HLS4ML Layer Implementation Tutorial

NOTE: THIS TUTORIAL WILL ONLY TEACH YOU HOW TO IMPLEMENT TF.KERAS LAYERS & CUSTOM LAYER (USING PR 528) IN HLS4ML. FOR PYTORCH LAYER & OTHER LAYERS PLEASE DISCUSS WITH YOUR ADVISOR

Regular TensorFlow Keras Layer Implementation Overview

1. Please check the current supported Layers at https://github.com/fastmachinelearning/hls4ml/tree/main/hls4ml/converters
2. Transformers and Bidirectional layers are also available based on the request.
3. The implementation for regular TensorFlow Keras Layer contains both the Python Files and the C++ file.

Python Files

1. Adding a converter under this directory
    a. https://github.com/fastmachinelearning/hls4ml/tree/main/hls4ml/converters
    b. The converter parses the needed args from the Keras layer to a new layer object.
    c. The converter also defines the output shape of the layer.
    d. After the converter is added, you should see the correct output shape printed by pushing the model through HLS4ML.
2. Adding required implementation in layers.py
    a. https://github.com/fastmachinelearning/hls4ml/blob/main/hls4ml/model/layers.py
    b. In layers.py, you would have to define the attribute of the current layer.
    c. You'll also need to read the weight & bias from the model. The path & name of the weight & bias has to be exactly the same as in the saved model. You can use HDFview https://www.hdfgroup.org/downloads/hdfview/ to see the weight/bias path & name in the saved model.
    d. Last, you need to add the layer into the layer map https://github.com/fastmachinelearning/hls4ml/blob/main/hls4ml/model/layers.py#L1256
3. Adding a python template under this directory
    a. https://github.com/fastmachinelearning/hls4ml/tree/main/hls4ml/backends/vivado/passes
    b. In this file, you'll need to first define the config_template for all the needed layers. For example, if your layer contains both A and B layers, and the B layer contains C & D layers. you'll need to define the config_template for A,B,C & D layer. This config_template will generate the config for each layer, and these configs will be passed into the C++ functions (will be defined in the C++ file).
    c. After this file is implemented, you should be able to see all layer configs in the *parameter.h* file.

C++ File

1. Adding HLS functions under this directory

   a. https://github.com/fastmachinelearning/hls4ml/tree/main/hls4ml/templates/vivado/nnet_utils
   b. In this file, you'll need to transfer all needed functions from Keras API (python) to C++ to reach the functional correct behavior.
   c. You don't have to worry about the optimization at the beginning. Make sure the layer is working properly and then you can think about the optimization.

Example of adding a normalization layer in HLS4ML

First, find the normalization layer in Keras:

LayerNormalization layer (keras.io)

1. Read the description and make sure you fully understand the structure of the layer.
2. Test your understanding by building a custom layer in Keras. Give the same input to the Keras layer, and your custom layer. Make sure the output is the same.

   For example:

   1) Create a data set that needs to be fed into the model:

   ```
   #2batch*(3,4)
   x=x.reshape((2,3,4))
   x
   ```

   ```
   array([[[-0.9       , -0.82173913, -0.74347826, -0.66521739],
           [-0.58695652, -0.50869565, -0.43043478, -0.35217391],
           [-0.27391304, -0.19565217, -0.1173913 , -0.03913043]],

          [[ 0.03913043,  0.1173913 ,  0.19565217,  0.27391304],
           [ 0.35217391,  0.43043478,  0.50869565,  0.58695652],
           [ 0.66521739,  0.74347826,  0.82173913,  0.9       ]]])
   ```

   Here, we have 2 batches, each batch contains a set of data with dimensions 3,4.

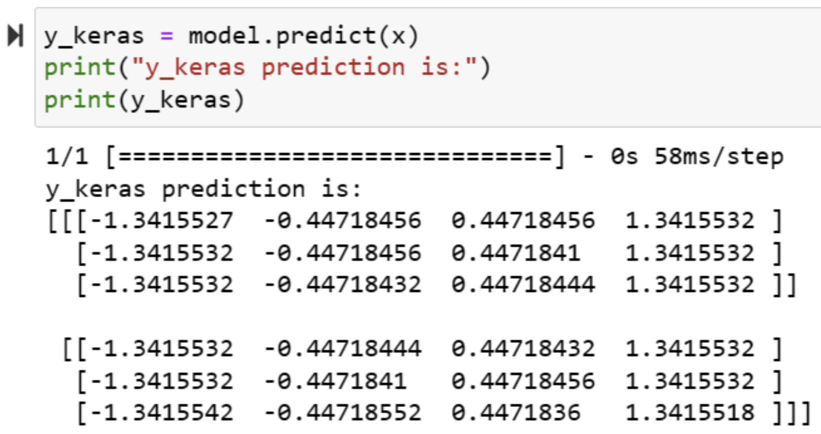
   2) Create a Keras Normalization layer:

   ```
   inputs = keras.Input(shape=x.shape[1:])
   ```

   ```
   output = tf.keras.layers.LayerNormalization(epsilon=1e-6)(inputs)
   model = keras.Model(inputs, output)
   ```

   Each time we will feed one batch to the model, so the input shape is (3,4). "3" is the sequence length, and "4" is the feature dimension.
   By default, LayerNormalization layer will normalize the data on the last axis.

