NXP Semiconductors
User Manual

Document Number: UM11226
Rev. 4, 05/2020

NXP eIQ™ Machine Learning Software Development Environment for i.MX Applications Processors

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 outputs 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 i.MX applications processors, a set of ML tools which allows developing and deploying ML applications on the i.MX 8 family of devices.

Contents

This document provides guidance for the supported ML software for the i.MX family. The document is divided into separate sections, starting with the NXP eIQ introduction, the Yocto installation guide, and the step-by step guide for running all supported DNN and non-DNN examples.

NOTE

This document describes the eIQ Machine Learning Software for the NXP L4.14 BSP release. Beginning with the L4.19 BSP, the eIQ software is pre-integrated in the BSP release and this document is no longer necessary or being maintained. For more information on the eIQ software in these releases (L4.19, L5.4, and so on), see the “NXP eIQ Machine Learning” chapter in the Linux user’s guide for that specific release. Be sure to join the eIQ Machine Learning Software Community (https://community.nxp.com/community/eiq), where you will find many new demos and sample applications (in addition to great community support).

2. NXP eIQ software introduction

The NXP eIQ machine learning software development environment provides a set of libraries and development tools for machine learning applications targeted at NXP MCUs and application 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, TensorFlow Lite, Arm® NN, and Arm Compute Library.

The current version of NXP eIQ software of i.MX processors delivers advanced and highly optimized machine learning enablement by providing ML support in Linux OS BSPs for the i.MX 8 family of devices. The NXP eIQ software contains these main Yocto recipes:

- OpenCV 4.0.1
- Arm Compute Library 19.02
- Arm NN 19.02
- ONNX runtime 0.3.0
- TensorFlow 1.12
- TensorFlow Lite 1.12

For more details about the i.MX 8 family of application processors, see the fact sheet [1].

For up-to-date information about NXP machine learning solutions, see the official NXP web page [2] for machine learning and artificial intelligence.

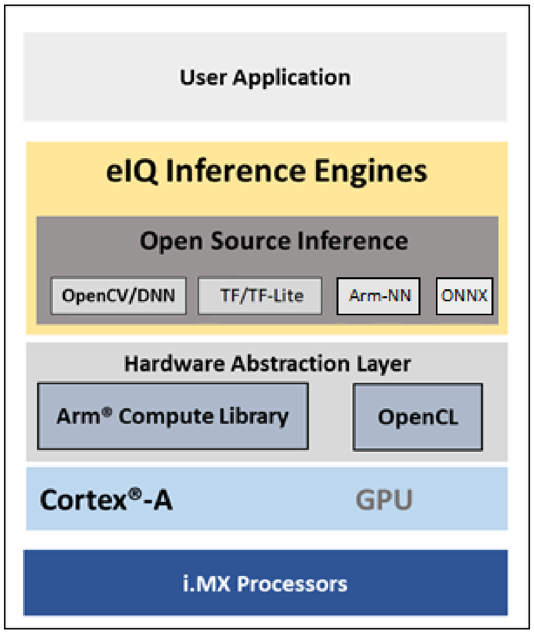


Figure 1. NXP eIQ machine learning software

3. Yocto installation guide

This chapter provides a step-by-step guide for configuring and building Linux L4.14.98 GA, the Linux Yocto BSP release for i.MX 8 family of devices [3], with support for NXP eIQ software.

NOTE

This document describes the eIQ Machine Learning Software for the NXP L4.14 BSP release. Beginning with the L4.19 BSP, the eIQ software is pre-integrated in the BSP release and this document is no longer necessary or being maintained. For more information on the eIQ software in these releases (L4.19, L5.4, and so on), see the "NXP eIQ Machine Learning" chapter in the Linux user's guide for that specific release. Be sure to join the eIQ Machine Learning Software Community (https://community.nxp.com/community/eiq), where you will find many new demos and sample applications (in addition to great community support).

To enable NXP eIQ machine learning software, the main configuration changes are:

- Mandatory: select the right machine learning manifest file (*.*xml*) – see Section 3.2.2, "Yocto project metadata downloading".
- Optional: modify the machine learning configuration file (*.*conf*) or layer files (*.*bb*), depending on which special configuration is needed; see Section 3.2.4, "Yocto configuration file modifying" or Section 3.2.5, "OpenCV user build modification".

For more information about the Linux Yocto BSP setup, see the Linux L4.14.98_2.0.0 documentation [4].

3.1. Prerequisites

3.1.1. Hardware requirements

- 1 x Linux OS host machine with a minimum of 120 GB HDD space available and internet connection
- 1 x MCIMX8QM-CPU board with internet connection
- 1 x SDHC card (tested with a 16-GB SDHC Class 10 UHS-I card)
- 1 x MIPI camera MCIMXCAMERA1MP with de-serializer MX8XMIPI4CAM2 for running OpenCV DNN examples using the live camera inputs (optional only)
- LCD HDMI monitor

3.1.2. Software requirements

1. Host OS: Ubuntu (tested with 16.04)
2. Host packages:
   - The essential Yocto project host packages are:

```
$: sudo apt-get install gawk wget git-core diffstat unzip texinfo \
   gcc-multilib build-essential chrpath socat libsdl1.2-dev
```

   - The i.MX layers host packages for the Ubuntu OS host setup are:

```
$: sudo apt-get install libsdl1.2-dev xterm sed cvs subversion \
   coreutils texi2html docbook-utils python-pysqlite2 help2man gcc \
   g++ make desktop-file-utils libgl1-mesa-dev libglu1-mesa-dev \
   mercurial autoconf automake groff curl lzop asciidoc u-boot-tools
```

3.2. Building NXP eIQ software support using Yocto Project tools

See the *i.MX Yocto Project User's Guide* document [4] or sections 3.2.1 to 3.2.6, and 3.2.9. See the *i.MX Linux User's Guide document* [4] or sections 3.2.7 to 3.2.8.

3.2.1. Repo utility installing

This must be done only once.

```
$: mkdir ~/bin
$: curl https://storage.googleapis.com/git-repo-downloads/repo > ~/bin/repo
$: chmod a+x ~/bin/repo
$: PATH=${PATH}:~/bin
```

3.2.2. Yocto project metadata downloading

```
$: mkdir fsl-arm-yocto-bsp
$: cd fsl-arm-yocto-bsp
$: repo init -u https://source.codeaurora.org/external/imx/imx-manifest -b imx-linux-sumo -m
imx-4.14.98-2.0.0_machinelearning.xml
$: repo sync
```

NOTE

The *imx-4.14.78-1.0.0_machinelearning* manifest file can be also used.

3.2.3. Yocto build setup

```
$: EULA=1 MACHINE=imx8qmmek DISTRO=fsl-imx-xwayland source ./fsl-setup-release.sh -b build-
xwayland
$: echo "BBLAYERS += \" \${BSPDIR}/sources/meta-imx-machinelearning \"" >> conf/bblayers.conf
```

3.2.4. Yocto configuration file modifying

OpenCV 4.0.1 is available to be built and is already installed in the suggested image. Therefore, the *local.conf* file does not have to be modified to include the OpenCV in the Yocto image. However, it is recommended to add some extra packages to this configuration file for a more convenient image. The *local.conf* file is in folder *fsl-arm-yocto-bsp/build-xwayland/conf*.

Add basic development capabilities:

```
EXTRA_IMAGE_FEATURES = " dev-pkgs debug-tweaks tools-debug tools-sdk ssh-server-openssh"
```

Add packages for networking capabilities:

```
IMAGE_INSTALL_append = " net-tools iputils dhcpcd"
```

Add some generic tools:

```
IMAGE_INSTALL_append = " which gzip python python-pip"
IMAGE_INSTALL_append = " wget cmake gtest git zlib patchelf"
IMAGE_INSTALL_append = " nano grep vim tmux swig tar unzip"
IMAGE_INSTALL_append = " parted e2fsprogs e2fsprogs-resize2fs"
```

Configure the OpenCV package:

```
IMAGE_INSTALL_append = " opencv python-opencv"
PACKAGECONFIG_remove_pn-opencv_mx8 = "python3"
PACKAGECONFIG_append_pn-opencv_mx8 = " dnn python2 qt5 jasper openmp test neon"
```

Remove the OpenCL support from packages:

```
PACKAGECONFIG_remove_pn-opencv_mx8 = "opencl"
PACKAGECONFIG_remove_pn-arm-compute-library = "opencl"
```

Add CMake for SDK's cross-compile:

```
TOOLCHAIN_HOST_TASK_append = " nativesdk-cmake nativesdk-make"
```

Add packages:

```
IMAGE_INSTALL_append = " arm-compute-library tensorflow tensorflow-lite armnn onnxruntime"
PREFERRED_VERSION_opencv = "4.0.1%"
PREFERRED_VERSION_tensorflow = "1.12.0%"
PREFERRED_VERSION_tensorflow-lite = "1.12.0%"
```

NOTE

OpenCL is currently not supported in the L4.14.98_2.0.0 and L4.14.78_1.0.0 Yocto configurations.

3.2.5. OpenCV user build modification

The OpenCV 4.0.1 is installed with all necessary DNN and ML dependencies in the NXP eIQ software. If some special OpenCV build options are required, add them to the OpenCV recipe file to their separate PACKAGECONFIG section. The *opencv_4.0.1-imx.bb* file is located on the Linux OS host PC in this folder:

fsl-arm-yocto-bsp/sources/meta-imx-machinelearning/recipes-graphics/opencv

3.2.6. Image building

The image should be built with Qt 5 support, because some OpenCV examples requires Qt 5 to be enabled in the image:

```
$: bitbake fsl-image-qt5
```

3.2.7. SD card image flashing

The result of the build process is a compressed image which can be found in *tmp/deploy/images/imx8qmmek/fsl-image-qt5-imx8qmmek-<timestamp>.rootfs.sdcard.bz2*, where <*timestamp*> is the image timestamp (for example: 20180509080732).

Decompress the image before flashing it to the SD card:

```
bunzip2 -k -f tmp/deploy/images/imx8qmmek/fsl-image-qt5-imx8qmmek-
<timestamp>.rootfs.sdcard.bz2
```

Flash the SD card (replace “sdX” with the actual SD card device):

```
dd if= tmp/deploy/images/imx8qmmek/fsl-image-qt5-imx8qmmek-<timestamp>.rootfs.sdcard
of=/dev/sdX bs=1M && sync
```

NOTE

The Win32DiskImager utility can be also used for the SD card image flashing.

3.2.8. SD card disk space extending

The ML applications require a lot of disk space to store the input model data. By default, the SD card image is created with a small amount of extra space (approximately 500 MB) in the rootfs, which may not be enough for all ML applications.

There are two methods how to extend the SD card free space:

1. Define additional free disk space before start the building process. It is done using the IMAGE_ROOTFS_EXTRA_SPACE variable in the *local.conf* file. This step is also described in the Yocto project manual here: https://www.yoctoproject.org/docs/current/mega-manual/mega-manual.html#var-IMAGE_ROOTFS_EXTRA_SPACE.
2. Extend the SD card disk space after the image building. This ex-post method is described in more detail in the below section.

