Image recognition on Arm Cortex-M with CMSIS-NN

Version 1.0

Image recognition on Arm Cortex-M with CMSIS-NN

Machine Learning Guide

Copyright © 2019 Arm Limited (or its affiliates). All rights reserved.

Release Information

Document History

Version	Date	Confidentiality	Change
1.0	01 April 2019	Non-Confidential	First release

Non-Confidential Proprietary Notice

Confidentiality Status

Product Status

The information in this document is Final, that is for a developed product.

Web Address

http://www.arm.com

Contents

1 Overview

This guide shows you how to perform real-time image recognition on a low-power Arm Cortex-M7 processor, using Arm's CMSIS-NN library. The Cortex-M7 processor is found in a range of solutions from a variety of microcontroller vendors.

Watch a video demo of the steps in this guide here.

Increased compute performance in the smallest Cortex-M based devices is enabling more data processing to move to the edge. *Machine learning* (ML) no longer needs to take place in the cloud or only on advanced application processors. Now, you can take advantage of the benefits afforded by edge computing, such as reduced bandwidth and time-to-decision, increased privacy and reliability, all without relying on an internet connection.

ML at the edge offers new possibilities for new real-time decision-making applications, from voice recognition for smart speakers to facial detection for surveillance cameras.

2 Before you begin

To complete this guide, you'll need some equipment and to download some software for building the final code that runs on the board:

Hardware

- STM32F746G-DISCO discovery board (Cortex-M7)
- STM32F4DIS-CAM Camera module

Note: All the steps can be easily adapted for a different board. We have chosen a specific board to demonstrate the full workflow of how to set up, deploy, and run the code.

Software

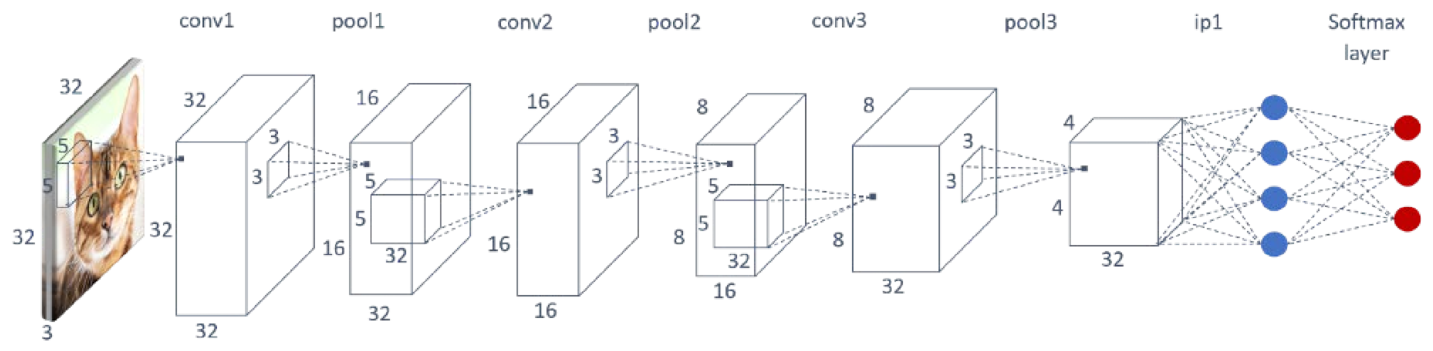
- Ubuntu 16.04
- Python 2.7.12
- Caffe
- GNU Arm Embedded Toolchain. Recommended version: gcc-arm-none-eabi-7-2017-q4-major

3 Define the problem and the model

Our objective here is to find the best way to have your embedded device run image recognition, in order to make your edge device more responsive, reliable, energy efficient and secure by ensuring that your data is processed on the device, rather than in the cloud.

Neural networks (NN) are a class of machine learning (ML) algorithms that have demonstrated good accuracy on image classification, object detection, speech recognition and natural language processing applications. The most popular types of neural networks are multi-layer perceptron (MLP), convolutional neural networks (CNN) and recurrent neural networks (RNN).

