AN12580
NXP eIQ™ Machine Learning Software Development Environment for QorIQ Layerscape Applications Processors

Rev. 3 — 12/2020
Application Note

Contents

1 Introduction

Machine Learning (ML) is a computer science domain that has its roots in the 1960s. ML provides algorithms capable of finding patterns and rules in data. ML is a category of algorithm that allows software applications to become more accurate in predicting outcomes without being explicitly programmed. The basic premise of ML is to build algorithms that can receive input data and use statistical analysis to predict an output while updating output as new data becomes available.

In 2010, the so-called deep learning started. It is a fast-growing subdomain of ML, based on Neural Networks (NN). Inspired by the human brain, deep learning achieved state-of-the-art results in various tasks; for example, Computer Vision (CV) and Natural Language Processing (NLP). Neural networks are capable of learning complex patterns from millions of examples. A huge adaptation is expected in the embedded world, where NXP is the leader. NXP created eIQ machine learning software for QorIQ Layerscape applications processors, a set of ML tools which allows developing and deploying ML applications on the QorIQ Layerscape family of devices.

This document provides guidance for the supported ML software for the QorIQ Layerscape processors. The document is divided into separate sections, starting with the NXP eIQ introduction, eIQ components, and the step-by step guide for running some examples.

2 NXP eIQ software introduction

The NXP eIQ machine learning software development environment provides a shell script to install machine learning applications targeted at NXP QorIQ Layerscape processors. The NXP eIQ software is concerned only with neural networks inference and standard machine-learning algorithms, leaving neural network training to other specialized software tools and dedicated hardware. The NXP eIQ is continuously expanding to include data-acquisition and curation tools and model conversion for a wide range of NN frameworks and inference engines, such as TensorFlow Lite, Arm NN, and Arm Compute Library.

The current version of NXP eIQ software delivers machine learning enablement by providing ML support in LSDK for the two QorIQ Layerscape processors LS1046A and LX2160A. The NXP eIQ software contains these main features:

- OpenCV 4.0.1
- Arm Compute Library 20.08
- Arm NN 20.08
- TensorFlow Lite 2.2.0
- Onnxruntime 1.1.2
- PyTorch 1.6.0

For up-to-date information about NXP machine learning solutions, see the official NXP web page for machine learning and artificial intelligence.

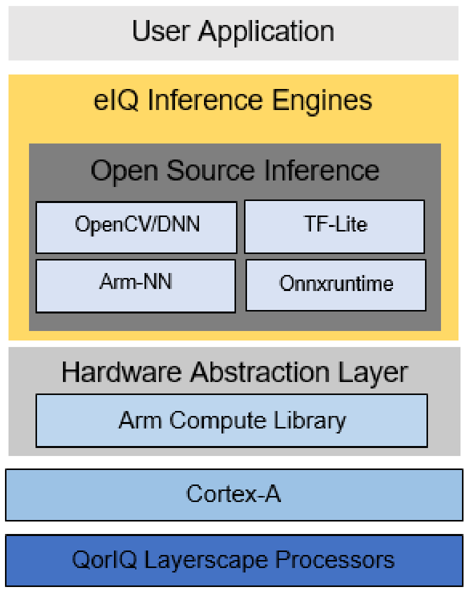


Figure 1. NXP eIQ machine learing software

3 Building eIQ Components

Cross-building eIQ components depends on lots of dependencies on host machine, so to avoid possible build issues in uncertain environment, please ensure to build it in docker container generated by FlexBuild as below:

Prepare FlexBuild docker:

1. Download LSDK flexbuild tarball from www.nxp.com/lsdk
2. Generate fbubuntu docker container.

```
$ tar xvzf flexbuild_<version>.tgz
$ cd flexbuild_<version>
$ source setup.env
$ flex-builder docker
[root@fbubuntu flexbuild]$ source setup.env
[root@fbubuntu flexbuild]$ flex-builder -h
```

Build eIQ in FlexBuild docker:

```
[root@fbubuntu flexbuild]$ source setup.env
```

Build only eIQ components:

```
Usage: flex-builder -c <component> [ -a <arch> ]

[root@fbubuntu flexbuild]$ flex-builder -c eiq # build all eIQ components (include armnn, tflite, opencv,
onnx, etc)

[root@fbubuntu flexbuild]$ flex-builder -c armnn # build armnn

[root@fbubuntu flexbuild]$ flex-builder -c tflite # build tflite

[root@fbubuntu flexbuild]$ flex-builder -c opencv # build opencv

[root@fbubuntu flexbuild]$ flex-builder -c caffe # build caffe

[root@fbubuntu flexbuild]$ flex-builder -c onnx # build onnx
```

```
[root@fbubuntu flexbuild]$ flex-builder -c onnxruntime # build onnxruntime
```

Build full LSDK with eIQ components for target arm64 on x86 host machine:

```
[root@fbubuntu flexbuild]$ flex-builder -i clean-eiq

[root@fbubuntu flexbuild]$ flex-builder -i mkrfs

[root@fbubuntu flexbuild]$ flex-builder -c eiq

[root@fbubuntu flexbuild]$ flex-builder -i mkbootpartition

[root@fbubuntu flexbuild]$ flex-builder -i merge-component -B eiq

[root@fbubuntu flexbuild]$ flex-builder -i install-eiq

[root@fbubuntu flexbuild]$ flex-builder -i packrfs
```

Deploy full LSDK distro with eIQ images to SD card:

```
[root@fbubuntu flexbuild]$ flex-installer -i pf -d /dev/sdx

[root@fbubuntu flexbuild]$ flex-installer -b build/images/bootpartition_LS_<arch>_lts_<version>.tgz -r
build/images/rootfs_<version>_LS_arm64_main.tgz -d /dev/sdx
```

eIQ components binaries and libraries will be installed in target rootfs folder:

```
/usr/local/bin

/usr/local/lib
```

4 OpenCV getting started guide

OpenCV is an open-source computer vision library. One of its modules (called ML) provides traditional machine learning algorithms. Another important module in the OpenCV is the DNN, which provides support for neural network algorithms.

