROOT}/examples/Models/InceptionV3/data/cropped"  "${TARGET
scp "${QNN_SDK_ROOT}/examples/Models/InceptionV3/data/target_raw_list.txt"
scp "${QNN_SDK_ROOT}/examples/Models/InceptionV3/data/imagenet_slim_labels"
scp "${QNN_SDK_ROOT}/examples/Models/InceptionV3/scripts/show_inceptionv3_cl
```

5. Transfer `qnn-net-run` from `$QNN_SDK_ROOT/bin/$QNN_TARGET_ARCH/qnn-net-run` to `/data/local/tmp/inception_v3` on the target device:

```
scp "$QNN_SDK_ROOT/bin/$QNN_TARGET_ARCH/qnn-net-run" "${TARGET_USER}@${TARGE
```

Doing inferences on the target device processor

1. Open a terminal instance on the target device. 1. Alternatively, you can `ssh` from your Linux host machine, run the following command to `ssh` into your target device. 2. These console variables were set in the above instructions for "Transferring all relevant files".

```
ssh "${TARGET_USER}@${TARGET_IP}"
```

> **Note**
> You will have to log in with your target device's login for that username.

2. Navigate to the directory containing the test files:

```
cd /data/local/tmp/inception_v3
```

3. Run the following command on the target device to execute an inference:

```
./qnn-net-run \
   --model "./libInception_v3.so" \
   --input_list "./target_raw_list.txt" \
   --backend "./libQnnCpu.so" \
   --output_dir "./output"
```

4. Run the following script on the target device to view the classification results:

Note
You can alternatively copy the output folder back to your host machine with `scp` and run the following script there to avoid having to install Python on your target device.

```
python3 ".\show_inceptionv3_classifications.py" \
    -i ".\cropped\raw_list.txt" \
    -o "output" \
    -l ".\imagenet_slim_labels.txt"
```

5. Verify that the classification results in `output` match the following:

File Path	Expected Output
${QNN_SDK_ROOT}/examples/Models/InceptionV3/data/cropped/trash_bin.raw	0.777344 413 ashcan
${QNN_SDK_ROOT}/examples/Models/InceptionV3/data/cropped/chairs.raw	0.253906 832 studio couch
${QNN_SDK_ROOT}/examples/Models/InceptionV3/data/cropped/plastic_cup.raw	0.980469 648 measuring cu
${QNN_SDK_ROOT}/examples/Models/InceptionV3/data/cropped/notice_sign.raw	0.167969 459 brass

GPU

Transferring over all relevant files

1. On the target device, open a terminal and make a destination folder by running:

```
mount -o remount,rw /
mkdir -p /data/local/tmp
cd /data/local/tmp
ln -s /etc/ /data/local/tmp
chmod -R 777 /data/local/tmp
mkdir -p /data/local/tmp/inception_v3
```

2. On the host device, use `scp` to transfer `libQnnGpu.so` from your host machine to `/data/local/tmp/inception_v3` on the target device.

```
scp "${QNN_SDK_ROOT}/lib/${QNN_TARGET_ARCH}/libQnnGpu.so" "${TARGET_USER}@${
```

3. Use `scp` to transfer the example built model. 1. Update the `x64` folder below to the proper folder for your built model. The folder name depends on your host machine's architecture.

```
scp "/tmp/qnn_tmp/model_libs/x64/libInception_v3.so" "${TARGET_USER}@${TARGE
```

4. Transfer the input data, input list, and script from the QNN SDK examples folder into `/data/local/tmp/inception_v3` on the target device using `scp` in a similar way:

```
scp -r "${QNN_SDK_ROOT}/examples/Models/InceptionV3/data/cropped"  "${TARGET
scp "${QNN_SDK_ROOT}/examples/Models/InceptionV3/data/target_raw_list.txt"
scp "${QNN_SDK_ROOT}/examples/Models/InceptionV3/data/imagenet_slim_labels"
scp "${QNN_SDK_ROOT}/examples/Models/InceptionV3/scripts/show_inceptionv3_cl
```

5. Transfer `qnn-net-run` from `$QNN_SDK_ROOT/bin/$QNN_TARGET_ARCH/qnn-net-run` to `/data/local/tmp/inception_v3` on the target device:

```
scp "$QNN_SDK_ROOT/bin/$QNN_TARGET_ARCH/qnn-net-run" "${TARGET_USER}@${TARGE
```

Doing inferences on the target device processor

1. Open a terminal instance on the target device. 1. Alternatively, you can `ssh` from your Linux host machine, run the following command to `ssh` into your target device. 2. These console variables were set in the above instructions for “Transferring all relevant files”.