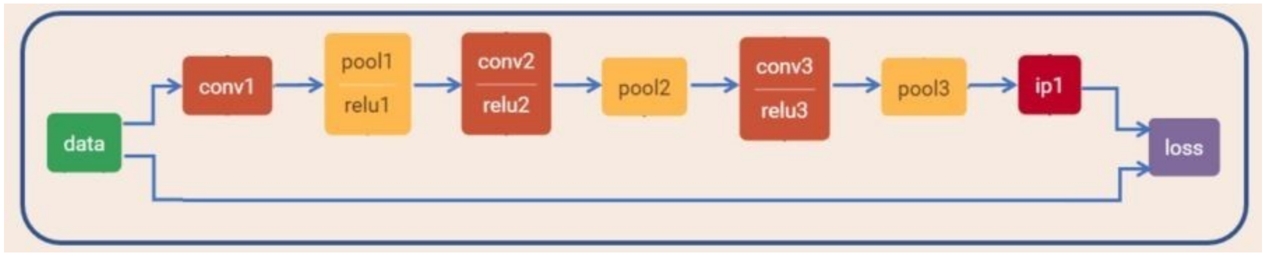
In this case we have chosen to use a CNN, provided in the Caffe examples, for CIFAR-10 image classification task, where the input image passes through the CNN layers to classify it into one of the 10 output classes. Typical CNN models have different types of layers to process the input data, below you can see the ones in the chosen CNN:



The input to the model is a 32x32 pixel color image, which will be classified into 10 classes (cat, deer, dog, horse, etc.) by the CNN. The classification is obtained by having the data flow through the following layers:

- Convolution layer - responsible for extracting features from the image.
- Pooling layer - responsible to progressively reduce the spatial size of the model reducing the number of parameters and the amount of computation in the network, and hence also controlling overfitting. Read more about Pooling on this article.
- Rectified Linear Unit (ReLU) - the activation function responsible for introducing non-linearity in the model. The function returns 0 if it receives any negative input, but for any positive value x it returns that value back. Read this ReLU article for a more thorough explanation.
- Inner product (Inp) or fully connected layer.
- Softmax is used to classify the output into one of the categories, by producing a probability.

For this guide we have pretrained the NN with Caffe, and it can be found on GitHub. If you wish to learn how to train the model on your own you can follow this tutorial. Below you can see the topology of the pretrained network that we will be using for the rest of this tutorial. Note: that this model is not very accurate we have chosen it only for simplicity.



The above image has been generated using this online tool.

4 Set up environment

The first step is to install the Arm Mbed CLI:

```
sudo pip install mbed-cli
```

Next, create the work space we will be using for the rest of the tutorial:

```
# In your <home> directory (cd ~)
mkdir CMSISNN_Webinar
cd CMSISNN_Webinar
# Create new Mbed project
# The --mbedlib flag picks Mbed-os version 2.0
mbed new cmsisnn_demo --mbedlib
```

If you are using the Mbed tools for the first time, install the Mbed CLI prerequisites as shown here, otherwise skip this step:

```
cd cmsisnn_demo/.temp/tools/
sudo pip install -r requirements.txt
```

You also need to install these tools:

```
sudo pip install jsonschema
sudo pip install pyelftools
sudo apt-get install mercurial
```

5 Build basic camera application

On your Linux machine:

```
cd ~/CMSISNN_Webinar/cmsisnn_demo
mbed add http://os.mbed.com/teams/ST/code/BSP_DISCO_F746NG/
mbed add http://os.mbed.com/teams/ST/code/LCD_DISCO_F746NG
```

To change to a stable Mbed build, open the mbed.bld file and replace the first line with this line:

```
https://mbed.org/users/mbed_official/code/mbed/builds/e95d10626187
```

Get all the required Mbed libraries:

```
mbed deploy
```

Clone the Arm Github repository that also contains the example camera application:

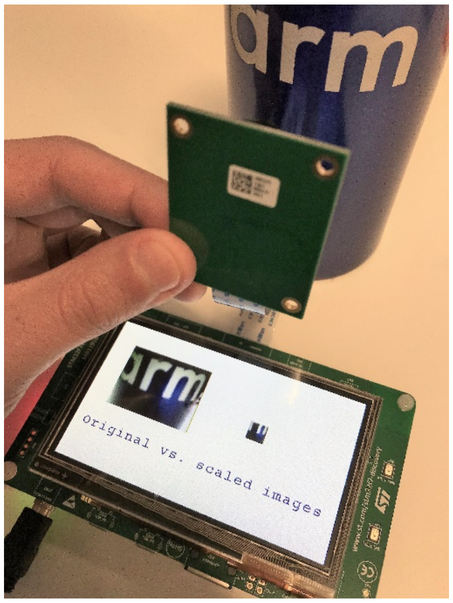
```
cd ~/CMSISNN_Webinar/
git clone https://github.com/ARM-software/ML-examples.git
```

Let's test the image acquisition application:

```
cd cmsisnn_demo/
mbed compile -m DISCO_F746NG -t GCC_ARM --source . --source ../ML-examples/cmsisnn-
cifar10/camera_demo/camera_app/
```

Copy the generated .bin file to the board, directly by dragging and dropping into the board that will appear as a USB device.

If these steps are successful, the board display will look like this:



6 Transform pre-trained Caffe model

In order to have a version of our NN that will be deployable on an Arm Cortex-M microcontroller you will have to follow the steps below.

Quantize the model

Once you have a trained model you will need to optimize it for your microcontroller. To do this you can use Arm's quantization script to convert the Caffe model weights and activations from 32-bit floating point to an 8-bit and fixed point format. This will not only reduce the size of the network, but also avoid floating point computations, that are more computationally expensive.

In the previously cloned `ML-examples` repository, you will find the NN quantizer script that works by testing the network and figuring out the best format for the dynamic 8-bit fixed-point representation ensuring minimal loss in accuracy on the test dataset. The output of this script is a serialized Python pickle(.pkl) file which includes the network's model, quantized weights and activations, and the quantization format of each layer. Running the following command generates the quantized model:

```
# Run command in the ML-examples/cmsisnn-cifar10 directory
cd ~/CMSISNN_Webinar/ML-examples/cmsisnn-cifar10
# Note: To enable GPU for quantization sweeps, use '--gpu' argument
python nn_quantizer.py --model models/cifar10_m7_train_test.prototxt --weights
models/cifar10_m7_iter_300000.caffemodel.h5 --save models/cifar10_m7.pkl
```

Convert the model

Now that you have a more optimized network you should take care of converting it to a C++ file that you will then be able to include in your camera application.

```
# Run command in the ML-examples/cmsisnn-cifar10 directory
python code_gen.py --model models/cifar10_m7.pkl --out_dir code/m7
```

This script gets the quantization parameters and network graph connectivity from the previous step and generates the code consisting of NN function calls. Note: that at this stage the supported layers are convolution, innerproduct, pooling (max/average) and relu. The output is a series of files:

- nn.cpp and nn.h are the files to be included to run the NN on the device
- weights.h and parameter.h consist of quantization rangesHTML

7 Build image recognition application

The last step is for you to include the neural network in the image acquisition application we have built earlier.

In order to achieve this, we need to make some modifications to `ML-examples/cmsisnn-cifar10/camera_demo/camera_app/camera_app.cpp`

```
# In ML-examples/cmsisnn-cifar10/camera_demo/camera_app/
# Rename the camera application
mv camera_app.cpp camera_with_nn.cpp
```

First let's include the code that we have just generated through the previous code_gen.py in the camera_with_nn.cpp file.

```
#include "nn.h"
```

Add classification capabilities

We need to start by defining a few variables:

1. Label variable in order to convert our final prediction into something meaningful.
2. Output data, as the array that will contain the output of the NN.

```
const char* cifar10_label[] = {"Plane", "Car", "Bird", "Cat", "Deer", "Dog", "Frog",
"Horse", "Ship", "Truck"};
q7_t output_data[10]; //10-classes
```

We will also need a function to obtain our top prediction out of the softmax:

```
int get_top_prediction(q7_t* predictions)
{
  int max_ind = 0;
  int max_val = -128;
  for(int i=0;i<10;i++) {
    if(max_val < predictions[i]) {
      max_val = predictions[i];
      max_ind = i;
    }
  }
  return max_ind;
}
```

Modify main function in camera_with_nn.cpp

It's now time to add the neural network function call that has been generated by the `code_gen.py.`

```
// run neural network
run_nn((q7_t*)resized_buffer, output_data);
// Softmax: to get predictions
arm_softmax_q7(output_data,IP1_OUT_DIM,output_data);
```