Print all SD card partitions of the target board:

```
$: fdisk -l
Device          Boot  Start      End  Sectors Size Id Type
/dev/mmcblk1p1        16384   147455   131072  64M  c W95 FAT32 (LBA)
/dev/mmcblk1p2       147456 10584063 10436608   5G 83 Linux
```

Start the “fdisk” utility:

```
$: fdisk /dev/mmcblk1
```

Delete the Linux-type partition (second in this case):

```
Command (m for help): d
Partition number (1,2, default 2): 2
Partition 2 has been deleted.
```

Create the new primary partition (second in this case) with the first sector being identical to the original partition:

```
Command (m for help): n
Partition type
   p   primary (1 primary, 0 extended, 3 free)
   e   extended (container for logical partitions)
Select (default p):

Using default response p.
Partition number (2-4, default 2):
First sector (2048-31116287, default 2048): 147456
Last sector, +sectors or +size{K,M,G,T,P} (147456-31116287, default 31116287):
```

Write the new partition and exit the “fdisk” utility:

```
Command (m for help): w

The partition table has been altered.
Syncing disks.
```

Increase the filesystem size of the second partition:

```
$: resize2fs /dev/mmcblk1p2
resize2fs 1.43.8 (1-Jan-2018)
Filesystem at /dev/mmcblk1p2 is mounted on /; on-line resizing required
old_desc_blocks = 1, new_desc_blocks = 1
The filesystem on /dev/mmcblk1p2 is now 3871104 (4k) blocks long.
```

NOTE

You can also use the “parted” Linux OS command to create a new partition instead of using the “fdisk” command.

Finally, check the free disk space after resizing:

```
$: df -h
```

3.2.9. Generating the Toolchain

The toolchain created by the Yocto Project tools provides a set of tools (compilers, libraries, and header files) to cross-compile the code for the previously-built images. Build the SDK with the Qt 5 support:

```
$: bitbake fsl-image-qt5 -c populate_sdk
```

After the build process finishes, it produces an installer script that can be used to install the SDK on the developing system. The script is created in the *tmp/deploy/sdk/fsl-imx-xwayland-glibc-x86_64-fsl-image-qt5-aarch64-toolchain-4.14-sumo.sh*.

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 i.MX, 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 [5].

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 [6].

4.1. OpenCV DNN demos

After creating a bootable SD card using Yocto, all OpenCV DNN demos are in the */usr/share/OpenCV/samples/bin/* folder (the default demo location). 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 using `unzip opencv_extra-4.0.1.zip` to the SD card root directory <*home_dir*>.
- Go to the <*home_dir*>*/opencv_extra-4.0.1/testdata/dnn/* folder and run `python download_models.py`. The script downloads the NN models, configuration files, and input images

for some OpenCV examples. This operation may take a while. Copy these dependencies to the */usr/share/OpenCV/samples/bin* folder (see also the demo dependencies parts of sections 4.1.x in this document).

- Download the configuration model file at this link: github.com/opencv/opencv/blob/master/samples/dnn/models.yml

  The *model.yml* file contains the pre-processing parameters for some DNN examples, which accept the "–zoo" parameter. Copy the model file to the */usr/share/OpenCV/samples/bin* folder.

4.1.1. Image classification example

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

Demo dependencies (taken from the "opencv_extra" package):

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

Other demo dependencies:

- *classification_classes_ILSVRC2012.txt* from */usr/share/OpenCV/samples/data/dnn*
- *models.yml* from github

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

```
$: ./example_dnn_classification --input=dog416.png --zoo=models.yml squeezenet
```

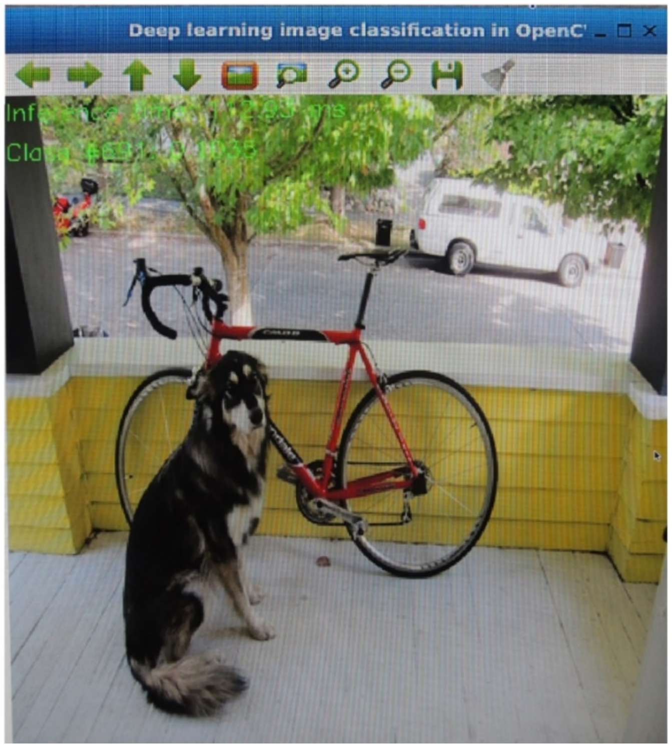


Figure 2. Image classification graphics output

Running the C++ example with the live camera input from the default location:

```
$: ./example_dnn_classification --zoo=models.yml squeezenet
```

4.1.2. YOLO object detection example

This demo performs the object detection using the You Only Look Once (YOLO) detector (arxiv.org/abs/1612.08242). It detects objects in a camera/video/image.

For more information about this demo, see the "Loading Caffe framework models" OpenCV tutorial: docs.opencv.org/4.0.1/da/d9d/tutorial_dnn_yolo.html.

Demo dependencies (taken from the "opencv_extra" package):

- *dog416.png*
- *yolov3.weights*
- *yolov3.cfg*

Other demo dependencies:

- *models.yml*
- *object_detection_classes_yolov3.txt* from */usr/share/OpenCV/samples/data/dnn*

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

```
$: ./example_dnn_object_detection -width=1024 -height=1024 -scale=0.00392 -input=dog416.png -
rgb -zoo=models.yml yolo
```

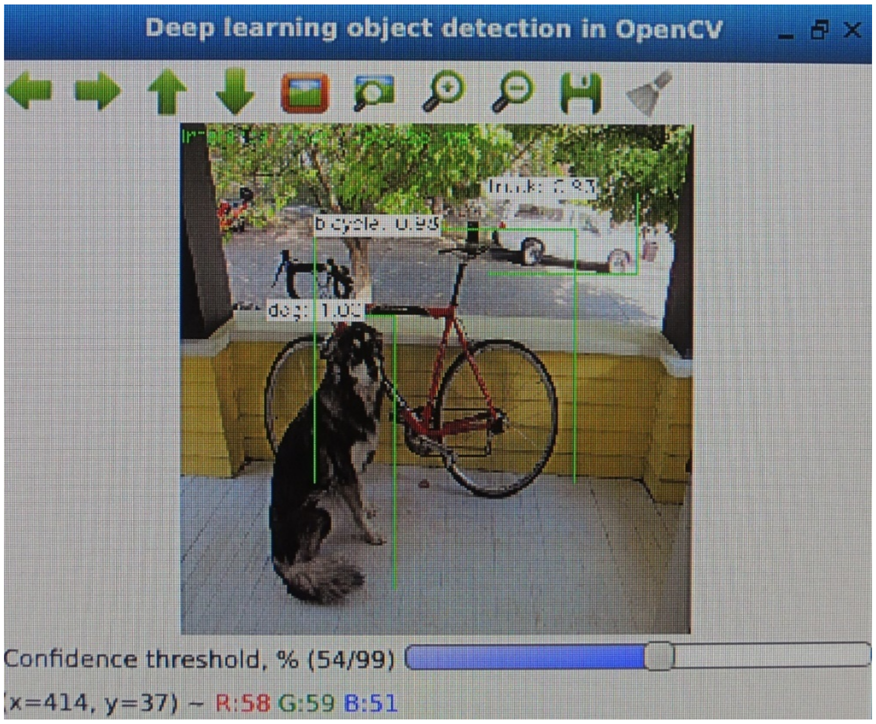

Deep learning object detection in OpenCV
truck: 0.93
bicycle: 0.99
dog: 1.00
Confidence threshold, % (54/99)
(x=414, y=37) ~ R:58 G:59 B:51

Figure 3. YOLO object detection graphics output

Running the C++ example with the live camera input from the default location:

```
$: ./example_dnn_object_detection -width=1024 -height=1024 -scale=0.00392 -rgb -
zoo=models.yml yolo
```

NOTE

Running this example with the live camera input is very slow, because this example runs only on the CPU.

4.1.3. Image segmentation example

The image segmentation means dividing the image into groups of pixels based on some criteria. You can do this grouping based on color, texture, or some other criteria that you choose.

Demo dependencies (taken from the “opencv_extra” package):

- *dog416.png*
- *fcn8s-heavy-pascal.caffemodel*
- *fcn8s-heavy-pascal.prototxt*

Other demo dependencies:

- *models.yml*

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

```
$: ./example_dnn_segmentation --width=500 --height=500 --rgb --mean=1 --input=dog416.png --
zoo=models.yml fcn8s
```

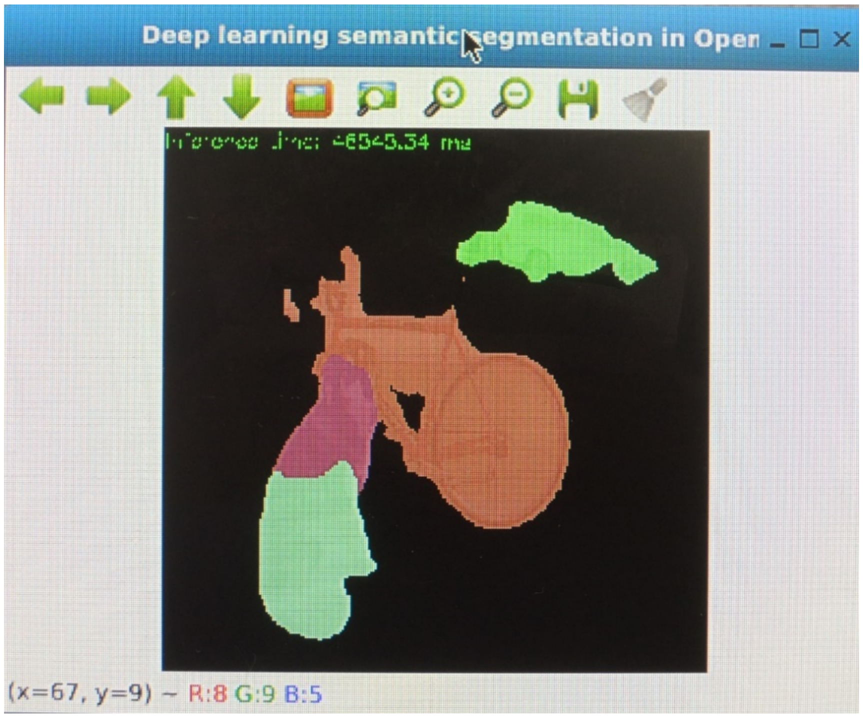

Deep learning semantic segmentation in Open
Inference time: 46545.34 ms
(x=67, y=9) ~ R:8 G:9 B:5

Figure 4. Image segmentation graphics output

Running the C++ example with the live camera input from the default location:

```
$: ./example_dnn_segmentation --width=500 --height=500 --rgb --mean=1 --zoo=models.yml fcn8s
```

NOTE

Running this example with the live camera input is very slow, because this example runs only on the CPU.

4.1.4. Image colorization example

This example demonstrates the recoloring of grayscale images using DNN. The demo supports input images only, not the live camera input.

