Getting started with HLS4ML C++ Code Compilation

Make sure you have set up a C++ compiler on your system.

On Mac or linux, this should already exist in the form of g++

On Windows you will most likely have to install your own if you have not already

1. Navigate to the firmware folder in your hls directory, this will have files like "myproject.cpp" and "parameters.h"
2. To get an understanding of the layers and how data flows through the system look at "myproject.cpp"
   a. There should be a function called myproject, this contains all of the layers and is what runs the neural network.
   b. Each layer will usually have 3 lines, the first creating a variable to store the output, the second to set the dimension of the layer, and the third to instantiate the layer with the inputs, outputs, and any weights or biases it takes in.
   c. In between these layers is where collecting data is the easiest and usually the most straightforward. This can be done with C++ print statement.
3. In this file, add the following function:
```
int main(int argc, char** argv) {
   input_t inputVals[/** PUT NUMBER OF INPUTS HERE **/];
   /** SET INPUT VALUES HERE **/
   result_t outputVals[/** PUT OUTPUT WIDTH HERE **/];
   unsigned short const_size_in_1;
   unsigned short const_size_out_1;
   myproject(inputVals, outputVals, const_size_in_1, const_size_out_1);
   return 0;
}
```
   a. This will call the neural network when it is compiled to make sure you are able to see the output
4. In the "parameter.h" file, each layer's settings are defined, as well as the location in the "nnet_utils" folder they will be located in. You can mess with these values to fine tune the neural net or see how different values affect the output. I generally recommend saving this for last
5. In the "defines.h" file, general defines are set. This includes the input size, output size, filter sizes, as well as typedefs of hls specific variables
6. In addition to these files, there is the nnet_utils folder. In this folder you will find all of the possible layers stored in header files. If you find you need more in-depth information from a layer, beyond its input and output, these are the files you modify

Compiling the HLS code

HLS4ML has some functions that will break in newer versions of g++. In general, first test if your code compiles with “g++ myproject.cpp”, but if it doesn’t try running “g++ myproject.cpp -std=c++11”

On successful compilation you should see something similar to “a.exe”, if you are on Windows, or an “a.out” if you are on Mac or Linux added to the direction. In your command prompt type “./(filename here)” and the code should run printing to the terminal

Printing from the C++ code and some formatting tricks

To start, at the top of your “myproject.cpp” add “#include <iostream>” and “#include <string>” If you are printing from another file and it is not compiling, try adding these lines to the top of that file as well.
In order to print to the console in C++ the following line of code is used:

```
std::cout << "Hello World" <<std::endl;
```

Cout is the destination for the text and endl is to guarantee a new line is printed

If you instead want to print a variable, put the variable you want to call in between the “<<” instead. If you want to print multiple variables or variables and text, you can have multiple sets of “<<” for one cout. For example:

```
std::cout << "Hello World, my name is " << name << ”!” std::endl;
```

Below is a for loop I used to print each of the outputs of a layer in an easier to read format. HLS4ML has a print format for its ap_fixed types however it includes a decimal point and can have an inconsistent number of bits as it removes leading 0s.

```
for (int i = 0; i < 10; i++) {
           std::string str = layer4_out[i].to_string();
           // If the variable is 0 (formatted as 0) output all 0s
           if (layer4_out[i] == 0) {
               std::cout << "0b00000000000000000" << std::endl;
           } else {
               // Find where the decimal point is and sign extend to make sure it is
the right length
               if (str.find('.') < 9) {
                   str.insert(str.find('b')+1, 9 - str.find('.'), str.at(2));
               }
               // Remove the decimal
               std::cout << str.erase(str.find('.'), 1) << std::endl;
           }
   }
```

This type of loop is how most inputs and outputs are checked. For additional information, such as values in the middle of a layer, the print code can be placed inside the for loops of the layers. This may take a bit more time to determine what each value represents. I usually recommend putting a print statement which prints a label for the for loop right before any loops of interest.

Python Helper Code

Below is python code which helps to convert floats into Two's Complement, and convert Two's Complement arrays into some which can be easily pasted into the main function of the "myproject.cpp" so you don't have to manually set each index. You'll need to install the binary-fractions python library first, which can be found here.

```
import binary_fractions
# Takes an input file of floats, converts them Twos Complement and then stores them in
an output file
with open('outputFile.txt', 'w') as o:
   with open('file.txt', 'r') as f:
       values = f.read().split(', ')
       print(values)
       for x in values:
           binaryVal = binary_fractions.TwosComplement(float(x),
len(binary_fractions.TwosComplement(float(x))) + 6)[0:17]
           o.write(binaryVal + ', ')


# Takes an input array of binaryNumbers and outputs them in a way which can be pasted
into the main function of the C++ code
binaryNumbers = [
   "000000.0010011100", "000000.0110001111", "000000.0001111101", "111111.1110110011",
"111111.1101111101", "000000.0001000101", "000000.0010011100", "000000.0011001010",
   "111111.1010010110", "000000.0010111101", "000000.0111100000", "111111.1111111001",
"000000.0100101001", "111111.1110101111", "000000.0000111011", "000000.0000100010",
   "000000.0010011100", "000000.0110001111", "000000.0001111101", "111111.1110110011",
"111111.1101111101", "000000.0001000101", "000000.0010011100", "000000.0011001010",
   "111111.1010010110", "000000.0010111101", "000000.0111100000", "111111.1111111001",
"000000.0100101001", "111111.1110101111", "000000.0000111011", "000000.0000100010",
   "000000.0010011100", "000000.0110001111", "000000.0001111101", "111111.1110110011",
"111111.1101111101", "000000.0001000101", "000000.0010011100", "000000.0011001010",
   "111111.1010010110", "000000.0010111101", "000000.0111100000", "111111.1111111001",
"000000.0100101001", "111111.1110101111", "000000.0000111011", "000000.0000100010",
   "000000.0010011100", "000000.0110001111", "000000.0001111101", "111111.1110110011",
"111111.1101111101", "000000.0001000101", "000000.0010011100", "000000.0011001010",
   "111111.1010010110", "000000.0010111101", "000000.0111100000", "111111.1111111001",
"000000.0100101001", "111111.1110101111", "000000.0000111011", "000000.0000100010"
]

for i, x in enumerate(binaryNumbers):
   print("inputVals[" + str(i) + "] = " +
str(binary_fractions.TwosComplement(x).to_float()) + ";")
```