Deploying an application using Arm NN

Document number: ARM-ECM-0744361 Version: 1.1

Date of Issue: 20/07/2018

Abstract

This guide is an accompaniment to the Arm NN launch and discusses an example program that allows you to deploy an application with Arm NN to an Arm-based device.

Keywords

Machine learning, artificial intelligence, neural network, Arm NN, TensorFlow.

Contents

1 Overview

This guide shows you how to run a TensorFlow model using Arm NN.

Some familiarization with neural networks and the MNIST dataset is expected. If you are new to either of these, read this TensorFlow introduction and this overview of MNIST.

This guide uses Arm NN to run a model following these steps:

1. Load and parse the MNIST test set.
2. Import graph.
3. Optimize and load onto compute device.
4. Run graph on device.

The complete example with source code, data and model is available here on GitHub. The guide explains each step of the example code to help you understand each stage of the process.

At the end of this guide you will be able to run the complete example yourself and use the knowledge you have gained to build and run your own models using Arm NN.

2 Load and parse the MNIST test set

The example code starts by loading and parsing the MNIST test set:

```
// Load a test image and its correct label
std::string dataDir = "data/";
int testImageIndex = 0;
std::unique_ptr<MnistImage> input = loadMnistImage(dataDir, testImageIndex);
```

The loadMnistImage helper function is not covered here. In simple terms, it just parses the MNIST data files and returns an MnistImage struct for the requested image with a label and a 28*28=784 element array containing the data:

```
// Helper struct for loading MNIST data
struct MnistImage
{
    unsigned int label;
    float image[g_kMnistImageByteSize];
};
```

3 Import graph

Arm NN provides parsers for reading model files from neural network frameworks. There are typically two steps to do this:

1. Load the model.
2. Bind the input and output points of its graph.

```
// Import the TensorFlow model. Note: use CreateNetworkFromBinaryFile for .pb files.
armnnTfParser::ITfParserPtr parser = armnnTfParser::ITfParser::Create();
armnn::TensorInfo inputTensorInfo({ 1, 784, 1, 1 }, armnn::DataType::Float32);
armnn::INetworkPtr network = parser->CreateNetworkFromTextFile("model/simple_mnist_tf.prototxt",
                                                               { {"Placeholder", inputTensorInfo} },
                                                               { "Softmax" });
```

TensorFlow graphs in both text and binary ProtoBuf formats are supported. For more details on freezing TensorFlow graphs to include their weights, see this guide.

Tip: After this step, the code is common regardless of the framework that you started with as the INetwork and two BindingPointInfo objects provide everything that is needed.

4 Optimize and load onto compute device

Arm NN supports optimized execution on multiple devices, including CPU and GPU. Before you start executing a graph, you need to select the appropriate device context and optimize the graph for that device.

Making use of an Arm Mali GPU is as simple as specifying Compute::GpuAcc when creating the context, no other changes are required.

For a Raspian base install the only dependency that you need to add is TensorFlow from Google's binaries. First, install some TensorFlow prerequisites by entering the following in the command line:

```
// Create a context and optimize the network for one or more compute devices in
order of preference
// e.g. GpuAcc, CpuAcc = if available run on Arm Mali GPU, else try to run on ARM v7
or v8 CPU
armnn::IRuntime::CreationOptions options;
armnn::IRuntimePtr context = armnn::IRuntime::Create(options);
armnn::IOptimizedNetworkPtr optNet = armnn::Optimize(*net, {armnn::Compute::GpuAcc,
armnn::Compute::CpuAcc}, runtime->GetDeviceSpec());
//Load the optimized network onto the device
armnn::NetworkID networkIdentifier;
context->LoadNetwork(networkIdentifier, std::move(optNet));
```

5 Run graph on device

Running inference on the compute device is performed through the context's EnqueueWorkload function:

```
// Run a single inference on the test image
std::array<float, 10> output;
armnn::Status ret = context->EnqueueWorkload(networkIdentifier,
                                             MakeInputTensors(inputBindingInfo, &input->image[0]),
                                             MakeOutputTensors(outputBindingInfo, &output[0]));
```

Here the input and output tensors are bound to data and the loaded network identifier is selected. The result of the inference can be read directly from the output array and compared to the MnistImage label we read from the data file:

```
// Convert 1-hot output to an integer label and print
int label = std::distance(output.begin(), std::max_element(output.begin(), output.end()));
std::cout << "Predicted: " << label << std::endl;
std::cout << "   Actual: " << input->label << std::endl;
```

In this case, the std::distance function is used to find the index of the largest element in the output - the equivalent to NumPy's argmax function.

6 Deploy an application using Arm NN

Your application needs to be linked with the Arm NN library and the TensorFlow parsing library:

```
g++ -std=c++11 -I$(ARMNN_INC) mnist.cpp -o mnist -L$(ARMNN_LIB) -larmnn -larmnnTfParser
```

For convenience, Arm NN can run with a reference implementation on x86 for development and testing, but the optimized kernels are only available for Arm CPUs and GPUs. This means that you must run your profiling and optimization steps on a real Arm-powered device as x86 performance is not representative!

7 Next steps

1. Explore this complete example in our GitHub repository.
2. Visit the Arm NN site to see what else is possible.