Demo dependencies (taken from the “opencv_extra” package):

- *colorization_release_v2.caffemodel*
- *colorization_deploy_v2.prototxt*

Other demo dependencies:

- *basketball1.png*

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

```
$: ./example_dnn_colorization --model=colorization_release_v2.caffemodel --
proto=colorization_deploy_v2.prototxt --image=../data/basketball1.png
```

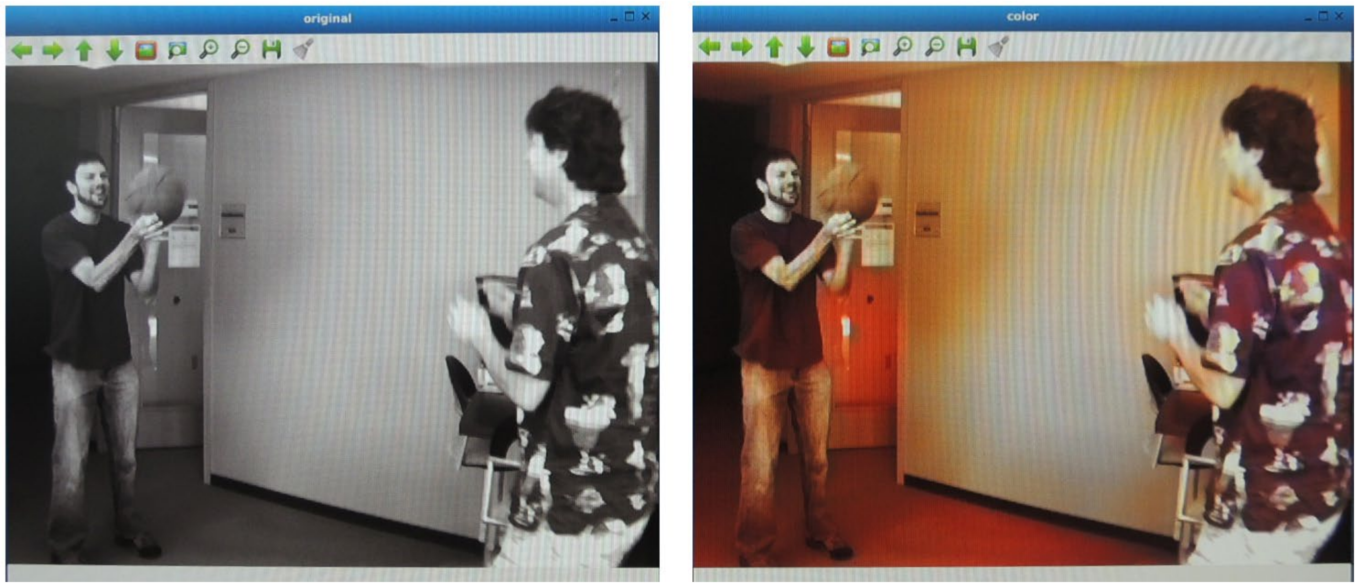


Figure 5. Image colorization demo graphics output

4.1.5. 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 the “opencv_extra” package):

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

- *openpose_pose_coco.prototxt*

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

```
$: ./example_dnn_openpose --model=openpose_pose_coco.caffemodel --
proto=openpose_pose_coco.prototxt --image=grace_hopper_227.png --width=227 --height=227
```

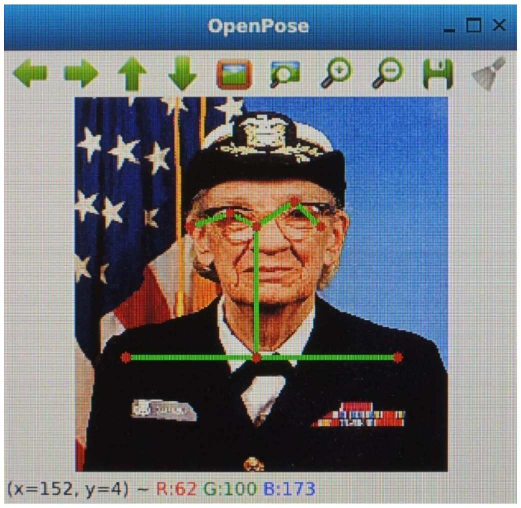


Figure 6. Human pose estimation graphics output

4.1.6. Object detection example

This demo performs object detection using SqueezeDet. The demo supports only input images, not the live camera input.

Demo dependencies:

- Download the model definition and model weight files from:
  github.com/kvmanohar22/caffe/tree/obj_detect_loss/proto
- *SqueezeDet.caffemodel*
- *SqueezeDet_deploy.prototxt*
- Download the input image from:
  github.com/opencv/opencv_contrib/blob/4.0.1/modules/dnn_objdetect/tutorials/images/aeroplane.jpg

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

```
$: ./example_dnn_objdetect_obj_detect SqueezeDet_deploy.prototxt SqueezeDet.caffemodel
aeroplane.jpg
```

Running the model on the *aeroplane.jpg* image produces the following text results in the console:

```
------
Class: aeroplane
Probability: 0.845181
Co-ordinates: 41 116 415 254
------
```

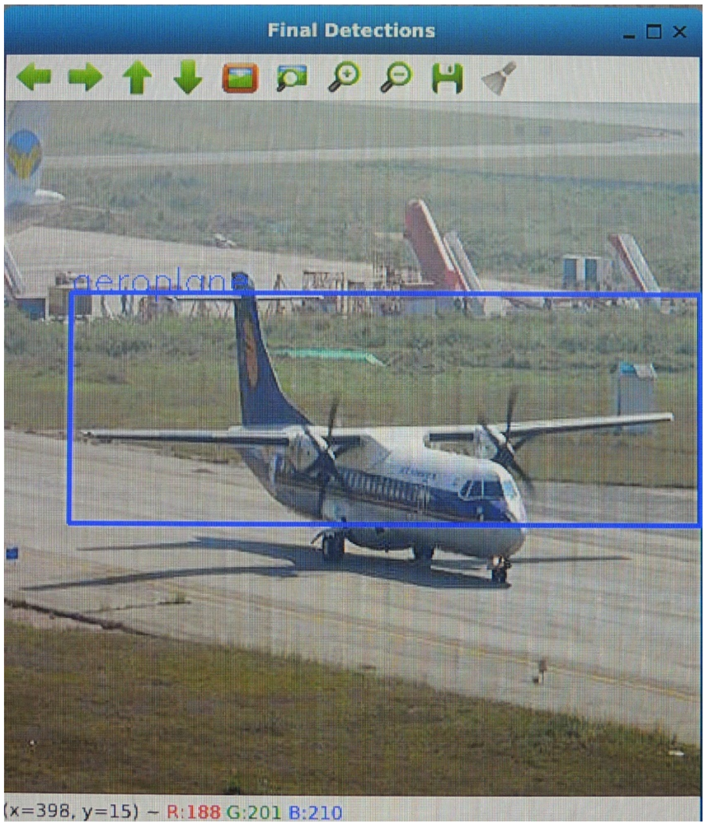


Figure 7. Object detection graphics output

4.1.7. CNN image classification example

This demo performs image classification using a pre-trained SqueezeNet network. The demo supports only input images, not the live camera input.

Demo dependencies (taken from the "opencv_extra" package):

- *space_shuttle.jpg*

Other demo dependencies:

- Download the *SqueezeNet.caffemodel* model weight file from:
  github.com/kvmanohar22/caffe/tree/obj_detect_loss/proto
- Download the *SqueezeNet_deploy.prototxt* model definition file from:
  github.com/opencv/opencv_contrib/tree/4.0.1/modules/dnn_objdetect/samples/data

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

```
$: ./example_dnn_objdetect_image_classification SqueezeNet_deploy.prototxt
SqueezeNet.caffemodel space_shuttle.jpg
```

Running the model on the *space_shuttle.jpg* image produces the following text results in the console:

```
Best class Index: 812
Time taken: 0.649153
Probability: 15.8467
```

4.1.8. Text detection example

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

Demo dependencies (taken from the *opencv_extra* package):

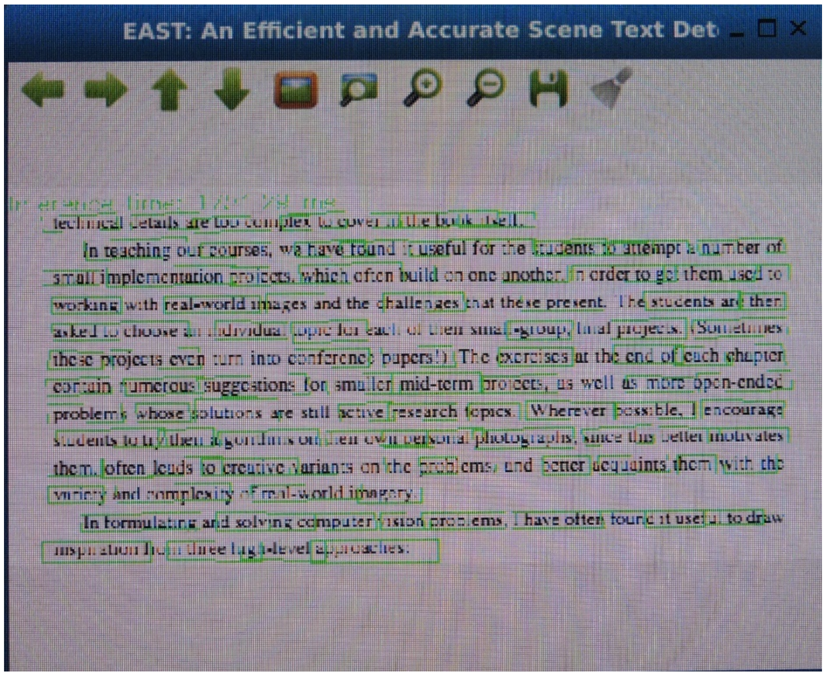
- *frozen_east_text_detection.pb*

Other demo dependencies:

- *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=../data/imageTextN.png
```



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 8. Text detection graphics output

NOTE

This example accepts only the PNG image format.

Running the C++ example with the live camera input from the default location:

```
$: ./example_dnn_text_detection --model=frozen_east_text_detection.pb
```

4.2. OpenCV standard machine learning demos

After deploying OpenCV on the target device, the non-neural-network demos are installed in the “rootfs” in the */usr/share/OpenCV/samples/bin/* folder. To display the results, a Yocto image with Qt 5 support is required.

4.2.1. Introduction to SVM

This example demonstrates how to create and train an SVM model using training data. When the model is trained, the labels for test data are predicted. The full description of the example is in tutorial_introduction_to_svm.

After running the demo, the graphics result is shown on the screen (Qt 5 support is required):