```
ssh "${TARGET_USER}@${TARGET_IP}"
```

> **Note**
> You will have to log in with your target device’s login for that username.

2. Navigate to the directory containing the test files:

```
cd /data/local/tmp/inception_v3
```

3. Run the following command on the target device to execute an inference:

```
./qnn-net-run \
   --model "./libInception_v3.so" \
   --input_list "./target_raw_list.txt" \
   --backend "./libQnnGpu.so" \
   --output_dir "./output"
```

4. Run the following script on the target device to view the classification results:

> **Note**
> You can alternatively copy the output folder back to your host machine with `scp` and run the following script there to avoid having to install Python on your target device.

```
python3 ".\show_inceptionv3_classifications.py" \
    -i ".\cropped\raw_list.txt" \
    -o "output" \
    -l ".\imagenet_slim_labels.txt"
```

5. Verify that the classification results in `output` match the following:

File Path	Expected Output
${QNN_SDK_ROOT}/examples/Models/InceptionV3/data/cropped/trash_bin.raw	0.777344 413 ashcan
${QNN_SDK_ROOT}/examples/Models/InceptionV3/data/cropped/chairs.raw	0.253906 832 studio couch
${QNN_SDK_ROOT}/examples/Models/InceptionV3/data/cropped/plastic_cup.raw	0.980469 648 measuring cu
${QNN_SDK_ROOT}/examples/Models/InceptionV3/data/cropped/notice_sign.raw	0.167969 459 brass

DSP

⚠ Warning

DSP processors require quantized models instead of full precision models. If you do not have a quantized model, please follow Step 2 of the CNN to QNN tutorial to build one.

Transferring over all relevant files

1. On the target device, open a terminal and make a destination folder by running:

```
mount -o remount,rw /
mkdir -p /data/local/tmp
cd /data/local/tmp
ln -s /etc/ /data/local/tmp
chmod -R 777 /data/local/tmp
mkdir -p /data/local/tmp/inception_v3
```

2. Determine your target device's SnapDragon architecture by looking up your chipset in the Supported Snapdragon Devices table.

3. Update the "X" values below and run the commands to set `DSP_ARCH` to match the version number found in the above table. Only the 2 digits at the end should update, and they should have the same version. Ex. For "V68", the proper value would be `hexagon-v68`.

```
export DSP_VERSION="XX"
export DSP_ARCH="hexagon-v${DSP_VERSION}"
```

4. Use `scp` to transfer `libQnnDsp.so` as well as other necessary executables from your host machine to `/data/local/tmp/inception_v3` on the target device.

```
scp "$QNN_SDK_ROOT/lib/${QNN_TARGET_ARCH}/libQnnDsp.so" "${TARGET_USER}@${TA
scp "${QNN_SDK_ROOT}/lib/${DSP_ARCH}/unsigned/libQnnDspV${DSP_VERSION}Skel.s
scp "${QNN_SDK_ROOT}/lib/${QNN_TARGET_ARCH}/libQnnDspV${DSP_VERSION}Stub.so"
scp "${QNN_SDK_ROOT}/examples/Models/InceptionV3/model_libs/${QNN_TARGET_ARC
```

5. Check the Backend table to see if there are any other processor-specific executables needed for your target processor (`DSP`) and your target device's architecture (`$QNN_TARGET_ARCH`). Use similar syntax above for `scp` to transfer any additional `.so` files listed **below** your selected target architecture in this table. **(There may be none!)**

⚠ **Warning**
Ensure you `scp` the `hexagon-v##` values (in addition to the other architecture files!)

6. Use `scp` to transfer the example built model. Update the `x64` folder below to the proper folder for your built model. The folder name depends on your host machine's architecture.