OpenCV offers a unitary solution for both the neural network inference (DNN module) and the standard machine learning algorithms (ML module). It includes many computer vision functions, making it easier to build complex machine learning applications in a short amount of time and without being dependent on other libraries.

OpenCV has wide adoption in the computer vision field and is supported by a strong and active community. The key algorithms are specifically optimized for various devices and instructions sets. For QorIQ Layerscape processor, OpenCV uses the Arm NEON acceleration. The Arm NEON technology is an advanced SIMD (Single Instruction Multiple Data) architecture extension for the Arm Cortex-A series. The Arm NEON technology is intended to improve multimedia user experience by accelerating the audio and video encoding/decoding, user interface, 2D/3D graphics, or gaming. The Arm NEON can also accelerate the signal-processing algorithms and functions to speed up applications such as the audio and video processing, voice and facial recognition, computer vision, and deep learning.

At its core, the OpenCV DNN module implements an inference engine and does not provide any functionalities for neural network training. For more details about the supported models and layers, see the official OpenCV DNN wiki page.

On the other hand, the OpenCV ML module contains classes and functions for solving machine learning problems such as classification, regression, or clustering. It involves algorithms such as Support Vector Machine (SVM), decision trees, random trees, expectation maximization, k-nearest neighbors, classic Bayes classifier, logistic regression, and boosted trees. For more information, see the official reference manual and machine learning overview. For more details about OpenCV 4.0.1, see the official OpenCV change log web page.

4.1 OpenCV DNN demos

OpenCV DNN demos are installed in this folder: */usr/local/bin/*

However, the input data, model configurations, and model weights are not located in this folder, because of their size. These files must be downloaded to the device before running the demos:

Download the *opencv_extra.zip* package at this link: github.com/opencv/opencv_extra/tree/4.0.1, unpack the file to the directory <*home_dir*>.

```
$ unzip 4.0.1.zip
```

The input images, model configurations for some OpenCV examples are available at this folder:

<home_dir>/opencv_extra-4.0.1/testdata/dnn

The models files can be obtained with two options:

- **Option 1:** Run *opencv_extra* download script to download all opencv_extra dependencies,

  ```
  $ python download_models.py
  ```

  The script downloads the NN models, configuration files, and input images for some OpenCV examples. This operation may take a long time.

- **Option 2:** Direct download required models using wget command

  For example, download caffe model of SqueezeNet,

  ```
  $ wget https://raw.githubusercontent.com/DeepScale/SqueezeNet/
  b5c3f1a23713c8b3fd7b801d229f6b04c64374a5/SqueezeNet_v1.1/squeezenet_v1.1.caffemodel
  ```

  When remote login the device by ssh command, "-X" parameter should be added to enable X11 forwarding to display some examples image result.

  ```
  $ ssh -X <device IP address>
  ```

4.1.1 Image classification example

This demo performs image classification using a pre-trained SqueezeNet network.

Demo dependencies (taken from *<home_dir>/opencv_extra-4.0.1/testdata/dnn*):

- *dog416.png*
- *squeezenet_v1.1.prototxt*
- *squeezenet_v1.1.caffemodel*

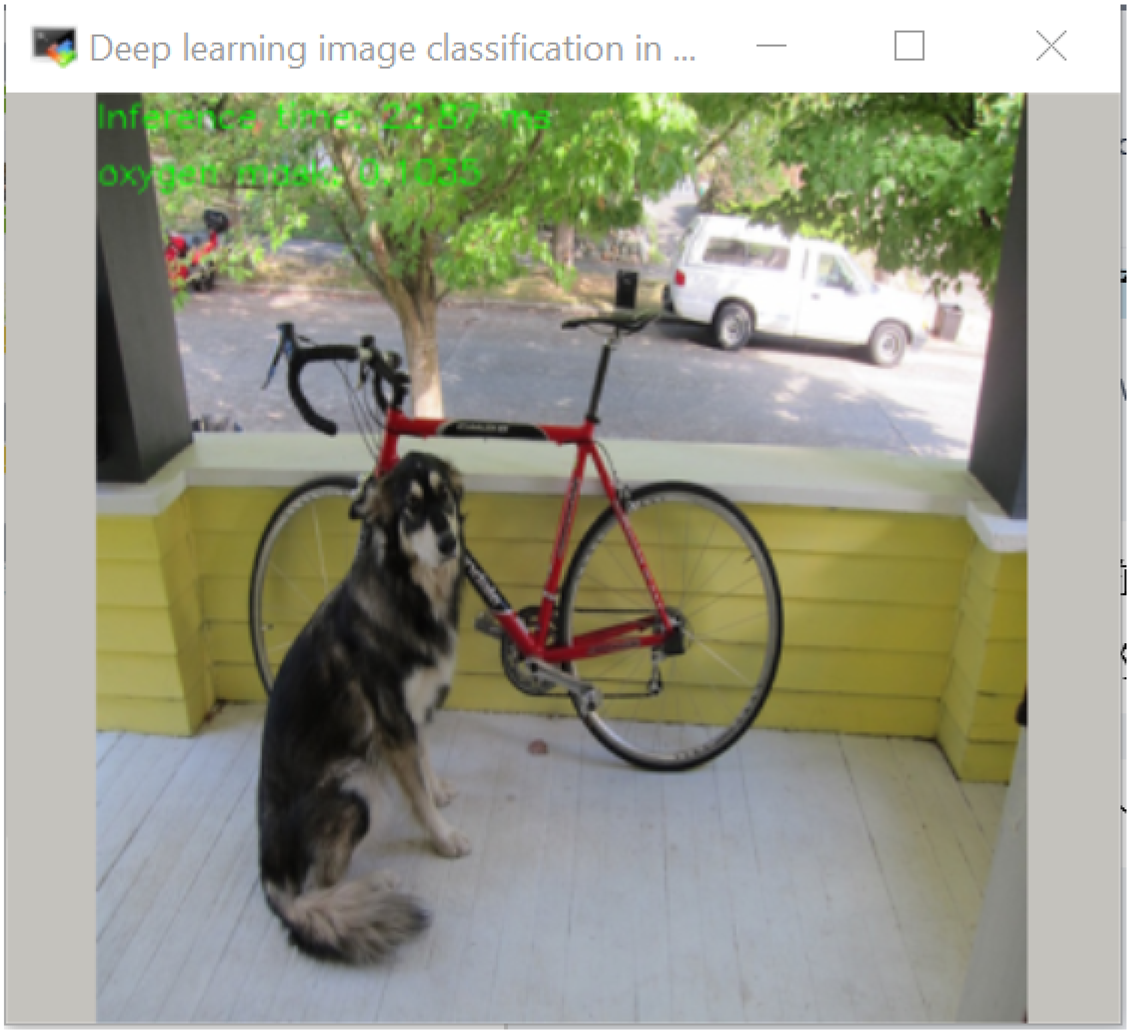
```
$ wget https://raw.githubusercontent.com/DeepScale/SqueezeNet/b5c3f1a23713c8b3fd7b801d229f6b04c64374a5/
SqueezeNet_v1.1/squeezenet_v1.1.caffemodel
```