We also need to identify which one of the classes has the highest probability and for this we add the following:

```
int top_ind = get_top_prediction(output_data);
```

Finally, to test the output add code to print the prediction and the confidence:

```
sprintf(lcd_output_string,"  Prediction: %s       ",cifar10_label[top_ind]);
lcd.DisplayStringAt(0, LINE(8), (uint8_t *)lcd_output_string, LEFT_MODE);
sprintf(lcd_output_string,"  Confidence: %.1f%%   ",(output_data[top_ind]/127.0)*100.0);
lcd.DisplayStringAt(0, LINE(9), (uint8_t *)lcd_output_string, LEFT_MODE);
```

8 CMSIS-NN

In order to fully take advantage of our microcontroller's capabilities let's use the CMSIS-NN optimized libraries, available open source on GitHub. You will make use of CMSIS-NN, a collection of efficient neural network kernels developed to maximize the performance and minimize the memory footprint of neural networks on Arm Cortex-M processor cores.

Neural network inference based on CMSIS-NN kernels achieves 4.6X improvement in runtime/throughput and 4.9X improvement in energy efficiency. For a more detailed overview of all the optimizations refer to this article. In order to use them in the code you should clone the repository as shown below:

```
cd ~/CMSISNN_Webinar
git clone https://github.com/ARM-software/CMSIS_5.git
```

The functions being called in the `run_nn(input, output)` function in the nn.cpp file is calling functions being defined in the CMSIS-NN library, examples are:

```
arm_convolve_HWC_q7_RGB(input_data, CONV1_IN_DIM, CONV1_IN_CH, conv1_wt, CONV1_OUT_CH,
CONV1_KER_DIM, CONV1_PAD, CONV1_STRIDE, conv1_bias, CONV1_BIAS_LSHIFT, CONV1_OUT_RSHIFT,
buffer1, CONV1_OUT_DIM, (q15_t*)col_buffer, NULL);
arm_maxpool_q7_HWC(buffer1, POOL1_IN_DIM, POOL1_IN_CH, POOL1_KER_DIM, POOL1_PAD,
POOL1_STRIDE, POOL1_OUT_DIM, col_buffer, buffer2);
arm_relu_q7(buffer2, RELU1_OUT_DIM*RELU1_OUT_DIM*RELU1_OUT_CH);
```

Now that we have optimized the model and built the complete application it's time to move on to the final step.

9 Deploy transformed model on an Arm Cortex-M processor

The last thing to do before being able to actually run your image recognition application on your device is to compile the files we have just created.

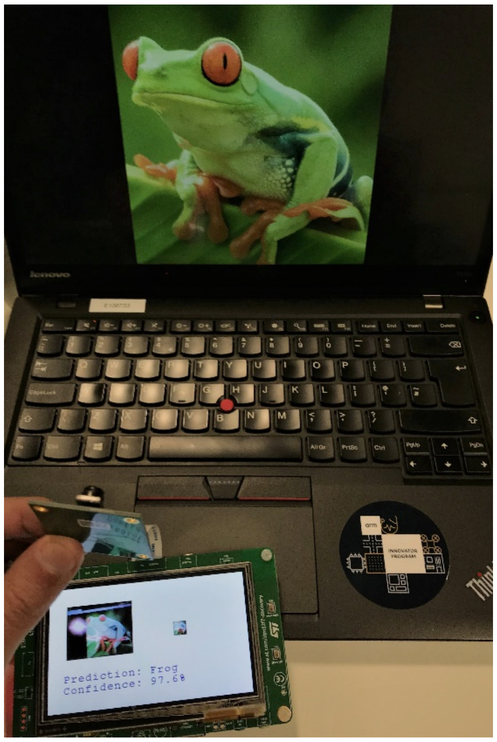
Let's test the image acquisition application:

```
cd cmsisnn_demo/
mbed compile -m DISCO_F746NG -t GCC_ARM --source .  --source ../ML-examples/cmsisnn-
cifar10/code/m7 --source ../ML-examples/cmsisnn-cifar10/camera_demo/camera_app/ --source
../CMSIS_5/CMSIS/NN/Include --source ../CMSIS_5/CMSIS/NN/Source --source
../CMSIS_5/CMSIS/DSP/Include --source ../CMSIS_5/CMSIS/DSP/Source --source
../CMSIS_5/CMSIS/Core/Include -j8
```

Copy the generated .bin file to the board, directly by dragging and dropping into the board that will appear as a USB device.