```
scp "/tmp/qnn_tmp/model_libs/x64/libInception_v3.so"  "${TARGET_USER}@${TARG
```

7. Transfer the input data, input list, and script from the QNN SDK examples folder into `/data/local/tmp/inception_v3` on the target device using `scp` in a similar way:

```
scp -r "${QNN_SDK_ROOT}/examples/Models/InceptionV3/data/cropped"  "${TARGET
scp "${QNN_SDK_ROOT}/examples/Models/InceptionV3/data/target_raw_list.txt"
scp "${QNN_SDK_ROOT}/examples/Models/InceptionV3/data/imagenet_slim_labels"
scp "${QNN_SDK_ROOT}/examples/Models/InceptionV3/scripts/show_inceptionv3_cl
```

8. Transfer `qnn-net-run` from `$QNN_SDK_ROOT/bin/$QNN_TARGET_ARCH/qnn-net-run` to `/data/local/tmp/inception_v3` on the target device:

```
scp "${QNN_SDK_ROOT}/bin/${QNN_TARGET_ARCH}/qnn-net-run" "${TARGET_USER}@${T
```

Doing inferences on the target device processor

1. Open a terminal instance on the target device.

```
ssh "${TARGET_USER}@${TARGET_IP}"
```

ⓘ **Note**
You will have to login with your target device's login for that username.

2. Navigate to the directory containing the test files:

```
cd /data/local/tmp/inception_v3
```

3. Run the following command on the target device to execute an inference:

```
./qnn-net-run \
   --model "./libInception_v3.so" \
   --input_list "./target_raw_list.txt" \
   --backend "./libQnnDsp.so" \
   --output_dir "./output"
```

4. Run the following script on the target device to view the classification results:

ⓘ **Note**

You can alternatively copy the output folder back to your host machine with `scp` and run the following script there to avoid having to install python on your target device.

```
python3 ".\show_inceptionv3_classifications.py" \
    -i ".\cropped\raw_list.txt" \
    -o "output" \
    -l ".\imagenet_slim_labels.txt"
```

5. Verify that the classification results in `output` match the following: 1. `${QNN_SDK_ROOT}/examples/Models/InceptionV3/data/cropped/trash_bin.raw 0.777344 413 ashcan` 2. `${QNN_SDK_ROOT}/examples/Models/InceptionV3/data/cropped/chairs.raw 0.253906 832 studio couch` 3. `${QNN_SDK_ROOT}/examples/Models/InceptionV3/data/cropped/plastic_cup.raw 0.980469 648 measuring cup` 4. `${QNN_SDK_ROOT}/examples/Models/InceptionV3/data/cropped/notice_sign.raw 0.167969 459 brass`

HTP

⚠ **Warning**

HTP processors require quantized models instead of full precision models. If you do not have a quantized model, please follow Step 2 of the CNN to QNN tutorial to build one.

Additional HTP Required Setup

*Running the model on a target device's HTP requires the generation of a **serialized context*.*

On the Linux Host:

1. Navigate to the directory where you built the model in the previous steps:

```
cd /tmp/qnn_tmp
```

2. Users can set the custom options and different performance modes to HTP Backend through the backend config. Please refer to QNN HTP Backend Extensions for various options available in the config.

3. Refer to the example below for creating a backend config file for the QCS6490/QCM6490 target with mandatory options passed in:

```
{
    "graphs": [
        {
            "graph_names": [
                "Inception_v3"
            ],
            "vtcm_mb": 2
        }
    ],
    "devices": [
        {
            "htp_arch": "v68"
        }
```

4. The above config file with minimum parameters such as backend extensions config specified through JSON is given below:

```
{
    "backend_extensions": {
        "shared_library_path": "path_to_shared_library",  // give path to sh
        "config_file_path": "path_to_config_file"         // give path to ba
    }
}
```

5. To generate the context, update `<path to JSON of backend extensions>` below with the config you wrote above, then run the command:

```
"$QNN_SDK_ROOT/bin/${QNN_TARGET_ARCH}/qnn-context-binary-generator" \
    --backend "${QNN_SDK_ROOT}/lib/${QNN_TARGET_ARCH}/libQnnHtp.so" \
    --model "${QNN_SDK_ROOT}/examples/Models/InceptionV3/model_libs/${QNN_TA
    --binary_file "libInception_v3.serialized" \
    --config_file <path to JSON of backend extensions>
```

6. This creates the serialized context at:
   - `${PWD}/output/libInception_v3.serialized.bin`

Transferring over all relevant files

1. On the target device, open a terminal and make a destination folder by running:

```
mount -o remount,rw /
mkdir -p /data/local/tmp
cd /data/local/tmp
ln -s /etc/ /data/local/tmp
chmod -R 777 /data/local/tmp
mkdir -p /data/local/tmp/inception_v3
```

2. Determine your target device's SnapDragon architecture by looking up your chipset in the Supported Snapdragon Devices table.

3. Update the "X" values below and run the commands to set `HTP_ARCH` to match the version number found in the above table. Only the 2 digits at the end should update, and they should have the same version. Ex. For "V68", the proper value would be `hexagon-v68`.

```
$HTP_VERSION="XX"
$HTP_ARCH="hexagon-v${HTP_VERSION}"
```

4. Use `scp` to transfer `libQnnHtp.so` as well as other necessary executables from your host machine to `/data/local/tmp/inception_v3` on the target device.

```
scp "$QNN_SDK_ROOT/lib/${QNN_TARGET_ARCH}/libQnnHtp.so" "${TARGET_USER}@${TA
scp "${QNN_SDK_ROOT}/lib/${DSP_ARCH}/unsigned/*" "${TARGET_USER}@${TARGET_IP
scp "${QNN_SDK_ROOT}/lib/${QNN_TARGET_ARCH}/libQnnHtpV${HTP_VERSION}Stub.so"
scp "${QNN_SDK_ROOT}/examples/Models/InceptionV3/output/Inception_v3.seriali
```

5. Check the Backend table to see if there are any other processor-specific executables needed for your target processor (`DSP`) and your target device's architecture (`$QNN_TARGET_ARCH`). Use similar syntax above for `scp` to transfer any additional `.so` files listed **below** your selected target architecture in this table. **(There may be none!)**

> ⓘ **Note**
> Ensure you transfer the `hexagon-v##` values (in addition to the other architecture files!)

6. Use `scp` to transfer the example built model. Update the `x64` folder below to the proper folder for your built model. The folder name depends on your host machine's architecture.

```
scp "/tmp/qnn_tmp/model_libs/x64/libInception_v3.so"  "${TARGET_USER}@${TARG
```

7. Transfer the input data, input list, and script from the QNN SDK examples folder into `/data/local/tmp/inception_v3` on the target device using `scp` in a similar way:

```
scp -r "${QNN_SDK_ROOT}/examples/Models/InceptionV3/data/cropped"  "${TARGET
scp "${QNN_SDK_ROOT}/examples/Models/InceptionV3/data/target_raw_list.txt"
scp "${QNN_SDK_ROOT}/examples/Models/InceptionV3/data/imagenet_slim_labels"
scp "${QNN_SDK_ROOT}/examples/Models/InceptionV3/scripts/show_inceptionv3_cl
```

8. Transfer `qnn-net-run` from `$QNN_SDK_ROOT/bin/$QNN_TARGET_ARCH/qnn-net-run` to `/data/local/tmp/inception_v3` on the target device:

```
scp "${QNN_SDK_ROOT}/bin/${QNN_TARGET_ARCH}/qnn-net-run" "${TARGET_USER}@${T
```

9. On the target device, set the environment variables:

```
export LD_LIBRARY_PATH="/data/local/tmp/inception_v3"
export ADSP_LIBRARY_PATH="/data/local/tmp/inception_v3"
```

Doing inferences on the target device processor

1. Open a terminal instance on the target device.