Other demo dependencies:

- *classification_classes_ILSVRC2012.txt from /usr/local/OpenCV/data/dnn/*
- *models.yml from /usr/local/OpenCV/*

  Running the C++ example with the image input from the default location:

  ```
  $ example_dnn_classification --input=./opencv_extra-4.0.1/testdata/dnn/
  dog416.png --zoo=/usr/local/OpenCV/models.yml --classes=/usr/local/OpenCV/data/dnn/
  classification_classes_ILSVRC2012.txt squeezenet
  ```



```
Inference time: 22.87 ms
oxygen mask: 0.1035
```

Figure 2. Image classification graphics output

4.1.2 Human pose estimation example

This application demonstrates the human or hand pose detection with a pretrained OpenPose DNN. The demo supports only input images, not the live camera input.

Demo dependencies (taken from *<home_dir>/opencv_extra-4.0.1/testdata/dnn*):

- *grace_hopper_227.png*
- *openpose_pose_coco.prototxt*
- *openpose_pose_coco.caffemodel*

```
$ wget http://posefs1.perception.cs.cmu.edu/OpenPose/models/pose/coco/pose_iter_440000.caffemodel -
O openpose_pose_coco.caffemodel
```

Running the C++ example with the image input from the default location:

```
$ example_dnn_openpose --model=openpose_pose_coco.caffemodel --proto=./opencv_extra-4.0.1/testdata/dnn/
openpose_pose_coco.prototxt --image=./opencv_extra-4.0.1/testdata/dnn/grace_hopper_227.png --width=227 --
height=227
```

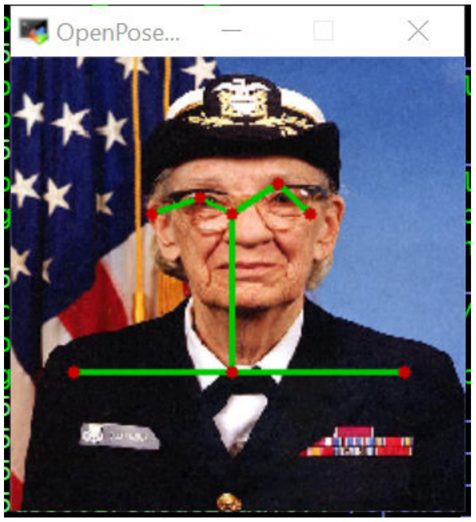


Figure 3. Human pose estimation graphics output

4.1.3 Text detection example

This demo is used for text detection in the image using the EAST algorithm.

Demo dependencies (using wget as option):

- *frozen_east_text_detection.pb*

  Download and unpack the model file,

  ```
  $ wget https://www.dropbox.com/s/r2ingd0l3zt8hxs/frozen_east_text_detection.tar.gz?dl=1 -
  O frozen_east_text_detection.tar.gz
  ```