```
$: ./example_tutorial_introduction_to_svm
```

Result:

1. The code opens an image and shows the training examples of both classes. The points of one class are represented with white circles and the other class uses black points.

2. The SVM is trained and used to classify all the pixels of the image. This results in the division of image into blue and green regions. The boundary between both regions is the optimal separating hyperplane.
3. Finally, the support vectors are shown using gray rings around the training examples.

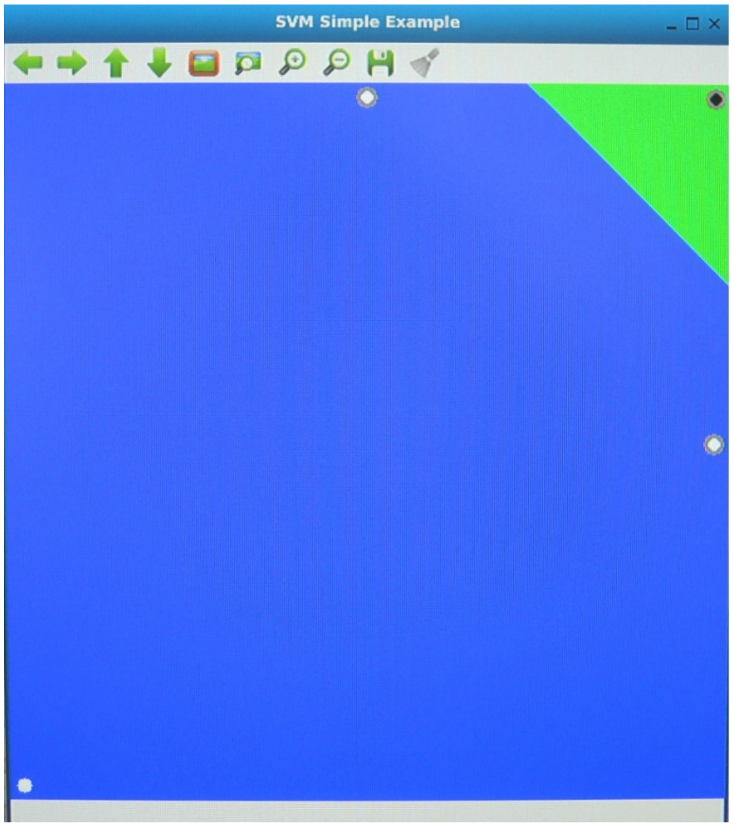


Figure 9. SVM introduction graphics output

4.2.2. SVM for non-linearly separable data

This example deals with non-linearly-separable data and shows how to set the parameters of the SVM with linear kernel for these data. For more details, see SVM_non_linearly_separable_data.

After running the demo, the graphics result is shown on the screen (Qt 5 support is required):

```
$: ./example_tutorial_non_linear_svms
```

Result:

1. The code opens an image and shows the training data of both classes. The points of one class are represented by a light-green color and the other class is shown as light-blue points.
2. The SVM is trained and used to classify all pixels of the image. This divides the image into blue and green regions. The boundary between both regions is the separating hyperplane. Because the training data is non-linearly separable, some examples of both classes are misclassified; some green points lay in the blue region and some blue points lay in the green one.
3. The support vectors are shown with gray rings around the training examples.

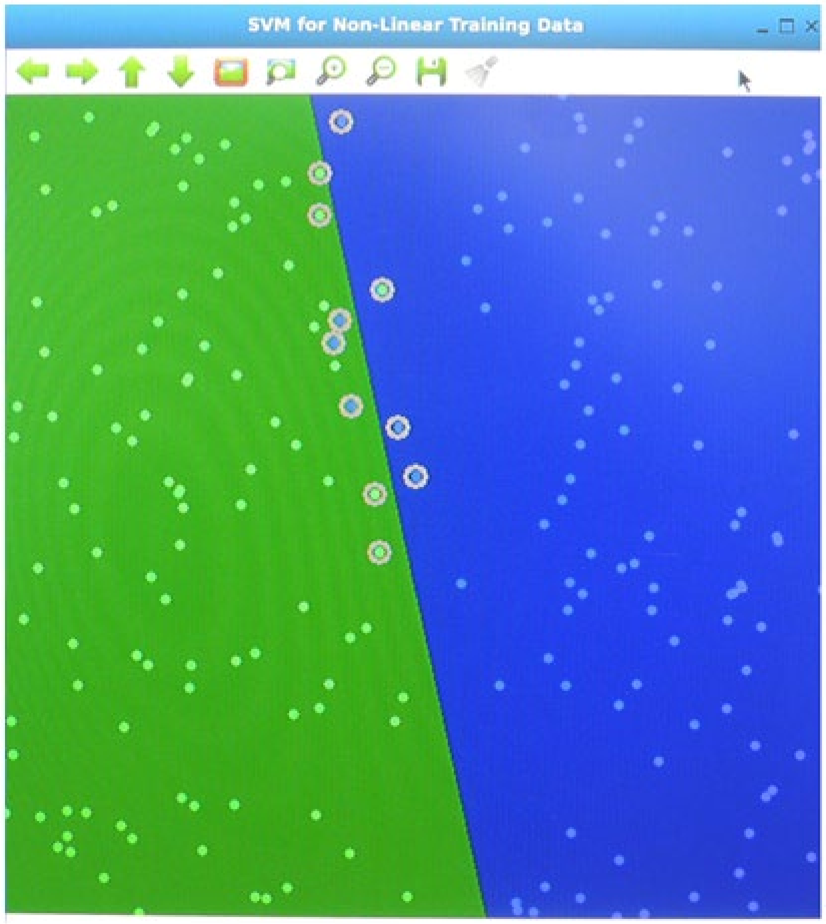


Figure 10. SVM non-linearity graphics output

4.2.3. Introduction to PCA

The Principal Component Analysis (PCA) is a statistical method that extracts the most important features of a dataset. In this tutorial, it is shown how to use the PCA to calculate the orientation of an object. For more details, see the OpenCV tutorial: Introduction_to_PCA.

After running the demo, the graphics result is shown on the screen (Qt 5 support is required):

```
$: ./example_tutorial_introduction_to_pca
```

Result:

- The code opens an image (loaded from *../data/pca_test1.jpg*), finds the orientation of the detected objects of interest, and visualizes the result by drawing the contours of the detected objects of interest, the center point, and the x-axis and y-axis regarding the extracted orientation.

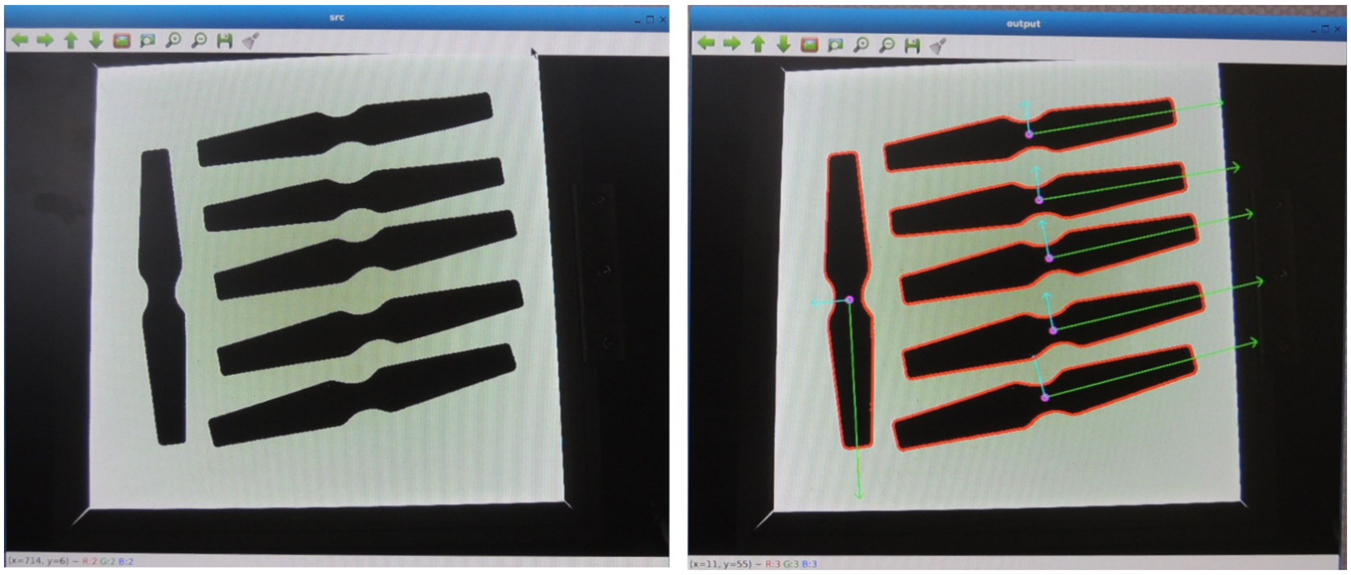


Figure 11. PCA graphics output

4.2.4. Logistic regression

In this example, logistic regression is used to predict two characters (0 or 1) from an image. Every image matrix is reshaped from its original size of 28 x 28 to 1 x 784. A logistic regression model is created and trained on 20 images. After the training, the model can predict the labels of test images. The source code is at this link and can be run using the below command.

Demo dependencies (preparing the train data files):

```
$: wget raw.githubusercontent.com/opencv/opencv/4.0.1/samples/data/data01.xml
```

After running the demo, the graphics result is shown on the screen (Qt 5 support is required):

```
$: ./example_cpp_logistic_regression
```

Result:

- The training and test data and the comparison between the original and predicted labels are shown. The trained model reaches 95 % accuracy. The console text output is as follows:

```
original vs predicted:
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1]
accuracy: 95%
saving the classifier to NewLR_Trained.xml
loading a new classifier from NewLR_Trained.xml
predicting the dataset using the loaded classifier...done!
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1]
                                                                   accuracy:
95%
```

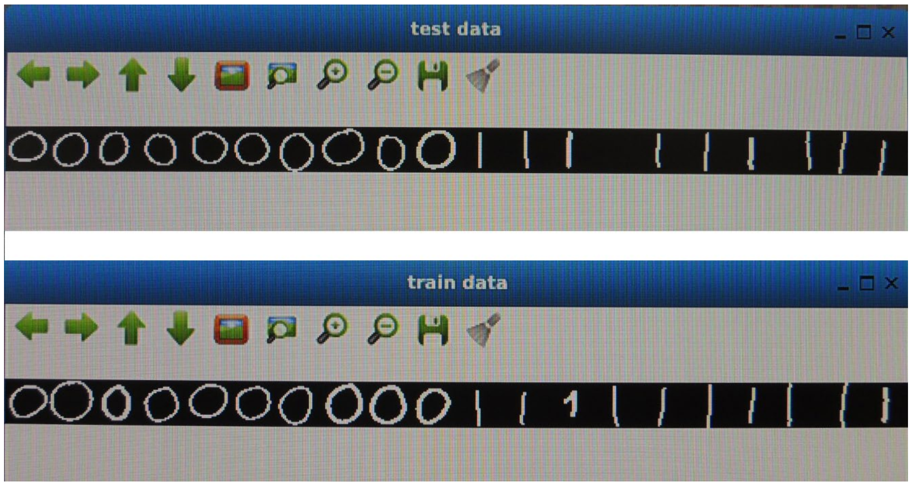


Figure 12. Logistic regression graphics output

5. Arm Compute Library getting started guide

The Arm Compute Library [7] is a collection of low-level functions optimized for the Arm CPU and GPU architectures targeted at image processing, computer vision, and machine learning. It is a convenient repository of optimized functions that developers can source either individually or as a part of complex pipelines to accelerate algorithms and applications. The Arm compute library also supports NEON acceleration.

Two types of examples are described in the following sub-sections:

- Example based on the DNN models with random weights and inputs
- Example based on the AlexNet using the graph API

5.1. Running DNN with random weight and inputs

The Arm Compute Library contains examples for most common DNN architectures, such as AlexNet, MobileNet, ResNet, Inception v3, Inception v4, Squeezenet, and others.

All available examples are at this example build location:

- */usr/share/arm-compute-library/build/examples*

Each model architecture can be tested using the “graph_[dnn_model]” application.

Here is an example of running the required DNN model with a random weight (run the example application without any arguments):

```
$: ./graph_mobilenet_v2
```

The application creates the required network model with random weights and predicts the random inputs. If all components work, the “Test passed” message is printed.

5.2. 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.

The C++ AlexNet example implementation [8] uses the graph API in this folder:

- */usr/share/arm-compute-library/build/examples*

Demo dependencies:

- Download the archive file to the example location folder from:
  developer.arm.com//-/media/developer/technologies/Machine learning on Arm/Tutorials/Running AlexNet on Pi with Compute Library/compute_library_alexnet.zip?revision=c1a232fa-f328-451f-9bd6-250b83511e01
- Create new sub-folder and unzip the file:
  ```
  $: mkdir assets_alexnet
  $: unzip compute_library_alexnet.zip -d assets_alexnet
  ```
- Set the environment variables for execution:
  ```
  $: export LD_LIBRARY_PATH=/usr/share/arm-compute-library/build/examples/
  $: export PATH_ASSETS=/usr/share/arm-compute-library/build/examples/assets_alexnet/
  ```
- Run the example with the command-line arguments from the default location:
  ```
  $: ./graph_alexnet --data=$PATH_ASSETS --image=$PATH_ASSETS/go_kart.ppm --
  labels=$PATH_ASSETS/labels.txt
  ```
- The output of the successful classification is as follows:
  ```
  ---------- 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 getting started guide

TensorFlow [9] is an end-to-end open-source platform for machine learning. It has a comprehensive, flexible ecosystem of tools, libraries, and community resources that enable the researchers to push the state-of-the-art in ML and give the developers the ability to easily build and deploy ML-powered applications.