```
ssh "${TARGET_USER}@${TARGET_IP}"
```

> ⓘ **Note**
> You will have to login with your target device's login for that username.

2. On your target device, navigate to the directory containing the test files:

```
cd /data/local/tmp/inception_v3
```

3. Run the following command on the target device to execute an inference:

```
./qnn-net-run \
   --backend "libQnnHtp.so" \
   --input_list "target_raw_list.txt" \
   --retrieve_context "Inception_v3.serialized.bin" \
   --output_dir "./output"
```

4. Run the following script on the target device to view the classification results:

> **Note**
> You can alternatively copy the output folder back to your host machine with `scp` and run the following script there to avoid having to install python on your target device.
>
> ```
> python3 ".\show_inceptionv3_classifications.py" \
>     -i ".\cropped\raw_list.txt" \
>     -o "output" \
>     -l ".\imagenet_slim_labels.txt"
> ```

5. Verify that the classification results in `output` match the following: 1. `${QNN_SDK_ROOT}/examples/Models/InceptionV3/data/cropped/trash_bin.raw 0.777344 413 ashcan` 2. `${QNN_SDK_ROOT}/examples/Models/InceptionV3/data/cropped/chairs.raw 0.253906 832 studio couch` 3. `${QNN_SDK_ROOT}/examples/Models/InceptionV3/data/cropped/plastic_cup.raw 0.980469 648 measuring cup` 4. `${QNN_SDK_ROOT}/examples/Models/InceptionV3/data/cropped/notice_sign.raw 0.167969 459 brass`

LPAI

> **Warning**
> LPAI Backend for model offline preparation was introduced at the time of v4 (**eNPU hardware version 4**). This means you must build the model first, then transfer it to the target device. The offline prepared model was executed by LPAI SDK released separately, but starting v5 (**eNPU hardware version 5**) the execution of serialized model is supported by QNN SDK and additional LPAI SDK is not required.

The offline generated model for v4 must be executed via the LPAI SDK. You will have to sign in to access the LPAI SDK, and it may be dependent on filling out specific paperwork for your account to access the SDK.

Preparing LPAI Configuration Files for Model Preparation

EXAMPLE of `config.json` file:

```
{
   "backend_extensions": {
      "shared_library_path": "${QNN_SDK_ROOT}/lib/x86_64-linux-clang/libQnnLpaiI
      "config_file_path": "./lpaiParams.conf"
   }
}
```

EXAMPLE of `lpaiParams.conf` file that includes only preparation parameters:

```
{
   "lpai_backend": {
      "target_env": "adsp",
      "enable_hw_ver": "v5"
   },
   "lpai_graph": {
      "prepare": {
         "enable_batchnorm_fold": true,
         "exclude_io": false
      }
   }
}
```

To configure `lpaiParams.conf`, consider using the following optional settings:

```
lpai_backend
   "target_env"              "arm/adsp/x86/tensilica, default adsp"
   "enable_hw_ver"           "v4,v5 default v5"
lpai_graph
   prepare
      "enable_batchnorm_fold"   "true/false,     default false"
      "exclude_io"              "true/false,     default false"
```

Using the above `config.json` and `lpaiParams.conf` you can use `qnn-context-binary-generator` to build the LPAI offline model.

When files are mentioned, ensure that they have the relative or absolute path to that value.

```
cd ${QNN_SDK_ROOT}/examples/QNN/converter/models
$LD_LIBRARY_PATH=$LD_LIBRARY_PATH:${QNN_SDK_ROOT}/lib/x86_64-linux-clang \
& ${QNN_SDK_ROOT}/bin/x86_64-linux-clang/qnn-context-binary-generator \
              --backend ${QNN_SDK_ROOT}/lib/x86_64-linux-clang/libQnnLpai.so \
              --model ${QNN_SDK_ROOT}/examples/Models/InceptionV3/model_libs/x8
              --config_file <config.json> \
              --log_level verbose \
              --backend_binary <output_graph.eai> \
              --binary_file <lpai_graph_serialized>
```

ⓘ **Note**

- Use generated **output_graph.eai** file in the format of eai model to be executed by LPAI SDK for **v4** version only.

- Use generated **lpai_graph_serialized.bin** file in QNN format to be executed directly by QNN SDK for **v5 and higher** versions.

Running LPAI Emulation Backend on Linux x86

While LPAI Backend on x86_64 Linux CPU is designed for offline model generation as described above, it still can run in HW simulation mode where internally it executes prepare and execution steps together.

EXAMPLE of `config.json` file for ARM:

```
{
   "backend_extensions": {
      "shared_library_path": "${QNN_SDK_ROOT}/lib/x86_64-linux-clang/libQnnLpai
      "config_file_path": "./lpaiParams.conf"
   }
}
```

EXAMPLE of `lpaiParams.conf` file that includes preparation and execution parameters:

```
{
   "lpai_backend": {
      "target_env": "x86",
      "enable_hw_ver": "v5"
   },
   "lpai_graph": {
      "prepare": {
         "enable_batchnorm_fold": false,
         "exclude_io": false
      },
      "execute": {
         "fps": 1,
         "ftrt_ratio": 10,
         "client_type": "real_time",
```

To configure `lpaiParams.conf`, consider using the following optional settings:

```
lpai_backend
   "target_env"                 "arm/adsp/x86/tensilica, default adsp"
   "enable_hw_ver"              "v4,v5 default v5"
lpai_graph
   prepare
      "enable_batchnorm_fold"   "true/false,     default false"
      "exclude_io"              "true/false,     default false"
   execute
      "fps"                     "Specify the fps rate number, used for clock vot
      "ftrt_ratio"              "Specify the ftrt_ratio number, default 10"
      "client_type"             "real_time/non_real_time, defult real_time"
      "affinity"                "soft/hard, default soft"
      "core_selection"          "Specify the core number, default 0"
```

Using the above `config.json` and `lpaiParams.conf` you can use `qnn-net-run` to directly execute LPAI backend in simulation mode.

With the appropriate libraries compiled, `qnn-net-run` is used with the following:

> **Note**
> If full paths are not given to `qnn-net-run`, all libraries must be added to LD_LIBRARY_PATH and be discoverable by the system library loader.

```
$ cd ${QNN_SDK_ROOT}/examples/QNN/converter/models
$ ${QNN_SDK_ROOT}/bin/x86_64-linux-clang/qnn-net-run \
              --backend ${QNN_SDK_ROOT}/lib/x86_64-linux-clang/libQnnLpai.so \
              --model ${QNN_SDK_ROOT}/examples/QNN/example_libs/x86_64-linux-cla
              --input_list ${QNN_SDK_ROOT}/examples/QNN/converter/models/input_
              --config_file <config.json>
```

Outputs from the run will be located at the default ./output directory.

Running LPAI on Target Platform via ARM Backend using offline prepared graph

While graph prepare is predefined by offline generation step the Execution paramaters still can be changed at the execution time.

EXAMPLE of `config.json` file:

```
{
   "backend_extensions": {
      "shared_library_path": "./libQnnLpaiNetRunExtensions.so",
      "config_file_path": "./lpaiParams.conf"
   }
}
```

EXAMPLE of `lpaiParams.conf` file that includes only execution parameters:

```
{
   "lpai_backend": {
      "target_env": "adsp",
      "enable_hw_ver": "v5"
   },
   "lpai_graph": {
      "execute": {
         "fps": 1,
         "ftrt_ratio": 10,
         "client_type": "real_time",
         "affinity": "soft",
         "core_selection": 0
      }
   }
```

To configure `lpaiParams.conf`, consider using the following optional settings:

```
lpai_graph
   execute
      "fps"                     "Specify the fps rate number, used for clock vo
      "ftrt_ratio"              "Specify the ftrt_ratio number, default 10"
      "client_type"             "real_time/non_real_time, defult real_time"
      "affinity"                "soft/hard, default soft"
      "core_selection"          "Specify the core number, default 0"
```

Using the above `config.json` and `lpaiParams.conf` you can use by `qnn-net-run` to directly execute LPAI backend on target.

> ⓘ **Note**
> Running the LPAI Backend on an Android via ARM target is supported only for **offline prepared graphs**. Follow **“Preparing LPAI Configuration Files Model Preparation”** paragraph first to prepare the graph on x86_64 host and then push the serialized context binary to the device.

In order to run on a particular target platform, the libraries compiled for that target must be used. Below are examples. The QNN_TARGET_ARCH variable can be used to specify the appropriate library for the target.