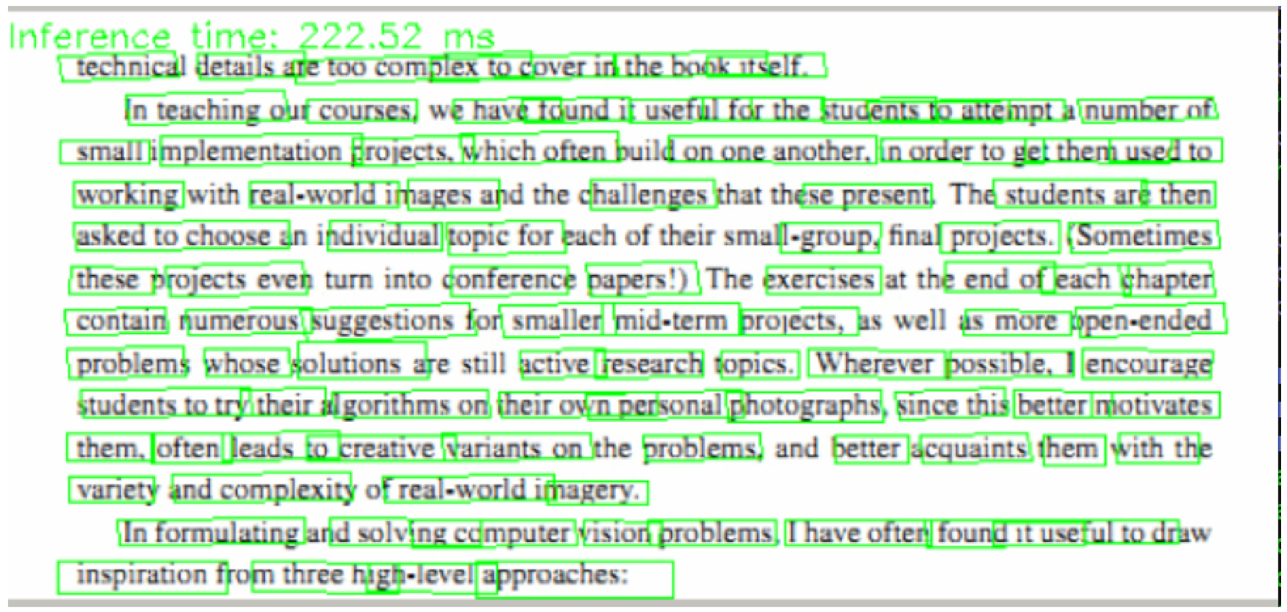
  ```
  $ tar xvf frozen_east_text_detection.tar.gz
  ```

  Other demo dependencies(taken from */usr/local/OpenCV/data/*):

- *imageTextN.png*

  Running the C++ example with the image input from the default location:

  ```
  $ example_dnn_text_detection --model=frozen_east_text_detection.pb --input=/usr/local/OpenCV/
  data/imageTextN.png
  ```



Inference time: 222.52 ms

technical details are too complex to cover in the book itself.

In teaching our courses, we have found it useful for the students to attempt a number of small implementation projects, which often build on one another, in order to get them used to working with real-world images and the challenges that these present. The students are then asked to choose an individual topic for each of their small-group, final projects. (Sometimes these projects even turn into conference papers!) The exercises at the end of each chapter contain numerous suggestions for smaller mid-term projects, as well as more open-ended problems whose solutions are still active research topics. Wherever possible, I encourage students to try their algorithms on their own personal photographs, since this better motivates them, often leads to creative variants on the problems, and better acquaints them with the variety and complexity of real-world imagery.

In formulating and solving computer vision problems, I have often found it useful to draw inspiration from three high-level approaches:

Figure 4. Text detection graphics output

5 Arm Compute Library getting started guide

Arm Compute Library is a collection of low-level functions optimized for Arm CPU and GPU architectures targeted at image processing, computer vision, and machine learning.

5.1 Running AlexNet using graph API

In 2012, AlexNet became famous when it won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), an annual challenge that aims to evaluate algorithms for object detection and image classification. AlexNet is made up of eight trainable layers (five convolution layers and three fully-connected layers). All the trainable layers are followed by the ReLu activation function, except for the last fully-connected layer, where the Softmax function is used.

Follow the below procedure to run this demo:

1. Download the archive file.

   ```
   https://armkeil.blob.core.windows.net/developer/developer/
   technologies/Machine%20learning%20on%20Arm/Tutorials/
   Running%20AlexNet%20on%20Pi%20with%20Compute%20Library/compute_library_alexnet.zip
   ```

   This file contains:

   - The trainable parameters from AlexNet model.
   - A text file, containing the ImageNet labels that are required to map the predicted objects to the name of the classes.
   - Several images in the ppm file format that are ready to be used with the network.

2. Unzip the file to a new folder.

   ```
   $ mkdir assets_alexnet
   ```

   ```
   $ unzip compute_library_alexnet.zip -d assets_alexnet
   ```

3. Run alexnet example.

   ```
   $ graph_alexnet --data=./assets_alexnet/ --image=./assets_alexnet/go_kart.ppm --labels=./
   assets_alexnet/labels.txt
   ```

The output of the successful classification is as follows:

```
graph_alexnet
Threads : 1
Target : NEON
Data type : F32
Data layout : NHWC
Tuner enabled? : false
Cache enabled? : false
Tuner mode : Normal
Tuner file :
Fast math enabled? : false
Data path : ./assets_alexnet/
Image file : ./assets_alexnet/go_kart.ppm
Labels file : ./assets_alexnet/labels.txt
---------- Top 5 predictions ----------
0.9736 - [id = 573], n03444034 go-kart
0.0118 - [id = 518], n03127747 crash helmet
0.0108 - [id = 751], n04037443 racer, race car, racing car
0.0022 - [id = 817], n04285008 sports car, sport car
0.0006 - [id = 670], n03791053 motor scooter, scooter
Test passed
```

6 TensorFlow Lite getting started guide

TensorFlow Lite is a light-weight version of and a next step from TensorFlow. TensorFlow Lite is an open-source software library focused on running machine learning models on mobile and embedded devices (available at www.tensorflow.org/lite). It enables on-device machine learning inference with low latency and small binary size. TensorFlow Lite also supports hardware acceleration using Android™ OS neural network APIs.

6.1 Running benchmark application

LSDK Linux image contains a pre-installed benchmarking application. It performs simple TensorFlow Lite benchmarking using the pre-defined models.