TensorFlow provides a collection of workflows [13] with intuitive, high-level APIs for both beginners and experts to create machine learning models in numerous languages. TensorFlow provides a variety of different toolkits that enable you to construct models at your preferred level of abstraction. Use the lower-level APIs to build models by defining a series of mathematical operations. Alternatively, you can use higher-level APIs to specify pre-defined architectures, such as linear regressors or neural networks.

6.1. Running benchmark application

This simple example is pre-installed by default on the prepared Yocto image with machine learning enablement. It performs simple TensorFlow benchmarking using the pre-defined model. The graph model file is not included in the target image due to its size. The benchmark binary file location is:

- */usr/bin/tensorflow-1.12.0/examples*

Demo dependencies:

- Download the inception graph model:
  ```
  $: wget storage.googleapis.com/download.tensorflow.org/models/inception5h.zip
  ```
- Unzip the model file to the example target location:
  ```
  $: unzip inception5h.zip
  ```
- Run the example with command-line arguments from the default location:
  ```
  $: ./benchmark --graph=tensorflow_inception_graph.pb --max_num_runs=10
  ```

The benchmark application outputs lots of useful information, such as:

- Run order
- Top by computation time
- Top by memory use
- Summary by node type

For example, the summary node output of the TensorFlow benchmarking is as follows:

```
[Node type]     [count]  [avg ms]      [avg %]    [cdf %]    [mem KB]    [times called]
---------------------------------------------------------------------------------------
Conv2D            22      171.150     64.825%    64.825%    10077.888            22
MatMul             2       35.295     13.368%    78.194%        8.128             2
MaxPool            6       23.723      8.985%    87.179%     3562.496             6
LRN                2       18.823      7.129%    94.309%     3211.264             2
BiasAdd           24        8.475      3.210%    97.519%        0.000            24
Relu              14        3.847      1.457%    98.976%        0.000            14
Concat             3        1.303      0.494%    99.469%     2706.368             3
Const             50        0.619      0.234%    99.704%        0.000            50
AvgPool            1        0.544      0.206%    99.910%       32.512             1
Softmax            1        0.097      0.037%    99.947%        0.000             1
NoOp               1        0.082      0.031%    99.978%        0.000             1
_Retval            1        0.022      0.008%    99.986%        0.000             1
Reshape            1        0.013      0.005%    99.991%        0.000             1
_Arg               1        0.012      0.005%    99.995%        0.000             1
Identity           1        0.012      0.005%   100.000%        0.000             1

Timings (microseconds): count=10 first=281154 curr=242529 min=240048 max=291365 avg=264068
std=19523
```

7. 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.

TensorFlow Lite supports a set of core operators (both quantized and float) tuned for mobile platforms. They incorporate pre-fused activations and biases to further enhance the performance and quantized accuracy. Additionally, TensorFlow Lite also supports the use of custom operations in models.

TensorFlow Lite defines a new model file format, based on FlatBuffers [10]. FlatBuffers is an open-source, efficient, cross-platform serialization library. It is similar to protocol buffers, but the primary difference is that FlatBuffers does not need a parsing/unpacking step for a secondary representation before you can access the data, often coupled with per-object memory allocation. Also, the code footprint of FlatBuffers is an order of magnitude smaller than protocol buffers.

TensorFlow Lite has a new mobile-optimized interpreter, which has the key goal to keep apps lean and fast. The interpreter uses static graph ordering and a custom (less-dynamic) memory allocator to ensure minimal load, initialization, and execution latency.

7.1. Running benchmark application

This simple example is pre-installed by default on the prepared Yocto image with machine learning enablement. Its name is “benchmark_model”. It performs simple TensorFlow Lite benchmarking using the pre-defined models. The model file is not included in the target image, because of its size. The example binary file location is:

- */usr/bin/tensorflow-lite-1.12.0/examples*

Demo dependencies:

- Download the model file [12] using this command:
  ```
  $: wget
  download.tensorflow.org/models/mobilenet_v1_2018_08_02/mobilenet_v1_1.0_224_quant.tgz
  ```
- Unpack the model file:
  ```
  $: tar -xzvf mobilenet_v1_1.0_224_quant.tgz
  ```
- Run the example with the command-line arguments from the default location:
  ```
  $: ./benchmark_model --graph=mobilenet_v1_1.0_224_quant.tflite
  ```

The output of a successful TensorFlow Lite benchmarking is as follows:

```
STARTING!
Num runs: [50]
Inter-run delay (seconds): [-1]
Num threads: [1]
Benchmark name: []
Output prefix: []
Warmup runs: [1]
Graph: [mobilenet_v1_1.0_224_quant.tflite]
Input layers: []
Input shapes: []
Use nnapi : [0]
```

```
Loaded model mobilenet_v1_1.0_224_quant.tflite
resolved reporter
Initialized session in 44.687ms
Running benchmark for 1 iterations
count=1 curr=180071
Running benchmark for 50 iterations
count=50 first=128160 curr=128079 min=127643 max=128319 avg=127944 std=138
Average inference timings in us: Warmup: 180071, Init: 44687, no stats: 127944
```

7.2. Running image classification example

This simple example classifies images of clothing, such as hats, shirts, and others. The “grace_hopper” input image (see Figure 13) is used as a typical sample for the image classification. By default, a proper model file for this example is not included in the target image due to its size. It should be downloaded by the user to the target image.

Figure 13. Image classification input picture

Two different approaches for running this example are used. The simplest way is to use the pre-installed binary application with minimum subsequent steps (see Section 7.2.1, “Using pre-installed example”). The second approach is intended for users who want to create (build) a custom application using sources (see Section 7.2.2, “Building example from sources”).

7.2.1. Using pre-installed example

The example is pre-installed by default in the prepared Yocto image with the machine-learning enablement. Its name is “label_image”. The example binary file location is:

- */usr/bin/tensorflow-lite-1.12.0/examples*

Demo dependencies:

- Download the TensorFlow model file to the example folder. It can be the model file used by the previous benchmark example (see Section 7.1, “Running benchmark application”):

```
$ wget
http://download.tensorflow.org/models/mobilenet_v1_2018_08_02/mobilenet_v1_1.0_224_quan
t.tgz
```

- Unpack the model file to the example binary location:

```
$: tar -xzvf mobilenet_v1_1.0_224_quant.tgz
```

- Run the example with the command-line arguments from the default location:

```
$: ./label_image -m mobilenet_v1_1.0_224_quant.tflite -t 1 -i grace_hopper.bmp -l
labels.txt
```

The output of a successful classification for the “grace_hopper” input image (see Figure 13) is as follows:

```
Loaded model mobilenet_v1_1.0_224_quant.tflite
resolved reporter
invoked
average time: 330.473 ms
0.780392: 653 military uniform
0.105882: 907 Windsor tie
0.0156863: 458 bow tie
0.0117647: 466 bulletproof vest
0.00784314: 835 suit
```

7.2.2. Building example from sources

The image classification example can be downloaded from the TensorFlow repository[13] and built from these sources on the target image.

Demo dependencies:

- Download and make the TensorFlow sources:

```
$ git clone https://github.com/tensorflow/tensorflow.git
$ cd tensorflow
$ git checkout r1.12
$ ./tensorflow/contrib/lite/tools/make/download_dependencies.sh
$ make -f tensorflow/contrib/lite/tools/make/Makefile
$ cd tensorflow/contrib/lite/examples/label_image
```

- Build the “label_image” example using the GNU C++ compiler:

```
$ g++ --std=c++11 -O3 bitmap_helpers.cc label_image.cc -I ../../../.. -I
../../tools/make/downloads/flatbuffers/include -L
../../tools/make/gen/linux_aarch64/lib -ltensorflow-lite -lpthread -ldl -o label_image
```

- Download the TensorFlow model file to the current directory. It is the model file used by the pre-installed example (see Section 7.2.1, “Using pre-installed example”):

```
$ wget
http://download.tensorflow.org/models/mobilenet_v1_2018_08_02/mobilenet_v1_1.0_224_quan
t.tgz
```

- Unpack the model file to the current directory:

```
$ tar -xzvf mobilenet_v1_1.0_224_quant.tgz
```

- Run the example with the command-line arguments from the default location:

```
$ ./label_image -m mobilenet_v1_1.0_224_quant.tflite -t 1 -i testdata/grace_hopper.bmp
-l ../../java/ovic/src/testdata/labels.txt
```

The output of a successful classification for the “grace_hopper” input image (see Figure 13) is the same as for the pre-installed application (see Section 7.2.1, “Using pre-installed example”):

```
Loaded model mobilenet_v1_1.0_224_quant.tflite
resolved reporter
```

```
invoked
average time: 229.14 ms
0.780392: 653 military uniform
0.105882: 907 Windsor tie
0.0156863: 458 bow tie
0.0117647: 466 bulletproof vest
0.00784314: 835 suit
```

8. 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 i.MX 8, Arm NN runs on the CPU with NEON and has multi-core support. Arm NN does not currently support the i.MX 8 GPUs due to the Arm NN OpenCL requirements, which are not met by i.MX 8 GPUs. For more details about Arm NN, check the Arm NN SDK webpage.

To build Arm NN 19.02 using the Yocto Project tools, follow the steps describes in Section 3, "Yocto installation guide". Make sure to perform the additional modifications needed for Arm NN, as described in Section 3.2.4, "Yocto configuration file modifying" (see the "Add packages" instruction).

8.1. 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, TensorFlowLite, ONNX), run the inference on a specified input data, and output the inference result. The Arm NN tests are built by default when building the rootfs image and installed in the */usr/bin* folder.

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. However, the Arm NN tests do not have documentation. Moreover, 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 i.MX 8 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
```

8.1.1. Caffe tests

The Arm NN 19.02 SDK provides the following set of tests for the Caffe models:

```
/usr/bin/CaffeAlexNet-Armnn
/usr/bin/CaffeCifar10AcrossChannels-Armnn
/usr/bin/CaffeInception_BN-Armnn
/usr/bin/CaffeMnist-Armnn
/usr/bin/CaffeResNet-Armnn
/usr/bin/CaffeVGG-Armnn
/usr/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. See also [15].

For example, supposing 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

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

The following sections explain how to run each of the tests, except for “CaffeCifar10AcrossChannels-Armnn” and “CaffeYolo-Armnn”. For the first one, a publicly available pre-trained model was not found. For the second one, there is no way to deduce the exact content of the input image originally used by this test.

