nn4mc

Sep 04, 2020

NN4MC

nn4mc is a software package that builds C code for neural networks to run on off-the-shelf microcontrollers!

CHAPTER 1

About

1.1 Why nn4mc?

nn4mc bridges the gap between high level neural network training in fully capable PCs and microcontrollers.

> We present a library to automatically embed signal processing and neural network predictions into the material robots are made of. Deep and shallow neural network models are first trained offline using state-of-the-art machine learning tools and then transferred onto general purpose microcontrollers that are co-located with a robot's sensors and actuators. We validate this approach using multiple examples: a smart robotic tire for terrain classification, a robotic finger sensor for load classification and a smart composite capable of regressing impact source localization. In each example, sensing and computation are embedded inside the material, creating artifacts that serve as stand-in replacement for otherwise inert conventional parts. The open source software library takes as inputs trained model files from higher level learning software, such as Tensorflow/Keras, and outputs code that is readable in a microcontroller that supports C. We compare the performance of this approach for various embedded platforms. In particular, we show that low-cost off-the-shelf microcontrollers can match the accuracy of a desktop computer, while being fast enough for real-time applications at different neural network configurations. We provide means to estimate the maximum number of parameters that the hardware will support based on the microcontroller's specifications.
>
> Make sure that the first layer in the neural network specifies the *input_shape* field when using Tensorflow 2.0 in order to get an appropriately functioning result.

```
Aguasvivas Manzano, Sarah, et al. “Embedded Neural Networks for Robot␣
↪Autonomy.” International Symposium on Robotics Research, 2019.
```

Or

```
Manzano, S. A., Hughes, D., Simpson, C., Patel, R., & Correll, N. (2019).␣
↪Embedded Neural Networks for Robot Autonomy. arXiv preprint arXiv:1911.
↪03848.
```

Or

```
@misc{nn4mc,
        title={Embedded Neural Networks for Robot Autonomy},
        author={Sarah Aguasvivas Manzano and Dana Hughes and Cooper Simpson␣
↪and Radhen Patel and Nikolaus Correll},
        year={2019},
        eprint={1911.03848},
        archivePrefix={arXiv},
        primaryClass={cs.RO}
    }
```

CHAPTER 2

nn4mc_cpp

nn4mc_cpp. . .

CHAPTER 3

Installation Guide

To get setup with *nn4mc* we first need the following dependencies:

- HDF5, pickle or the preferred library depending on the source
- Boost
- g++
- cmake
- json

3.1 Get all Dependencies in One Step

Go to the scripts/ folder and type:

3.1.1 Linux

```
./scripts/setup_linux_mint.sh
```

3.1.2 MacOS

```
./scripts/setup_macos.sh
```

3.2 HDF5

3.2.1 Installing HDF5 >=1.10.4 from conda

If you are a conda user, the simplest way to obtain a version of hdf5 that is stable across platforms is using conda. The command to type is:

```
conda install -c anaconda hdf5
```

3.2.2 Installing HDF5 < 1.8.16 from HDF5 Group

This is a more manual installation. This installation will lead a very stable Linux parsing in HDF5, but leads to some compatibility problems for MacOS. Go to this website: HDF5 Group or type:

```
wget http://h5cpp.org/download/hdf5-1.10.4.tar.gz
```

Untar the file as in :

```
tar -xvzf hdf5-1.10.4.tar.gz
```

Configure the files as in:

```
cd hdf5-1.10.4 && ./configure --prefix=/usr/local && make -j2 && sudo make install
```

Then download the deb files:

```
wget http://h5cpp.org/download/h5cpp_1.10.4.1_amd64.deb
```

Then type:

```
sudo dpkg -i h5cpp_1.10.4.1_amd64.deb
```

```
cd /usr/lib/x86_64-linux-gnu
```

```
sudo ln -s libhdf5_serial.so.8.0.2 libhdf5.so
sudo ln -s libhdf5_serial_hl.so.8.0.2 libhdf5_hl.so
```

3.3 Installing the boost library

From Linux:

```
sudo apt-get install libboost-all-dev
```

From MacOS:

```
brew install boost
```

3.4 Installing nlohmann/json

Go back to *nn4mc* and go to the *lib/* folder.

```
git clone https://github.com/nlohmann/json nlohmann_json
```

CHAPTER 4

Tutorials

4.1 Getting Started

The first step after the installation is done is to create a build folder and build our first example to make sure you are setup! Go to the root *nn4mc* folder and create a *build/* folder.

```
cd path/to/nn4mc
mkdir build
cd build
```

Now, you are ready to make your first example project. We recommend to add your hdf5 files under the *data/* folder to keep your version of the repository clean. However, you are free to store that file wherever you'd like (inside or outside the folder, as long as you can provide the path). We added some sample hdf5 files under *data/*. To test your installation do the following:

```
cd build
cmake ..
make
```

The sample code will generate many executables under *build/*, the one we recommend testing because it covers all the necessary components is called *generator_test_file*. Run it by typing in the command line:

```
./generator_test_file
```

If this works, it will generate code in the root folder with the file name specified. By default, this filename is called: *example_out/* and is located at the root folder of *nn4mc*.