Follow the steps as below to run the benchmark:

1. Download the model file used in this example.

   ```
   $ mkdir tfliteTests
   ```

   ```
   $ cd tfliteTests
   ```

   ```
   $ wget download.tensorflow.org/models/mobilenet_v1_224_android_quant_2017_11_08.zip
   ```

2. Unpack the model file.

   ```
   $ unzip mobilenet_v1_224_android_quant_2017_11_08.zip
   ```

3. Run the example with the command-line arguments from the default location.

   ```
   $ benchmark_model --graph=mobilenet_quant_v1_224.tflite --use_nnapi=true
   ```

   The output of benchmarking application should be similar as below:

   ```
   STARTING!
   Min num runs: [50]
   Min runs duration (seconds): [1]
   Max runs duration (seconds): [150]
   Inter-run delay (seconds): [-1]
   Num threads: [1]
   Benchmark name: []
   ```

```
Output prefix: []
Min warmup runs: [1]
Min warmup runs duration (seconds): [0.5]
Graph: [mobilenet_quant_v1_224.tflite]
Input layers: []
Input shapes: []
Input value ranges: []
Input layer values files: []
Allow fp16 : [0]
Require full delegation : [0]
Enable op profiling: [0]
Max profiling buffer entries: [1024]
CSV File to export profiling data to: []
Max number of delegated partitions : [0]
Loaded model mobilenet_quant_v1_224.tflite
The input model file size (MB): 4.2761
Initialized session in 29.969ms.
Running benchmark for at least 1 iterations and at least 0.5 seconds but terminate if exceeding
150 seconds.
count=6 first=87280 curr=84477 min=84477 max=87280 avg=85015.3 std=1015
Running benchmark for at least 50 iterations and at least 1 seconds but terminate if exceeding
150 seconds.
count=50 first=84593 curr=84484 min=84441 max=85168 avg=84582.6 std=148
Average inference timings in us: Warmup: 85015.3, Init: 29969, Inference: 84582.6
Note: as the benchmark tool itself affects memory footprint, the following is only APPROXIMATE
to the actual memory footprint of the model at runtime. Take the information at your discretion.
Peak memory footprint (MB): init=7.03516 overall=8.96875
```

6.2 Running Python API application

The interpreter-only package is installed in LSDK, meantime LSDK Linux image contains a simple pre-installed example called 'label_image.py' which comes from TensorFlow Lite example code. It classifies images of hundreds of object classes like people, activities, animals, plants and places. The example file location is at:

`/usr/share/tflite/examples`

Follow the below procedure to run this demo:

1. In the example file: `$ cd /usr/share/tflite/examples`

   Edit this line of 'lable_image.py': `import tensorflow as tf`

   to:

   `import tflite_runtime.interpreter as tflite`

   And change this line:

   `interpreter = tf.lite.Interpreter(model_path=args.model_file)`

   to:

   `interpreter = tflite.Interpreter(model_path=args.model_file)`

   For more details please refer to the following guide in TensorFlow: https://www.tensorflow.org/lite/guide/python?hl=en

2. Download the model file used in this example:

   ```
   $ wget https://storage.googleapis.com/download.tensorflow.org/models/mobilenet_v1_2018_08_02/
   mobilenet_v1_1.0_224_quant.tgz
   ```

3. Unpack the model file:

```
$ tar zxvf mobilenet_v1_1.0_224_quant.tgz
```

4. Run the example with the command-line arguments:

```
$ python3 label_image.py --image grace_hopper.bmp --model_file mobilenet_v1_1.0_224_quant.tflite
--label_file labels.txt
```

The output of label_image is as follows:

```
root@localhost:/usr/share/tflite/examples# python3 label_image.py --image grace_hopper.bmp --
model_file mobilenet_v1_1.0_224_quant.tflite --label_file labels.txt
0.874510: military uniform
0.031373: Windsor tie
0.015686: mortarboard
0.011765: bulletproof vest
0.007843: bow tie
```

7 Arm NN getting started guide

Arm NN is an open-source inference engine framework developed by Arm and supporting a wide range of neural-network model formats, such as Caffe, TensorFlow, TensorFlow Lite, and ONNX. For LayerScape CPUs, Arm NN runs on the CPU with NEON and has multi-core support.

7.1 Running Arm NN tests

The Arm NN SDK provides a set of tests, which can also be considered as demos, showing what the Arm NN does and how to use it. They load neural network models of various formats (Caffe, TensorFlow, TensorFlow Lite, ONNX), run the inference on a specified input data, and output the inference result.

Note that the input data, model configurations, and model weights are not distributed with Arm NN. Download them separately and make sure they are available on the device before running the tests. Please also note that the input file names are hardcoded, so you must investigate the code to find out what input file names are expected.

To get started with Arm NN, the following sections explain how to prepare the input data and how to run the Arm NN tests. All of them use well-known neural network models. With only few exceptions, such pre-trained networks are available to download from the Internet. The input image files and their name, format, and content are deduced by analyzing the code. However, this was not possible for all the tests. It is recommended to prepare the data on the host and then deploy them on the Layerscape board, where the current Arm NN tests are run.

The following sections assume that the neural network model files are stored in a folder called models, and the input image files are stored in a folder called data. Both of them are created inside a folder called ArmnnTests. Create this folder structure on the larger partition using the following commands:

```
$ mkdir ArmnnTests

$ cd ArmnnTests

$ mkdir data

$ mkdir models
```

7.2 Caffe tests

The Arm NN SDK provides the following set of tests for the Caffe models:

```
/usr/local/bin/CaffeAlexNet-Armnn

/usr/local/bin/CaffeCifar10AcrossChannels-Armnn
```

```
/usr/local/bin/CaffeInception_BN-Armnn

/usr/local/bin/CaffeMnist-Armnn

/usr/local/bin/CaffeResNet-Armnn

/usr/local/bin/CaffeVGG-Armnn

/usr/local/bin/CaffeYolo-Armnn
```

Two important limitations might require a pre-processing of the Caffe model file before running the Arm NN Caffe test. Firstly, the Arm NN tests require the batch size to be set to 1. Secondly, the Arm NN does not support all Caffe syntaxes, so some previous neural-network model files require updates to the latest Caffe syntax. How to perform these pre-processing steps is described at the Arm NN GitHub page. Note that you should install Caffe on the host.