8.1.1.1. CaffeAlexNet-Armnn

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

1. Download the model files from:
   raw.githubusercontent.com/BVLC/caffe/master/models/bvlc_alexnet/deploy.prototxt
   dl.caffe.berkeleyvision.org/bvlc_alexnet.caffemodel
2. Transform the network as explained in the introductory part of Section 8.1.1, “Caffe tests”.
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
```

8.1.1.2. CaffeInception_BN-Armnn

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

1. Download the model files from:
   raw.githubusercontent.com/pertusa/InceptionBN-21K-for-Caffe/master/deploy.prototxt
   www.dlsi.ua.es/~pertusa/deep/Inception21k.caffemodel
2. Transform the network as explained in the introductory part of Section 8.1.1, “Caffe tests”.
3. Rename the *Inception21k.caffemodel* file to *Inception-BN-batchsize1.caffemodel*.
4. Copy the *Inception-BN-batchsize1.caffemodel* file to the *models* folder on the device.
5. Find a **.jpg* file containing a shark. Rename it to *shark.jpg* and copy it to the *data* folder on the device.
6. Run the test:

```
$: cd ArmnnTests
$: CaffeInception_BN-Armnn --data-dir=data --model-dir=models
```

8.1.1.3. CaffeMnist-Armnn

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

1. Download the model files from:
   raw.githubusercontent.com/BVLC/caffe/master/examples/mnist/lenet.prototxt
   github.com/ARM-software/ML-examples/blob/master/armnn-mnist/model/lenet_iter_9000.caffemodel
2. Transform the network as explained in the introductory part of Section 8.1.1, “Caffe tests”.
3. Download these two archives and unpack them:
   yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
   yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
4. Copy the *lenet_iter_9000.caffemodel* file to the *models* folder on the device.
5. Copy the *t10k-images.idx3-ubyte* and *t10k-labels.idx1-ubyte* files to the *data* folder on the device.
6. Run the test:

```
$: cd ArmnnTests
$: CaffeMnist-Armnn --data-dir=data --model-dir=models
```

8.1.1.4. CaffeResNet-Armnn

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

1. Download the model file for ResNet50 from:
   onedrive.live.com/?authkey=%21AAFW2-FVoxeVRck&cid=4006CBB8476FF777&id=4006CBB8476FF777%2117895&parId=4006CBB

8476FF777%2117887&o=OneUp

2. Rename the *RestNet-50-model.caffemodel* file to *ResNet_50_ilsvrc15_model.caffemodel*.
3. Copy the *ResNet_50_ilsvrc15_model.caffemodel* file to the *models* folder on the device.
4. Download this image file and copy it to the *data* folder on the device:

   raw.githubusercontent.com/ameroyer/PIC/d136e9ceded0ceb700898725405d8eb7bd273bbe/val_samples/ILSVRC2012_val_00000018.JPEG

5. Find a **.jpg* file containing a shark. Rename it to *shark.jpg* and copy it to the *data* folder on the device.
6. Run the test:

```
$: cd ArmnnTests
$: CaffeResNet-Armnn --data-dir=data --model-dir=models
```

8.1.1.5. CaffeVGG-Armnn

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

1. Download the model files for VGG19 from:

   www.robots.ox.ac.uk/~vgg/software/very_deep/caffe/VGG_ILSVRC_19_layers.caffemodel

   gist.githubusercontent.com/ksimonyan/3785162f95cd2d5fee77/raw/f02f8769e64494bcd3d7e97d5d747ac275825721/VGG_ILSVRC_19_layers_deploy.prototxt

2. Transform the network as explained in the introductory part of Section 8.1.1, “Caffe tests”.
3. Rename the *VGG_ILSVRC_19_layers.caffemodel* file to *VGG_CNN_S.caffemodel*.
4. Copy the *VGG_CNN_S.caffemodel* file to the *models* folder on the device.
5. Find a **.jpg* file containing a shark. Rename it to *shark.jpg* and copy it to the *data* folder on the device.
6. Run the test:

```
$: cd ArmnnTests
$: CaffeVGG-Armnn --data-dir=data --model-dir=models
```

8.1.2. TensorFlow tests

The Arm NN 19.02 SDK provides the following set of tests for the TensorFlow models:

```
/usr/bin/TfCifar10-Armnn
/usr/bin/TfInceptionV3-Armnn
/usr/bin/TfMnist-Armnn
/usr/bin/TfMobileNet-Armnn
/usr/bin/TfResNext-Armnn
```

Before running the tests, the TensorFlow models must be prepared for inference. This process is TensorFlow-specific and uses TensorFlow tools. Therefore, TensorFlow must be installed on your host machine.

The following sections explain how to run each of the tests, except for “TfResNext-Armnn” and “TfCifar10-Armnn”, for which the publicly available pre-trained models were not found.

8.1.2.1. TfInceptionV3-Armnn

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

1. From your host machine, generate the graph definition for the Inception model:

```
# model preparation
$: mkdir checkpoints
# clone the models repository
$: git clone https://github.com/tensorflow/models.git
$: cd models/research/slim/
# export the inference graph
$: python export_inference_graph.py --model_name=inception_v3 --
output_file=../../../checkpoints/inception_v3_inf_graph.pb
```

2. From your host machine, download the pre-trained model and use the TensorFlow tools to prepare it for inference. Note that <*path_to_tensorflow_repo*> refers to the path where you cloned or downloaded the TensorFlow repo.

```
$: cd ../../../checkpoints
# download and extract the checkpoint
$: wget http://download.tensorflow.org/models/inception_v3_2016_08_28.tar.gz -qO- | tar
-xvz
# freeze the model
$: python <path_to_tensorflow_repo>/tensorflow/python/tools/freeze_graph.py --
input_graph=inception_v3_inf_graph.pb --input_checkpoint=inception_v3.ckpt --
input_binary=true --output_graph=inception_v3_2016_08_28_frozen.pb --
output_node_names=InceptionV3/Predictions/Reshape_1
```

3. Copy the *inception_v3_2016_08_28_frozen.pb* file to the *models* folder on the device.
4. Find a **.jpg* file containing a shark. Rename it to *shark.jpg* and copy it to the *data* folder on the device.
5. Find a **.jpg* file containing a dog. Rename it to *Dog.jpg* and copy it to the *data* folder on the device.
6. Find a **.jpg* file containing a cat. Rename it to *Cat.jpg* and copy it to the *data* folder on the device.
7. Run the test:

```
$: cd ArmnnTests
$: TfInceptionV3-Armnn --data-dir=data --model-dir=models
```

8.1.2.2. TfMnist-Armnn

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

1. Download the model file from:

   raw.githubusercontent.com/ARM-software/ML-examples/master/armnn-mnist/model/simple_mnist_tf.prototxt

2. Copy the *simple_mnist_tf.prototxt* file to the *models* folder on the device.

3. Download these two archives and unpack them:
   yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
   yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
4. Copy the *t10k-images.idx3-ubyte* and *t10k-labels.idx1-ubyte* files to the *data* folder on the device.
5. Run the test:

```
$: cd ArmnnTests
$: TfMnist-Armnn --data-dir=data --model-dir=models
```

8.1.2.3. TfMobileNet-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:
   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 dog. Rename it to *Dog.jpg* and copy it to the *data* folder on the device.
5. Find a **.jpg* file containing a 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
```

8.1.3. TensorFlow Lite tests

The Arm NN 19.02 SDK provides the following test for the TensorFlow Lite models:

```
/usr/bin/TfLiteInceptionV3Quantized-Armnn
/usr/bin/TfLiteInceptionV4Quantized-Armnn
/usr/bin/TfLiteMnasNet-Armnn
/usr/bin/TfLiteMobileNetSsd-Armnn
/usr/bin/TfLiteMobilenetQuantized-Armnn
/usr/bin/TfLiteMobilenetV2Quantized-Armnn
/usr/bin/TfLiteResNetV2-Armnn
/usr/bin/TfLiteVGG16Quantized-Armnn
```

The following sections explain how to run some of the tests. Some of the tests are excluded, because it was not possible to find a publicly available model or they need more resources than available on the i.MX 8 embedded application processors.

8.1.3.1. TfLiteInceptionV3Quantized-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:
   download.tensorflow.org/models/tflite_11_05_08/inception_v3_quant.tgz
2. Copy the *inception_v3_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 dog. Rename it to *Dog.jpg* and copy it to the *data* folder on the device.
5. Find a **.jpg* file containing a cat. Rename it to *Cat.jpg* and copy it to the *data* folder on the device.
6. Run the test:

```
$: cd ArmnnTests
$: TfLiteInceptionV3Quantized-Armnn --data-dir=data --model-dir=models
```

8.1.3.2. TfLiteMnasNet-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:
   download.tensorflow.org/models/tflite/mnasnet_1.3_224_09_07_2018.tgz
2. Copy the *mnasnet_1.3_224/mnasnet_1.3_224.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 dog. Rename it to *Dog.jpg* and copy it to the *data* folder on the device.
5. Find a **.jpg* file containing a cat. Rename it to *Cat.jpg* and copy it to the *data* folder on the device.
6. Run the test:

```
$: cd ArmnnTests
$: TfLiteMnasNet-Armnn --data-dir=data --model-dir=models
```

8.1.3.3. TfLiteMobilenetQuantized-Armnn

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

1. From your host machine, download 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 dog. Rename it to *Dog.jpg* and copy it to the *data* folder on the device.
5. Find a **.jpg* file containing a 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
```

8.1.3.4. TfLiteMobilenetV2Quantized-Armnn

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

1. From your host machine, download the model file:
   download.tensorflow.org/models/tflite_11_05_08/mobilenet_v2_1.0_224_quant.tgz
2. Copy the *mobilenet_v2_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 dog. Rename it to *Dog.jpg* and copy it to the *data* folder on the device.
5. Find a **.jpg* file containing a cat. Rename it to *Cat.jpg* and copy it to the *data* folder on the device.
6. Run the test:

```
$: cd ArmnnTests
$: TfLiteMobilenetV2Quantized-Armnn --data-dir=data --model-dir=models
```

8.1.4. ONNX tests

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

```
/usr/bin/OnnxMnist-Armnn
/usr/bin/OnnxMobileNet-Armnn
```

The following sections explain how to run each of the tests.

8.1.4.1. 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:
   onnxzoo.blob.core.windows.net/models/opset_8/mnist/mnist.tar.gz
2. Rename the *model.onnx* file to *mnist_onnx.onnx* and copy it to the *models* folder on the device.

3. Download the following two archives and unpack them (after unpacking, rename the files to use dots instead of hyphens: *t10k-images.idx3-ubyte* and *t10k-labels.idx1-ubyte*, respectively):

   yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz

   yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
4. Copy the *t10k-images.idx3-ubyte* and *t10k-labels.idx1-ubyte* files to the *data* folder on the device.
5. Run the test:

```
$: cd ArmnnTests
$: OnnxMnist-Armnn --data-dir=data --model-dir=models
```

8.1.4.2. OnnxMobileNet-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 dog. Rename it to *Dog.jpg* and copy it to the *data* folder on the device.
5. Find a **.jpg* file containing a 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
```

8.2. Using Arm NN in a custom C/C++ application

You can create your own C/C++ applications for the i.MX 8 family of devices using Arm NN capabilities. This requires writing the code using the Arm NN API, setting up the build dependencies, building the code, and deploying your application. Below is a detailed description for each of these steps. Note that the scenario is cross-compiling a C/C++ application on a Linux OS machine for an i.MX 8 family device board.