The code contained in the generated file is ready to be dragged and dropped into an IDE, for example, the Arduino IDE as seen below.

4.2 Using your HDF5 File

The example code contained under *examples/generator_test_actual_file.cpp* contains all the necessary components to create your C files. Here is an example on how to use this library on hdf5 files. In this example we generate the code necessary to implement lenet in a microcontroller:

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include "parser/HDF5Parser.h"
#include "datastructures/tensor.h"
#include "generator/weight_generator.h"
#include "datastructures/weights.h"
#include "generator/layer_generator.h"
#include "generator/code_generator.h"

#include "datastructures/Layer.h"
#include "datastructures/NeuralNetwork.h"

int main()
{

   HDF5Parser P("../data/lenet.hdf5");
   P.parse();
   NeuralNetwork* nn = P.get_neural_network();
   nn->BFS();
   nn->reset();

   CodeGenerator* code_gen = new CodeGenerator(nn, "../templates/esp32", "../example_
↪out");
   code_gen->generate();
   code_gen->dump();

   delete nn;
   return 0;
}
```

In the future, we expect to remove some of these necessary lines of code in a way that the end-user does not have to declare so many pointers.

[TBD]

CHAPTER 5

Guide for Devs

5.1 Main Structure of the Code

`nn4mc` is composed by the following modules:

data_structures Contains *Tensors*, *Weights*, *Layers* and *NeuralNetwork*.

parser Parses an incoming file to output a *NeuralNetwork* object.

code_generator Takes in a *NeuralNetwork* object to generate files with C/C++ code.

5.2 data_structues/Tensor

Tensor objects are mostly used in this work when we need to store large arrays, especially of the *float* or *double* variable types that occupy lots of space. What makes a *Tensor* special is that any matrix or array that is converted into a tensor becomes part of a large sequence of characters and this object type makes us capable of converting to and from said sequence of characters. For example, instead of having *float arr[3] = {1.000, 1.000, 1.000}* we have a *tensor* that under the hood stores our array into a slightly more complicated verion but similar to *"1.00001.0001.000"* . To declare a *Tensor* we need two things: A *std::vector<int>* that indicates the sizes of each dimension in the tensor. Then, to assign numerical values at specific points in a *Tensor*, we use a pointer to the tensor and the assignment operator. Here is an example of how to declare and populate a *Tensor*.

```
std::vector<int> dimensions;
dimensions.push_back(5);
dimensions.push_back(2); // a 5x2 matrix

Tensor<double> T = Tensor<double>(dimensions);

for(int i=0; i<dimensions[0]; i++)
  {
          for(int j=0; j<dimensions[1]; j++)
          {
```

(continues on next page)

(continued from previous page)