For example, suppose you have a Caffe model that either has the batch size different than 1 or uses another Caffe defined by files model_name.prototxt and model_name.caffemodel, create a copy of the *.prototxt file (new_model_name.prototxt), modify this file to use the new Caffe syntax, change the batch size to 1, and finally run this Python script:

```
import caffe

new_net = caffe.Net('new_model_name.prototxt', 'model_name.caffemodel', caffe.TEST)

new_net.save('new_model_name.caffemodel')
```

Following is an example about how to run Caffe ArmNN.

7.2.1 CaffeAlexNet-Armnn

To run this test using the folder structure described in the introductory part, perform these steps:

1. Download the caffe model files:

   ```
   $ wget raw.githubusercontent.com/BVLC/caffe/master/models/bvlc_alexnet/deploy.prototxt
   $ wget dl.caffe.berkeleyvision.org/bvlc_alexnet.caffemodel
   ```

2. Transform the network as explained in C:/USERS/NXA20516/DOCUMENTS/INFOSHARE/1813435967/WBI_NXA20516/GUID-D39A0FF3-4B61-4B3A-B6D8-B658108292C7
3. Rename bvlc_alexnet.caffemodel to bvlc_alexnet_1.caffemodel.
4. Copy the bvlc_alexnet_1.caffemodel file to the models folder on the device.
5. Find a *.jpg file that contains a shark. Rename it to shark.jpg and copy it to the data folder on the device.
6. Run the test.

   ```
   $ cd ArmnnTests
   $ CaffeAlexNet-Armnn --data-dir=data --model-dir=models
   ```

The output of a successfully classification is as following:

```
Info: ArmNN v22.0.0

Info: Initialization time: 0.11 ms

Info: Network parsing time: 984.11 ms

Info: Optimization time: 294.33 ms

Info: = Prediction values for test #0

Info: Top(1) prediction is 2 with value: 0.99206

Info: Top(2) prediction is 0 with value: 1.42573e-08

Info: Total time for 1 test cases: 0.245 seconds

Info: Average time per test case: 244.623 ms
```

```
Info: Overall accuracy: 1.000

Info: Shutdown time: 7.53 ms
```

7.3 TensorFlow test

The Arm NN SDK provides the following set of tests for the TensorFlow models:

```
/usr/local/bin/TfCifar10-Armnn

/usr/local/bin/TfInceptionV3-Armnn

/usr/local/bin/TfMnist-Armnn

/usr/local/bin/TfMobileNet-Armnn

/usr/bin/TfResNext-Armnn
```

Following section is an TensorFlow example based on MobileNet.

Before running the tests, the TensorFlow models must be prepared for inference.

7.3.1 TfMobileNet-Armnn

Follow these steps below:

1. From your host machine, download and unpack the model file:

   ```
   $ wget download.tensorflow.org/models/mobilenet_v1_2018_08_02/mobilenet_v1_1.0_224.tgz
   ```

2. Copy the mobilenet_v1_1.0_224_frozen.pb file to the models folder on the device.
3. Find a *.jpg file containing a shark. Rename it to shark.jpg and copy it to the data folder on the device.
4. Find a *.jpg file containing a Labrador dog. Rename it to Dog.jpg and copy it to the data folder on the device.
5. Find a *.jpg file containing a tiger cat. Rename it to Cat.jpg and copy it to the data folder on the device.
6. Run the test.

   ```
   $ cd ArmnnTests
   $ TfMobileNet-Armnn --data-dir=data --model-dir=models
   ```

The output of a successfully classification is as following:

```
Info: ArmNN v22.0.0

Info: Initialization time: 0.12 ms

Info: Network parsing time: 503.82 ms

Info: Optimization time: 21.32 ms

Info: = Prediction values for test #0

Info: Top(1) prediction is 249 with value: 0.86118

Info: Top(2) prediction is 175 with value: 0.00309588

Info: Top(3) prediction is 173 with value: 3.6192e-05

Info: Top(4) prediction is 170 with value: 2.56881e-05

Info: Top(5) prediction is 105 with value: 2.43816e-06

Error: Prediction for test case 0 (249) is incorrect (should be 209)

Info: = Prediction values for test #1

Info: Top(1) prediction is 283 with value: 0.40865
```

```
Info: Top(2) prediction is 282 with value: 0.174808

Info: Top(3) prediction is 264 with value: 0.0136645

Info: Top(4) prediction is 187 with value: 0.00238261

Info: Top(5) prediction is 152 with value: 0.000177185

Info: = Prediction values for test #2

Info: Top(1) prediction is 3 with value: 0.638294

Info: Top(2) prediction is 1 with value: 1.38877e-05

Info: Top(3) prediction is 0 with value: 9.46071e-12

Info: Total time for 3 test cases: 0.537 seconds

Info: Average time per test case: 179.010 ms

Error: One or more test cases failed

Info: Shutdown time: 3.40 ms
```

7.4 TensorFlow Lite test

The Arm NN SDK provides the following set of tests for the TensorFlow models:

/usr/local/bin/TfLiteInceptionV3Quantized-Armnn

/usr/local/bin/TfLiteInceptionV4Quantized-Armnn

/usr/local/bin/TfLiteMnasNet-Armnn

/usr/local/bin/TfLiteMobileNetQuantizedSoftmax-Armnn

/usr/local/bin/TfLiteMobileNetSsd-Armnn

/usr/local/bin/TfLiteMobilenetQuantized-Armnn

/usr/local/bin/TfLiteMobilenetV2Quantized-Armnn

/usr/local/bin/TfLiteResNetV2-50-Quantized-Armnn

/usr/local/bin/TfLiteResNetV2-Armnn

/usr/local/bin/TfLiteVGG16Quantized-Armnn

/usr/local/bin/TfLiteYoloV3Big-Armnn

Following section is an TensorFlow Lite example based on MobileNet.