1. Write the code.

   A good starting point to understand how to use Arm NN API in your own application is the armnn-mnist example provided by Arm. It includes two applications; one shows how to load and run inference for a MNIST TensorFlow model, and the second one shows how to load and run inference for a MNSIT Caffe model. See the Arm tutorial Deploying a TensorFlow MNIST model on Arm NN.
2. Set up the build dependencies.

From a software developer's perspective, Arm NN is a library. Therefore, create and build an application which uses the Arm NN features, set of Arm NN headers, and set of Arm NN libraries for the target device. The Arm NN headers and libraries are all available within the SDK. Build the SDK when building the Yocto image and install it on your local machine, as described in Section 3.2, "Building NXP eIQ support using Yocto Project tools". When this is done, find:

- Arm NN headers in:

  <Yocto_SDK_install_folder>/sysroots/aarch64-poky-linux/usr/include
- Arm NN libraries in:

  <Yocto_SDK_install_folder>/sysroots/aarch64-poky-linux/usr/lib

3. Build the code.

   To build the "armnn-mnist" example provided by Arm, use the Makefile included in the project with a few minor changes:

   - Remove the definition of "ARMNN_INC" and all its uses. The Arm NN headers are already available in the default include directories.
   - Remove the definition of "ARMNN_LIB" and all its uses. The Arm NN libraries are already available in the default linker search path.
   - Replace "g++" by "${CXX}".

   Build the example:

   - Source the SDK environment:

     ```
     $: source <Yocto_SDK_install_folder>/environment-setup-aarch64-poky-linux
     ```
   - Run make:

     ```
     $: make
     ```
4. Deploy the applications.

At this point, you have two binaries ready to be deployed on the i.MX 8 family device board. All you need to take care of are the runtime dependencies. Regarding the input data, these dependencies are described at the "armnn-mnist" example page. The suggested image described in this document requires Arm NN library dependencies already available on the board and you can run your Arm NN application on the i.MX 8 family device board.

9. ONNX Runtime getting started guide

ONNX Runtime is an open-source inference engine framework developed by Microsoft, supporting the ONNX model format. ONNX Runtime runs on the CPU with NEON and has multi-core support. ONNX Runtime does not currently support the i.MX 8 GPUs due to the lack of OpenCL support. For more details about ONNX Runtime, see the official ONNX Runtime project webpage.

To build Yocto with ONNX Runtime, follow the steps described in Section 3, "Yocto installation guide". Make sure to perform the additional modifications needed for ONNX Runtime, as described in Section 3.2.4, "Yocto configuration file modifying" (see the "Add packages" part).

9.1. Running ONNX Runtime test

ONNX Runtime provides a tool that runs a collection of standard tests provided in the ONNX model Zoo. The tool named “onnx_test_runner” is installed in the */usr/bin* folder.

The ONNX tests are available at github.com/onnx/models and consist of various models in the ONNX format with associated input and expected output data.

Here is an example with the steps required to run the “squeezenet” test:

1. Download and unpack the latest release of the “squeezenet” test archive:
   github.com/onnx/models/tree/master/squeezenet
   s3.amazonaws.com/download.onnx/models/opset_8/squeezenet.tar.gz

2. Copy the *squeezenet* folder containing the model and test data on the device; for example, to the */home/root* folder.
3. Run the “onnx_test_runner” tool, providing the *squeezenet* folder path as the command-line parameter:

```
$: ls /home/root/squeezenet/
model.onnx        test_data_set_11  test_data_set_5  test_data_set_9
test_data_set_0   test_data_set_2   test_data_set_6
test_data_set_1   test_data_set_3   test_data_set_7
test_data_set_10  test_data_set_4   test_data_set_8
$: onnx_test_runner /home/root/squeezenet/
result:
        Models: 1
        Total test cases: 12
                Succeeded: 12
                Not implemented: 0
                Failed: 0
        Stats by Operator type:
                Not implemented(0):
                Failed:
Failed Test Cases:
$:
```

10. Security for machine learning

With the wide-scale deployment of machine learning models, both the safety and security issues become a significant threat. This section describes some of these issues and the countermeasures made available in eIQ to mitigate their impact.

Users concerned about security should also consider the i.MX security features available at the SoC level. Enabling such features can benefit the general system security. For further details, see the processor security reference manual document available in the SoC documentation page at www.nxp.com.

10.1. Adversarial examples

One example of a security and safety issue in the practical large-scale deployment of machine learning is that of adversarial examples (Biggio, et al.), (Szegedy, et al. 2013). These are "*inputs formed by applying small but intentionally worst-case perturbations to examples from the dataset, such that the perturbed input results in the model outputting an incorrect answer with high confidence*" (Goodfellow, Shlens and Szegedy 2014). Hence, one can create specifically crafted inputs (video-images or sounds examples) which try to mislead the machine-learning model such that it misclassifies a road-sign (safety concern) or circumvent authentication when using your voice or face as the authentication (security concern).

For example, Figure 14 and Figure 15 are classified differently by an inception v3 network trained for the ImageNet dataset.

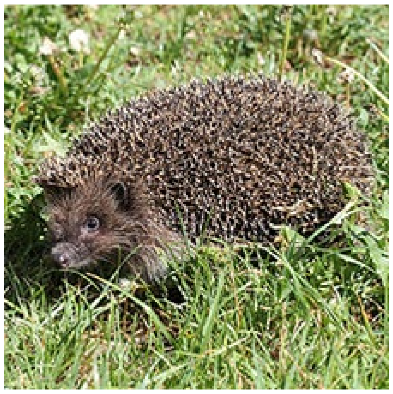


Figure 14. Top 3 Classes: 90.5% Porcupine/Hedgehog, 2.1% Marmot, 1.0% Beaver

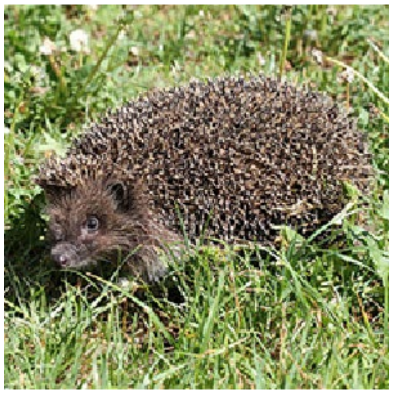


Figure 15. Top 3 Classes: 99.0% Banana, 0.1% Pineapple, 0.05% Porcupine/Hedgehog

Source (public domain): https://commons.wikimedia.org/wiki/File:Erinaceus_roumanicus_2013_G5.jpg

Figure 15 is intentionally modified to be misclassified as a banana. Even though the changes are not visible to the naked eye, the neural net even has a higher confidence that the second image is a banana than that the original image was a hedgehog.

In our package, we have hardening functionality which makes these types of attacks significantly more difficult. This functionality has the advantage that it can harden a model against such carefully selected perturbations without modifying the trained machine learning model. The main idea is to transform the input in specific ways which ensure that the outcome on real input remains unchanged, while the adversarial perturbations of an attack no longer work. By computing multiple of such input transformations and combining the model outputs, it becomes harder to mislead the target system.

Such transformations are application-specific. The code for models which have images as inputs is provided. Specifically, the transformations provided are blurring, rotation, adding noise, and JPEG compression. A detailed description of the API is available in the Doxygen documentation in the subfolder */api-docs*. A complete example can be found in *examples/ax_hardening.cpp*.

Classifying Figure 15 (the adversarial example) with a model, which performs both input rotation and adding noise as the transformations, and voting on the correct result (including the original Figure 15 output) correctly identifies this image as a hedgehog. The rotated version is correctly identified with a 93 % confidence, which is even higher than the unmodified image.

It is necessary to carefully select the right parameters for these transformations. The radius of the blurring must depend on the size of the images and the objects in the image, the angle of rotation must depend on the rotation tolerance of the model, and so on. Even with this additional hardening, it is still possible to create adversarial examples. However, it does require more time to construct them and the search space must be increased when compared to the model that does not employ this technique.

The ml-security package includes an example of hardening against adversarial examples. Two transformations are applied to the inputs: rotation by 3 degrees and blurring. The original image and the transformed images are classified by the network and followed by a voting of the most occurring class. In this example, we use the same Hedgehog images as above, class 335 is a porcupine/hedgehog and class 955 is a banana.

This example runs on the i.MX8QXP board and both images could be classified as a porcupine/hedgehog after applying the techniques mentioned in this section:

```
# ./ax_sample
Using hedgehog image
Top 3 for original image
1: Class 335, Confidence: 0.905243
2: Class 337, Confidence: 0.0213407
3: Class 338, Confidence: 0.0109267
Top 3 for rotated image
1: Class 335, Confidence: 0.947023
2: Class 337, Confidence: 0.0062071
3: Class 338, Confidence: 0.00346831
Top 3 for blurred (bilateral) image
1: Class 335, Confidence: 0.812905
2: Class 338, Confidence: 0.0487511
3: Class 337, Confidence: 0.043767
Classification by votes: 335
Using adversarial example
Top 3 for original image
1: Class 955, Confidence: 0.990323
2: Class 954, Confidence: 0.00107823
3: Class 335, Confidence: 0.000476602
Top 3 for rotated image
1: Class 335, Confidence: 0.935247
2: Class 337, Confidence: 0.0106328
3: Class 338, Confidence: 0.0096081
Top 3 for blurred (bilateral) image
1: Class 335, Confidence: 0.719883
2: Class 337, Confidence: 0.0393073
3: Class 338, Confidence: 0.0233211
```

```
Classification by votes: 335
```

10.2. Model cloning

Machine learning models are susceptible to model extraction using retraining attacks (Tramèr et al. 2016, Correia-Silva et al. 2018), where an adversary exploits the information gained by querying the model for selected inputs and uses that as training data for a counterfeit model. Model extraction attacks enable an adversary to copy the functional behavior of the model into a clone, which can then be used to undermine pay-per-query mechanisms or the competitive edge of the creator of the original model. In addition, the attack can also be used to create a copy which can be inspected to gather additional information for other attacks, like the adversarial examples attack described in Section 10.1, "Adversarial examples".

Our package hardens machine learning models against model extraction by perturbing the fine-grained confidence information included in model output. Perturbing the confidence values produced by the model reduces the information that is leaked about the model's functional behavior. Therefore, an adversary must perform an increased number of queries to the model to gain sufficient information about the model to reproduce it. Since a successful model extraction attack requires more queries, the attack becomes more invasive and hence increases the spent effort of the adversary. Additionally, it becomes easier to detect and to subsequently take effective measures upon.

There are several strategies to perturb the confidence information produced by the machine-learning model. All strategies keep the top-1 accuracy unchanged by keeping the number 1 rank, that means the output class with the highest confidence, intact. The strategies add a small amount of noise to the confidence levels of the predictions, such that model extraction attacks become as effective as when they are performed on only the top label, while some useful information about the remaining ranks is preserved for the legitimate user.

There are two functions that perturb the predicted confidence levels without affecting the rank of the class which has the highest confidence: `addNoise` and `addPseudoNoiseSin`. The former adds random noise to each of the confidence values, the latter adds noise according to the sine function

$\text{confidence}_{\text{new}} = \text{confidence}_{\text{old}} + 0.05 + 0.1 \cdot \sin(50 \cdot (\text{confidence}_{\text{old}} - 0.09)$), plotted in Figure 2, which results in a considerable increase in the number of queries required for successful model extraction. Although this comes at the cost of a reduced precision of the model's prediction confidence, this reduction is limited. If the addition of noise caused the first ranked class to change, this class is swapped back into the top position. Therefore neither `addNoise` nor `addPseudoNoiseSin` affects the top-1 accuracy. The functions support normalization, which scales the new confidence levels such that they sum to 1.