```
                T(i,j) = i*dimensions[1] + j;
            }
    }
```

5.3 Data Structures

The *data_structures* module in *nn4mc* contains all the built-in data structures, such as weights, layers, tensors and neural networks.

5.4 Parser

Parser is the first interface between the model file and *nn4mc* this is independent of *code_generator*, however, *code_generator* takes as input an object that may be generated using a *Parser*. It takes as input the file that contains the model and outputs a *NeuralNetwork* object with all the complete information.

5.5 Code Generator

code_generator takes as input the *NeuralNetwork* object generated and populated using any of the *Parser* instances and outputs a set of C/C++ files with all the code that can execute the neural network operations.

5.6 data_structures/Weight

Weights contain a single *Tensor*. Even though layers contain both weights and biases, these weights and biases are saved as instances of weight. This means that a layer will have two pointers to weight objects; one for the weights and one for the biases. *Weights* have a specific method that might come in handy and it is called *Weight::get_weight_tensor()*. This allows us to copy the values for the weight tensor.

5.7 data_structures/Layer

Layers are the most functional objects for neural networks. Different layer types will yield different outputs and will process differently the data, e.g. a fully connected (Dense) layer's output will be different from a convolutional layer output. To declare a new layer we just instantiate the name of the layer type because layer type objects derive from *Layer*, which is an abstract class. Currently, the layer type objects that can be instantiated are the following:

Dense This is a fully connected or dense layer. An MLP will have only Dense layers.

Activation Sometimes Keras users add activations as if they were independent layers. This accounts for when the user does that.

Conv1D Performs 1D convolutions for 1D signal processing. This object fills out all the required arguments needed for these layer types.

Conv2D Contains all the necessary components to perform 2D convolutions.

MaxPooling1D Contains all the necessary components to perform 1D maxpooling.

MaxPooling2D Contains all the necessary components to perform 2D maxpooling.

Activation Contains all the necessary components to perform activation layers as separate layers.

Dropout Contains all the necessary components to at least read dropout layers even though the feedforward process does not require any additional processing from the dropout part.

In the future, we will add the following layer types:

GRU Will contain the necessary components to perform GRUs.

LSTM Will contain the necessary components to perform LSTMs.

SimpleRNN Will contain the necessary components to perform simple RNNs.

5.8 data_structures/NeuralNetwork

NeuralNetwork is a directed graph that can be transversed using BFS. Each node of *NeuralNetwork* points at a layer type object and at a weight and bias tensor.

5.9 parser/HDF5Parser

HDF5Parser takes as input an *.hdf5* file and extracts all the information to output a *NeuralNetwork* object with all of the necessary pieces filled out from the information in the file.

5.10 parser/PickleParser

PickleParser will soon be a capability in *nn4mc*. This will take as input a *.pth* file coming from PyTorch and will generate the same neural network that *HDF5Parser* exports.

CHAPTER 6

nn4mc_py

nn4mc_py…

CHAPTER 7

Installation

This will talk about installing nn4mc_py

CHAPTER 8

Using nn4mc_py

This will discuss using nn4mc_py

CHAPTER 9

Development

This will discuss ongoing development and how to get involved.

CHAPTER 10

Examples

CHAPTER 11

Tutorials

CHAPTER 12

Using Generated Code

12.1 Arduino

First, open a new sketch in Arduino, then, open all the files that were generated using nn4mc in Arduino.

The following code is an example of what it would take to get *nn4mc* to work on your Arduino code. In this example, we allocate and send to the neural network an input of ones. This code looks as follows:

First, create the prototypes for the functions that we will be using from *nn4mc*:

```
void buildLayers();
float * fwdNN(float* data);
```

In the setup function after we begin our serial port, we call the function *buildLayers()*. This function will initialize all the layers and create the necessary components for our feed forward.

```
void setup() {
    Serial.begin(115200); // feel free to adjust this as desired
    buildLayers();

}
```

In the loop function, after we collect our input, we call *fwdNN(input)*, which is the function that will output a pointer. This pointer will contain the output data from the neural network. You can access this output as a regular array, for example, output[0] indicates the first element in the output layer and so forth. In the following example, we have a neural network whose input layer is is of size *(None, 10, 1)* and output size *(3)*.

```
void loop() {

      float * input= (float*)malloc(10*sizeof(float));
      for (int i=0; i<10; i++) input[i] = 1.0;

      float * output;
```

(continues on next page)

(continued from previous page)

```
      output = fwdNN(input);

      for (int i=0; i<3; i++) {
          Serial.print(output[i]);
          Serial.print(" ");
      }

      Serial.println();

      free(output); // output needs to be freed for best performance. Please do not␣
↪free input.
      delay(1);
    }
```

CHAPTER 13

Testing Instructions

13.1 Arduino

You can start with a trained Keras file, run it through `nn4mc` using the instructions above and import the code in Arduino. Make sure you test the specific layers. Use `loadModel.py` under data to compare against the actual Keras results/outputs at specific places within the neural network. We usually test the performance of the layer by sending an array of ones as input to the neural newtork and seeing how faithful the results should be. There should be a 1:1 fidelity with those results since both languages use float32 (Arduino serial monitor sometimes rounds the numbers, so you could just print the first 16 digits for the number or something like that). Whatever the result of the test is, write a new issue with the following template:

```
FILE: filename.cpp
RESULT: passing/failing/need_further_test
TEST TYPE: Unit test/Integration test
DESCRIPTION:

input:
output:
desired output:
NOTES:
[write additional notes here]

RECOMMENDATIONS:
[write specific template fixes that you think might fix it (optional)]
```

Please do not try to modify source code or template codes, but if you find an error in the templates make sure to add it as a recommendation.

13.2 ESP-IDF

[TBD]

CHAPTER 14

Indices and tables

- genindex
- modindex
- search