Before running the tests, the TensorFlow Lite models must be prepared for inference.

7.4.1 TfLiteMobilenetQuantized-Armnn

To run this test using the folder structure described in the introductory part, perform these steps:

1. From your host machine, download and unpack the model file:

   ```
   http://download.tensorflow.org/models/mobilenet_v1_2018_08_02/mobilenet_v1_1.0_224_quant.tgz
   ```

2. Copy the mobilenet_v1_1.0_224_quant.tflite file to the models folder on the device.
3. Find a **.jpg* file containing a shark. Rename it to *shark.jpg* and copy it to the *data* folder on the device.
4. Find a **.jpg* file containing a Labrador dog. Rename it to *Dog.jpg* and copy it to the *data* folder on the device.
5. Find a **.jpg* file containing a tiger cat. Rename it to *Cat.jpg* and copy it to the *data* folder on the device.

6. Run the test.

```
$ cd ArmnnTests
$ TfLiteMobilenetQuantized-Armnn --data-dir=data --model-dir=models
```

The output of a successfully classification is as following:

```
Info: ArmNN v22.0.0

Info: Initialization time: 0.10 ms

Info: Network parsing time: 78.09 ms

Info: Optimization time: 6.51 ms

Info: = Prediction values for test #0

Info: Top(1) prediction is 249 with value: 0.886719

Info: Top(2) prediction is 251 with value: 0.0585938

Info: Top(3) prediction is 250 with value: 0.0429688

Info: Top(4) prediction is 0 with value: 0

Error: Prediction for test case 0 (249) is incorrect (should be 209)

Info: = Prediction values for test #1

Info: Top(1) prediction is 283 with value: 0.402344

Info: Top(2) prediction is 282 with value: 0.246094

Info: Top(3) prediction is 264 with value: 0.0234375

Info: Top(4) prediction is 278 with value: 0.0117188

Info: Top(5) prediction is 288 with value: 0.0078125

Info: = Prediction values for test #2

Info: Top(1) prediction is 3 with value: 0.945312

Info: Top(2) prediction is 4 with value: 0.046875

Info: Top(3) prediction is 0 with value: 0

Info: Total time for 3 test cases: 2.816 seconds

Info: Average time per test case: 938.713 ms

Error: One or more test cases failed

Info: Shutdown time: 0.91 ms
```

7.5 ONNX test

The Arm NN SDK provides the following set of tests for ONNX models:

```
/usr/bin/OnnxMnist-Armnn

/usr/bin/OnnxMobileNet-Armnn
```

The following is an example about how to run ONNX test.

7.5.1 OnnxMnist-Armnn

To run this test using the folder structure described in the introductory part, perform these steps:

1. From your host machine, download and unpack the model file:

```
s3.amazonaws.com/onnx-model-zoo/mobilenet/mobilenetv2-1.0/mobilenetv2-1.0.tar.gz
```

2. Copy the unpacked mobilenetv2-1.0.onnx file to the models folder on the device.
3. Find a *.jpg file containing a shark. Rename it to shark.jpg and copy it to the data folder on the device.
4. Find a *.jpg file containing a Labrador dog. Rename it to Dog.jpg and copy it to the data folder on the device.
5. Find a *.jpg file containing a tiger cat. Rename it to Cat.jpg and copy it to the data folder on the device.
6. Run the test:

```
$ cd ArmnnTests

$ OnnxMobileNet-Armnn --data-dir=data --model-dir=models -i 3
```

The output of a successfully classification is as following:

```
Info: ArmNN v22.0.0

Info: Initialization time: 0.11 ms

Info: Network parsing time: 153.61 ms

Info: Optimization time: 21.24 ms

Info: = Prediction values for test #0

Info: Top(1) prediction is 248 with value: 20.0252

Info: Top(2) prediction is 174 with value: 13.4369

Info: Top(3) prediction is 169 with value: 10.6327

Info: Top(4) prediction is 158 with value: 5.05895

Info: Top(5) prediction is 104 with value: 4.93934

Info: = Prediction values for test #1

Info: Top(1) prediction is 281 with value: 9.54058

Info: Top(2) prediction is 277 with value: 9.47629

Info: Top(3) prediction is 151 with value: 7.58024

Info: Top(4) prediction is 8 with value: 4.882

Info: Top(5) prediction is 1 with value: 3.74129

Info: = Prediction values for test #2

Info: Top(1) prediction is 2 with value: 19.9061

Info: Top(2) prediction is 0 with value: 7.19381

Info: Total time for 3 test cases: 1.148 seconds

Info: Average time per test case: 382.694 ms

Info: Overall accuracy: 0.667

Info: Shutdown time: 2.76 ms
```

7.6 Python interface to Arm NN (PyArmNN)

PyArmNN is a Python extension for Arm NN SDK. PyArmNN provides interface similar to Arm NN C++ API. It is supported only for Python 3.x but not Python 2.x.