```
$ export QNN_TARGET_ARCH=aarch64-android
$ export DSP_ARCH=hexagon-v79
$ export DSP_VER=V79
$ export HW_VER=v5
```

To execute the LPAI backend on Android device the following conditions must be met:

```
1. ${QNN_SDK_ROOT}/lib/lpai-${HW_VER}/unsigned/libQnnLpaiSkel.so has to be signe
2. qnn-net-run to be executed with root permissions
```

After offline model preparation of `./output/lpai_graph_serialized.bin` push the serialized model to device: .. code-block:

```
$ adb push ./output/lpai_graph_serialized.bin /data/local/tmp/LPAI
```

Push configuration files to device:

```
$ adb push ./config.json /data/local/tmp/LPAI
$ adb push ./lpaiParams.conf /data/local/tmp/LPAI
```

Push LPAI artifacts to device:

```
$ adb push ${QNN_SDK_ROOT}/lib/lpai-${HW_VER}/libQnnLpai.so /data/local/tmp/LPA
$ adb push ${QNN_SDK_ROOT}/lib/lpai-${HW_VER}/libQnnLpaiNetRunExtensions.so /da
$ adb push ${QNN_SDK_ROOT}/lib/lpai-${HW_VER}/libQnnLpaiStub.so /data/local/tmp/
$ # Additionally, the LPAI requires Hexagon specific libraries
$ adb push ${QNN_SDK_ROOT}/lib/lpai-${HW_VER}/unsigned/libQnnLpaiSkel.so /data/
```

Push the input data and input lists to device:

```
$ adb push ${QNN_SDK_ROOT}/examples/QNN/converter/models/input_data_float /data/
$ adb push ${QNN_SDK_ROOT}/examples/QNN/converter/models/input_list_float.txt /
```

Push the `qnn-net-run` tool:

```
$ adb push ${QNN_SDK_ROOT}/bin/aarch64-android/qnn-net-run /data/local/tmp/LPAI
```

Setup the environment on device:

```
$ adb shell
$ cd /data/local/tmp/LPAI
$ export LD_LIBRARY_PATH=/data/local/tmp/LPAI
$ export ADSP_LIBRARY_PATH="/data/local/tmp/LPAI"
```

Finally, use `qnn-net-run` for execution with the following:

```
$ ./qnn-net-run --backend libQnnLpai.so --retrieve_context lpai_graph_serialize
```

Running LPAI on Target Platform via Native aDSP Backend using offline prepared graph

While graph prepare is predefined by offline generation step the Execution paramaters still can be changed at the execution time.

EXAMPLE of `config.json` file:

```
{
   "backend_extensions": {
      "shared_library_path": "./libQnnLpaiNetRunExtensions.so",
      "config_file_path": "./lpaiParams.conf"
   }
}
```

EXAMPLE of `lpaiParams.conf` file that includes only execution parameters:

```
{
   "lpai_backend": {
      "target_env": "adsp",
      "enable_hw_ver": "v5"
   },
   "lpai_graph": {
      "execute": {
         "fps": 1,
         "ftrt_ratio": 10,
         "client_type": "real_time",
         "affinity": "soft",
         "core_selection": 0
      }
   }
```

To configure `lpaiParams.conf`, consider using the following optional settings:

```
lpai_graph
   execute
      "fps"                     "Specify the fps rate number, used for clock vot
      "ftrt_ratio"              "Specify the ftrt_ratio number, default 10"
      "client_type"             "real_time/non_real_time, defult real_time"
      "affinity"                "soft/hard, default soft"
      "core_selection"          "Specify the core number, default 0"
```

Using the above `config.json` and `lpaiParams.conf` you can use by `qnn-net-run` to directly execute LPAI backend on target.

> **Note**
> Running the LPAI Backend on an Android via native aDSP target is supported only for **offline prepared graphs**. Follow **“Preparing LPAI Configuration Files Model Preparation”** paragraph first to prepare the graph on x86_64 host and then push the serialized context binary to the device.

In order to run on a particular target platform, the libraries compiled for that target must be used. Below are examples. The QNN_TARGET_ARCH variable can be used to specify the appropriate library for the target.