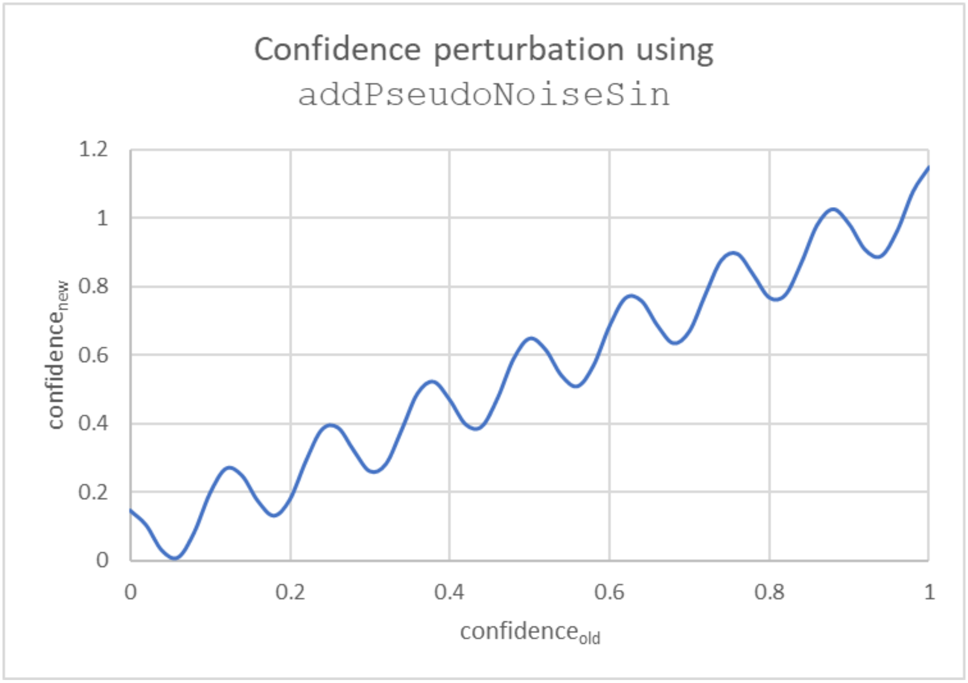


Figure 16. Remapping of the original confidence levels to confidence levels with addPseudoNoiseSin

An additional function for perturbing the confidence levels is `roundConfidenceLevels` that rounds the confidence levels to the nearest given step. Contrary to the other two functions, this function may affect the top-1 accuracy as the highest confidence level may be shared by multiple classes.

All three of these functions can be used as follows:

```
// calculate confidence levels with the neural network
cv::Mat confidences = googleNet.forward();
bool applyNormalization = true;

// add noise to the confidence levels with a range of [-0.1, 0.1]
addNoise(confidences, 0.2f, applyNormalization);

// add pseudo noise to the confidence levels
addPseudoNoiseSin(confidences, applyNormalization);

// round the confidences to quarters precision
roundConfidenceLevels(confidences, 0.25f);
```

The effectiveness of each of the offered perturbation functions is shown in Figure 3. This table shows the loss in top-1 accuracy of a model cloned using the attack by (Correia-Silva, et al. 2018) compared to the top-1 accuracy of the oracle model. The loss is given for attacks with 125,000, 250,000, and 1,000,000 queries to the oracle that the clone is trained on. Labels only is added as a reference and shows the loss in top-1 accuracy for the clone if the confidence levels are disregarded and only the label of the first ranked class is used for the model extraction attack. This shows that the described countermeasure offers some protection against the attack without removing all information given to a legitimate user about the remaining confidence levels, as is the case for a model that would output only the top-1 label.

Table 1. Loss of clone top-1 accuracy compared to oracle model top-1 accuracy

Number of queries \ Hardening method	No hardening	Top-1 label only	`roundConfidenceLevels (conf, 0,25f)`	`addPseudoNoiseSin (conf, true)`	`addNoise (conf, 0.2f, true)`
125,000	8.4%	16.6%	10.6%	16.3%	19.0%
250,000	5.4%	10.7%	7.2%	9.8%	12.4%
1,000,000	2.1%	8.2%	2.8%	3.8%	6.6%

The choice of which hardening method to use or to use no method depends on the use-case in which a model is deployed. We identify a tradeoff between the protection against model extraction and the precision of the model. If precision of the confidence levels is not a requirement, then a model is best protected with the `addNoise` method with a large parameter for the range (e.g. ≥ 0.2). If a use-case requires higher precision of the confidence levels, then the methods `addPseudoNoiseSin`, `addNoise` with a small range (e.g. < 0.2) or `roundConfidenceLevels` with a small range (e.g. < 0.2) can suffice. The parameter range of the noise for `addPseudoNoiseSin` is limited to the interval $[-0.05, 0.15]$.

10.3. Model Inversion

When machine learning is used for privacy-sensitive applications or when the data used for training contains privacy-sensitive information, this sensitive information can be learned by the model. This means that private information can be contained inside the trained model as part of the learned internal parameters. Therefore, model inversion attacks (Fredrikson, et al. 2015) may become an issue. In such an attack an adversary uses information obtained by querying the model to extract privacy-sensitive information from the model.

One example application for model inversion attacks is an adversary attempting to extract the faces of individuals from a model used for face recognition. Another example application is an adversary trying to infer sensitive medical information about a person based on some easily available attributes and a machine learning model trained to predict medical conditions or drug dosages (Fredrikson, et al. 2014).

To counter the threat of exposing privacy-sensitive information, the literature recommends limiting the accuracy of confidence values returned by the model. Generally, that means that similar countermeasures can be used to harden a model against model inversion attacks as the ones recommended to harden a model against the model extraction attack (as shown in Section 7.2 Model Cloning), because both types of attacks are based on information leaked through fine-grained confidence output.

Our package allows to harden pre-trained machine learning models against these attacks by rounding the confidence levels included in the output of the model to a given step size. Experimental results (from Fredrikson, et al. 2015) show that these types of attacks can be mitigated in this way, while it keeps the confidence levels accurate enough to be useful for the intended application of the model. We note however that model inversion has not been the focus of as much research and as many academic publications as the other threats detailed in this chapter. It is therefore possible that stronger attacks (using more queries or combined model extraction-inversion attacks) will allow an attacker to extract confidential information even from a hardened model.

The following code example shows how the rounding function can be applied to the output of a model:

```
// calculate confidence levels with the neural network
cv::Mat confidences = googleNet.forward();

// round the confidence levels
roundConfidenceLevels(confidences, 0.05f);
```

As previously indicated, the countermeasure of adding small perturbations to the confidence output (as recommended to harden models against model extraction in Section 7.2 Model Cloning) can also be used to harden models against model inversion.

The following example code shows how these output transformations can be applied to the output of a model:

```
// calculate confidence levels with the neural network
cv::Mat confidences = googleNet.forward();
bool applyNormalization = true;

// add noise to the confidence levels with a range of [-0.1, 0.1]
addNoise(confidences, 0.2f, applyNormalization);

// add pseudo noise to the confidence levels
addPseudoNoiseSin(confidences, applyNormalization);
```

For further details on these countermeasures see Section 7.2 Model Cloning.

10.4. Library Usage

The library can be included in an image by adding `IMAGE_INSTALL_append = " ml-security-staticdev"` to conf/local.conf, assuming that the appropriate layers and recipes are available.

In that case it should be available for static linking from both, the toolchain and the final image. It is unlikely that the static library in the image will be used but it might be useful for rapid development and it is not very large.

For description on how to create the toolchain see Section 3.2.7 Build the Yocto SDK Toolchain.

On an image that includes the package the library should be found in /usr/lib/libml-security.a and the include files in /usr/include/ml-security/. The pre-built examples and their source should be located in /usr/share/ml-security/examples.

Using the Yocto SDK Toolchain for a supporting image should allow re-building the examples as follows:

```
mkdir /home/user/examples-build
```

```
cd /home/user/examples-build
cmake $SDK_SYSROOT/usr/share/ml-security/examples
make
```

This should result in a binary which works on a device running the accompanying image. Using the examples requires the test_images folder from the examples directory and test_model/inception_v3.pb. This model can be downloaded and converted using the get_model.sh script in the examples folder but it requires Bash, Python and Tensorflow and is intended for use on a Ubuntu host.

11. References

1. NXP eIQ Software
2. NXP eIQ Software Support Community
3. i.MX8 family of Application processor fact sheet:
4. i.MX Software and development tools
5. L4.14.98_2.0.0_LINUX_DOCS documentation
6. Deep learning in OpenCV
7. OpenCV Change Logs
8. ARM Compute library
9. Running Alexnet on Rapsberry PI with Compute Library
10. TensorFlow
11. FlatBuffers
12. What is the difference between TensorFlow and TensorFlow lite
13. TensorFlow Hosted Models
14. TensorFlow sources
15. https://searchenterpriseai.techtarget.com/definition/machine-learning-ML
16. Arm NN documentation for caffe support
17. Biggio, Battista, Igino Corona, Davide Maiorca, Blaine Nelson, Pavel Laskov Nedim Srndic, Giorgio Giacinto, and Fabio Roli. 2013. "Evasion attacks against machine learning at test time." *Machine Learning and Knowledge Discovery in Databases – European Conference, ECML PKDD 2013.* Springer. 387-402.
18. Goodfellow, Ian J., Jonathon Shlens, and Christian Szegedy. 2014. "Explaining and harnessing adversarial examples." *International Conference on Learning Representations.*
19. Szegedy, Christian, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. 2013. "Intriguing properties of neural networks." *International Conference on Learning Representations.*
20. Jacson Rodrigues Correia-Silva, Rodrigo F. Berriel, Claudine Badue, Alberto F. de Souza, and Thiago Oliveira-Santos. 2018. "Copycat CNN: Stealing Knowledge by Persuading Confession

with Random Non-Labeled Data" *2018 International Joint Conference on Neural Networks (IJCNN),* Rio de Janeiro, 2018, pp. 1-8. DOI:https://doi.org/10.1109/IJCNN.2018.8489592

21. Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2016. "Stealing Machine Learning Models via Prediction APIs." *25th USENIX Secur. Symp. USENIX Secur. 16*, Austin, TX, USA, August 10-12, 2016. Ml (2016). DOI:https://doi.org/10.1103/PhysRevC.94.034301
22. Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. 2015. "Model Inversion Attacks that Exploit Confidence Information and Basic Countermeasures". *CCS '15 Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Pages 1322-1333*, Denver, Colorado, USA, October 12-16, 2015. DOI: https://doi.org/10.1145/2810103.2813677
23. Matthew Fredrikson, Eric Lantz, Somesh Jha, Simon Lin, David Page and Thomas Ristenpart. 2014. "Privacy in Pharmacogenetics: An End-to-End Case Study of Personalized Warfarin Dosing". *Proceedings of the 23rd USENIX Security Symposium, Pages 17-32* , San Diego, CA, USA, August 20–22, 2014.

12. Revision history

Table summarizes the changes done to this document since the initial release.

Table 2. Revision history

Revision number	Date	Substantive changes
0	05/2019	Initial release.
1	06/2019	Updated Section 7.2, "Running image classification example".
2	06/2019	Minor formatting changes.
3	09/2019	Added Section 10, "Security for machine learning".
4	05/2020	Added two notes

How to Reach Us:

Home Page:
www.nxp.com

Web Support:
www.nxp.com/support

Document Number: UM11226
Rev. 4
05/2020