7.6.1 Running examples

PyArmNN examples require requests, PIL and maybe some other Python3 modules depending on your examples. You may install the missing modules using pip3.

Follow these steps to run the example:

1. Run the test:

   ```
   $ cd /usr/share/armnn/examples
   ```

   ```
   $ python3 tflite_mobilenetv1_quantized.py
   ```

   The output of a successfully classification is as following:

   ```
   Downloading 'tmp/mobilenet_v1_1.0_224_quant_and_labels.zip' from 'https://storage.googleapis.com/
   download.tensorflow.org/models/tflite/mobilenet_v1_1.0_224_quant_and_labels.zip' ...
   Finished.
   Downloading 'tmp/kitten.jpg' from 'https://s3.amazonaws.com/model-server/inputs/kitten.jpg' ...
   Finished.
   Running inference on 'tmp/kitten.jpg' ...
   class=tabby ; value=99
   class=Egyptian cat ; value=84
   class=tiger cat ; value=71
   class=cricket ; value=0
   class=zebra ; value=0
   ```

8 ONNX Runtime getting started guide

ONNX Runtime is a performance-focused inference engine for ONNX (Open Neural Network Exchange) models.

Models in the TensorFlow, Keras, PyTorch, scikit-learn, CoreML, and other popular supported formats can be converted to the standard ONNX format, providing framework interoperability and helping to maximize the reach of hardware optimization investments. This provides a solution for systems to integrate a single inference engine to support models trained from a variety of frameworks, while taking advantage of specific hardware accelerators where available.

ONNX Runtime was designed with a focus on performance and scalability in order to support heavy workloads in high-scale production scenarios. It also has extensibility options for compatibility with emerging hardware developments.

8.1 Running standard tests

Follow the steps to run onnx_test_runner:

1. Download onnx model zoo model.

   ```
   s3.amazonaws.com/onnx-model-zoo/mobilenet/mobilenetv2-1.0/mobilenetv2-1.0.tar.gz
   ```

   This file contains:

   - The mobilenetv2-1.0 onnx model.
   - The test data sets 0/1/2, containing the input and output data.

2. Unzip the file to a new folder.

   ```
   $ mkdir onnxruntimeTests
   $ cd onnxruntimeTests
   $ tar xzvf mobilenetv2-1.0.tar.gz
   ```

3. Run example:

   ```
   $ onnx_test_runner mobilenetv2-1.0
   result:
   Models: 1
   ```

```
Total test cases: 3
Succeeded: 3
Not implemented: 0
Failed: 0
Stats by Operator type:
Not implemented(0):
Failed:
Failed Test Cases:
```

9 PyTorch getting started guide

PyTorch is a scientific computing package based on Python that facilitates building deep learning projects using power of graphics processing units. It emphasizes flexibility and speed which provides two high-level features.

9.1 Running image classification example

There is an example located in the examples folder, which requires urllib, PIL and maybe some other Python3 modules depending on your image. You may install the missing modules using pip3.

```
$ cd /usr/share/pytorch/examples/
```

To run the example with inference computation on the CPU, use the following command. There are no arguments and the resources will be downloaded automatically by the script.

```
$ python3 pytorch_mobilenetv2.py
```

The output should look similar as the following:

```
File does not exist, download it from

https://download.pytorch.org/models/mobilenet_v2-b0353104.pth

... 100.00%, downloaded size: 13.55 MB

File does not exist, download it from

https://raw.githubusercontent.com/Lasagne/Recipes/master/examples/resnet50/imagenet_classes.txt

... 100.00%, downloaded size: 0.02 MB

File does not exist, download it from

https://s3.amazonaws.com/model-server/inputs/kitten.jpg

... 100.00%, downloaded size: 0.11 MB

('tabby, tabby cat', 46.34805679321289)

('Egyptian cat', 15.802854537963867)

('lynx, catamount', 1.1611212491989136)

('lynx, catamount', 1.1611212491989136)

('tiger, Panthera tigris', 0.20774540305137634)
```

A Revision history

The table below summarizes the revisions to this document.

Table 1. Revision history

Revision	Date	Topic cross-reference	Change description
Rev. 3	12/2020	NXP eIQ software introduction	Updated eIQ software feature list. Updated the figure Figure 1.
		eIQ installation guide	Removed the section.
		Building eIQ Components	Added new section.
		OpenCV DNN demos	Updated the section.
		Image classification example	Updated the section.
		Human pose estimation example	Updated the section.
		Text detection example	Updated the section.
		Arm Compute Library getting started guide	Updated the section.
		Running AlexNet using graph API	Updated the section.
		TensorFlow	Removed the section.
		TensorFlow Lite getting started guide	Updated the section.
		Arm NN getting started guide	Updated the section.
		ONNX Runtime getting started guide	Updated the section.
		PyTorch getting started guide	Updated the section.
Rev. 2	03/2020	NXP eIQ software introduction	Updated the section and added the feature Onnxruntime 0.4.0. Updated the figure Figure 1
		ONNX Runtime getting started guide	Added the chapter.
		Installing eIQ software	Updated the section.
		OpenCV DNN demos	Updated the section including Image classification example, Human pose estimation example, Text detection example
		Running image classification example	Updated this section in TensorFlow Lite getting started guide
		Running benchmark application	Updated this section in TensorFlow Lite getting started guide.
Rev. 1	09/2019	Installing eIQ software	Updated the file path where eIQ software installation script is located.

Table continues on the next page...

Table 1. Revision history (continued)

Revision	Date	Topic cross-reference	Change description
Rev. 0	09/2019	-	Initial public release.

Date of release: 12/2020

Document identifier: AN12580