```
$ export QNN_TARGET_ARCH=aarch64-android
$ export DSP_ARCH=hexagon-v79
$ export DSP_VER=V79
$ export HW_VER=v5
```

To execute the LPAI backend on Android device the following conditions must be met:

```
1. The following artifacts in ${QNN_SDK_ROOT}/lib/lpai-${HW_VER}/unsigned  has t
   a. libQnnLpai.so
   b. libQnnLpaiNetRunExtensions.so
2. The following artifacts in ${QNN_SDK_ROOT}/lib/${DSP_ARCH}/unsigned  has to b
   a. libQnnHexagonSkel_dspApp.so
   b. libQnnNetRunDirect${DSP_VER}Skel.so
2. qnn-net-run to be executed with root permissions
```

After offline model preparation of `./output/lpai_graph_serialized.bin` push the serialized model to device: .. code-block:

```
$ adb push ./output/lpai_graph_serialized.bin /data/local/tmp/LPAI
```

Push configuration files to device:

```
$ adb push ./config.json /data/local/tmp/LPAI
$ adb push ./lpaiParams.conf /data/local/tmp/LPAI
```

Push the necessary libraries to device:

```
$ adb push ${QNN_SDK_ROOT}/lib/lpai-${HW_VER}/unsigned/libQnnLpai.so /data/loca
$ adb push ${QNN_SDK_ROOT}/lib/lpai-${HW_VER}/unsigned/libQnnLpaiNetRunExtensior
$ adb push ${QNN_SDK_ROOT}/lib/${DSP_ARCH}/unsigned/libQnnHexagonSkel_dspApp.so
$ adb push ${QNN_SDK_ROOT}/lib/${DSP_ARCH}/unsigned/libQnnNetRunDirect${DSP_VER}
```

Push the input data and input lists to device:

```
$ adb push ${QNN_SDK_ROOT}/examples/QNN/converter/models/input_data_float /data/
$ adb push ${QNN_SDK_ROOT}/examples/QNN/converter/models/input_list_float.txt /
```

Push the `qnn-net-run` tool and its dependent libraries:

```
$ adb push ${QNN_SDK_ROOT}/bin/${QNN_TARGET_ARCH}/qnn-net-run /data/local/tmp/LP
$ adb push ${QNN_SDK_ROOT}/lib/${QNN_TARGET_ARCH}/libQnnNetRunDirect${DSP_VER}St
```

Setup the environment on device:

```
$ adb shell
$ cd /data/local/tmp/LPAI
$ export LD_LIBRARY_PATH=/data/local/tmp/LPAI
$ export ADSP_LIBRARY_PATH="/data/local/tmp/LPAI"
$ export DSP_VER=V79
```

Finally, use `qnn-net-run` for execution with the following:

```
$ ./qnn-net-run --backend libQnnLpai.so --direct_mode adsp --retrieve_context qr
```

← Previous
CNN to QNN for Linux Host

Next →
CNN to QNN for Linux Host on Windows Target

Light Dark Auto

Qualcomm relentlessly innovates to deliver intelligent computing everywhere, helping the world tackle some of its most important challenges. Our leading-edge AI, high performance, low-power computing, and unrivaled connectivity deliver proven solutions that transform major industries. At Qualcomm, we are engineering human progress.

Quick links

Products

Support

Partners

Contact us

Developer

Company info

About us

Careers

Investors

News & media

Our businesses

Email Subscriptions

Stay connected

Get the latest Qualcomm and industry information delivered to your inbox.

Subscribe

Manage your subscription

Terms of Use Privacy Cookie Policy Accessibility Statement Cookies Settings Language: English (US)