Note: that this model is very easily affected by changes in light conditions, we have chosen it only for simplicity.

This is what you should see:



```
Prediction: Frog
Confidence: 97.6%
```

Note: The image classification is being done on the resized image. For this reason, make sure to keep the image you are trying to classify centred in the small square.

10 Troubleshooting

Here are some tips to help you troubleshoot some common issues.

Mbed CLI issues

Purge the cache with the following command:

```
mbed cache purge
```

No module named caffe

```
ImportError: No module named caffe
```

Check if it has been appended in pythonpath properly by typing

```
python >>> import sys >>> sys.path ['', '/home/embedded/caffe/python',
'/usr/lib/python2.7', '/usr/lib/python2.7/plat-x86_64-linux-gnu',
'/usr/lib/python2.7/lib-tk', '/usr/lib/python2.7/lib-old', '/usr/lib/python2.7/lib-
dynload', '/home/embedded/.local/lib/python2.7/site-packages',
'/usr/local/lib/python2.7/dist-packages', '/usr/lib/python2.7/dist-packages']
```

if /caffe/python is not there

```
>>> exit()
```

```
export PYTHONPATH=/caffe/python
```

Check failed: mdb_status == 0 (2 vs. 0) No such file or directory

```
Check failed: mdb_status == 0 (2 vs. 0) No such file or directory
```

Open the file:

```
gedit ~/CMSISNN_Webinar/ML-examples/cmsisnn-cifar10/models/cifar10_m7_train_test.prototxt
```

and check that the mean_file and source are pointing to the files in caffe/examples/cifar10/directory as shown below

```
# The following lines will have to be updated.# Note: You will have to replace  with the
path where you have cloned the Caffe repositorymean_file:
"/caffe/examples/cifar10/mean.binaryproto"source:
"/caffe/examples/cifar10/cifar10_train_lmdb"source:
"/caffe/examples/cifar10/cifar10_test_lmdb/"
```

11 Next steps

Now that you have implemented real-time *Machine Learning* (ML) on a Cortex-M device, what other ML applications can you deploy using this approach with CMSIS-NN?

To recap, your models need to be small, pre-trained, and optimized for your input data.

CMSIS-NN is optimized for *Convolutional Neural Networks* (CNNs) and makes use of SIMD instructions. SIMD is only available in Arm Cortex-M4, Cortex-M7, Cortex-M33 and Cortex-M35P processors. Although it is possible to run CMSIS-NN on earlier processors, such as the Cortex-M0, you won't see the same performance benefits that are on offer with Cortex-M4, Cortex-M7, Cortex-M33 and Cortex-M35P based devices.

To keep learning about ML with Arm, here are some resources related to this guide:

- Find out what the Arm Innovators are up to - DevDay: Explore AI on Arm
- Read how to add intelligent vision to your next embedded product
- Learn more by reading the latest white paper on CMSIS-NN
- Read how to do keyword spotting using CMSIS-NN
- Learn more about how to use the OpenMV camera for ML applications
- Explore more about Arm NN

GitHub Repositories

You can find more code examples and resources for performing ML with Arm technologies on our GitHub pages:

- Arm ML example GitHub
- CMSIS-NN GitHub
- KWS GitHub

This guide shows that you don't need a high-spec machine or cloud-based engine to do real-time *Machine Learning* (ML) tasks. Thanks to the recent advancements in Arm hardware and software, you can now do machine learning fast and efficiently on embedded devices, taking advantage of all the benefits of data processing at the edge.

Given the exponential growth in the number of IoT devices being deployed, and considering that the majority of them have an Arm Cortex-M microprocessor at their heart, it is only natural to want to make sure that these are running ML workloads efficiently. Now you know the steps required for deploying a trained Caffe model on an Arm Cortex-M based microcontroller device and how to optimize your neural network functions using the CMSIS-NN library.