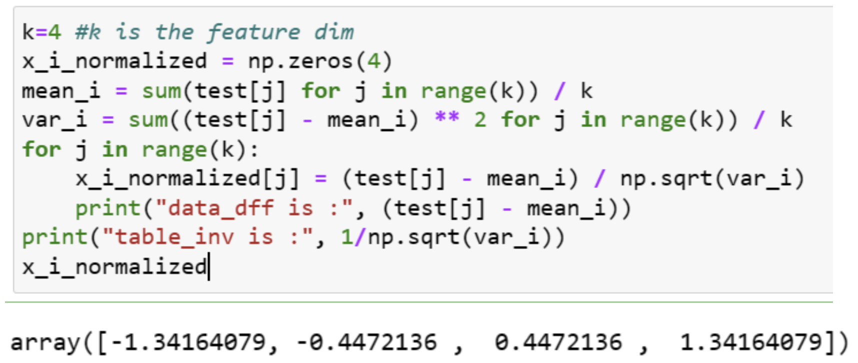
   3) Then, we can use keras prediction to find the output of the Keras model:



```
y_keras = model.predict(x)
print("y_keras prediction is:")
print(y_keras)
```

```
1/1 [==============================] - 0s 58ms/step
y_keras prediction is:
[[[-1.3415527  -0.44718456  0.44718456  1.3415532 ]
  [-1.3415532  -0.44718456  0.4471841   1.3415532 ]
  [-1.3415532  -0.44718432  0.44718444  1.3415532 ]]

 [[-1.3415532  -0.44718444  0.44718432  1.3415532 ]
  [-1.3415532  -0.4471841   0.44718456  1.3415532 ]
  [-1.3415542  -0.44718552  0.4471836   1.3415518 ]]]
```

4) Implement a function/custom layer to verify your understanding of the model:



```
k=4 #k is the feature dim
x_i_normalized = np.zeros(4)
mean_i = sum(test[j] for j in range(k)) / k
var_i = sum((test[j] - mean_i) ** 2 for j in range(k)) / k
for j in range(k):
    x_i_normalized[j] = (test[j] - mean_i) / np.sqrt(var_i)
    print("data_dff is :", (test[j] - mean_i))
print("table_inv is :", 1/np.sqrt(var_i))
x_i_normalized
```

```
array([-1.34164079, -0.4472136 ,  0.4472136 ,  1.34164079])
```

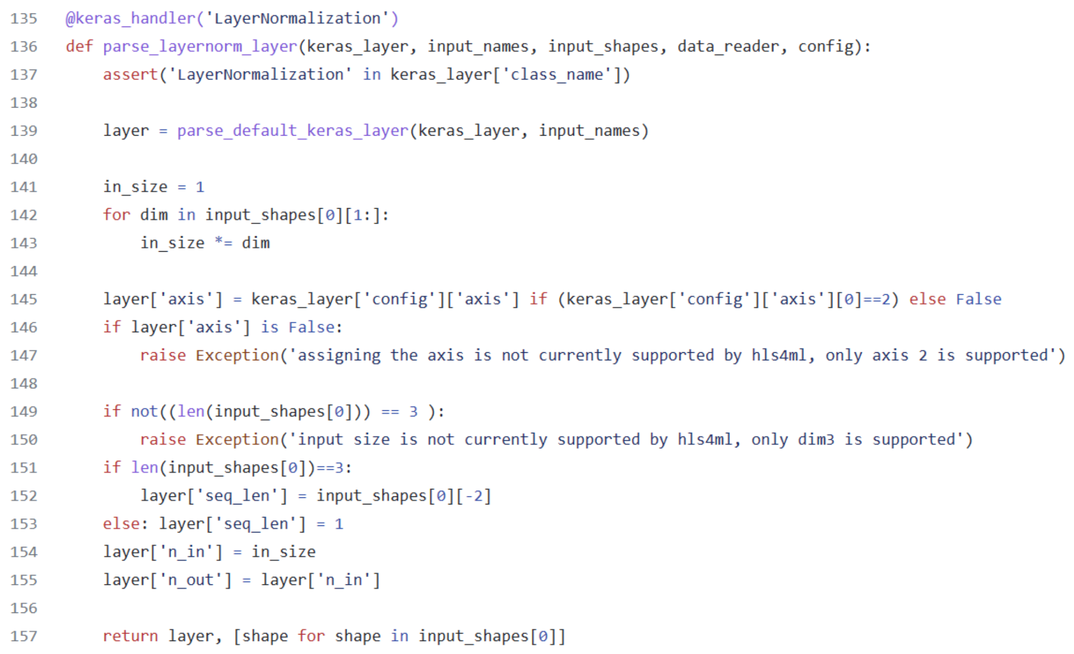
Here we verified that our understanding of the model matches the Keras output. Thus, we can start to implement the model in hls4ml.

3. Implement the Converter (Parser) in hls4ml

In hls4ml/hls4ml/converters/keras at main · fastmachinelearning/hls4ml · GitHub, we can find a lot of examples of the parser for each layer. As you can see, the task of the parser is to extract the configuration parameters from the layer, such as the number of inputs, the number of feature dimensions, the sequence length, the attention axis, etc…

The parser of each layer has a similar structure, with the inputs, such as `keras_layer, input_names, input_shapes, data_reader, config,` and outputs, such as `layer, [shape for shape in input_shapes[0]].`

The parser of the normalization layer:

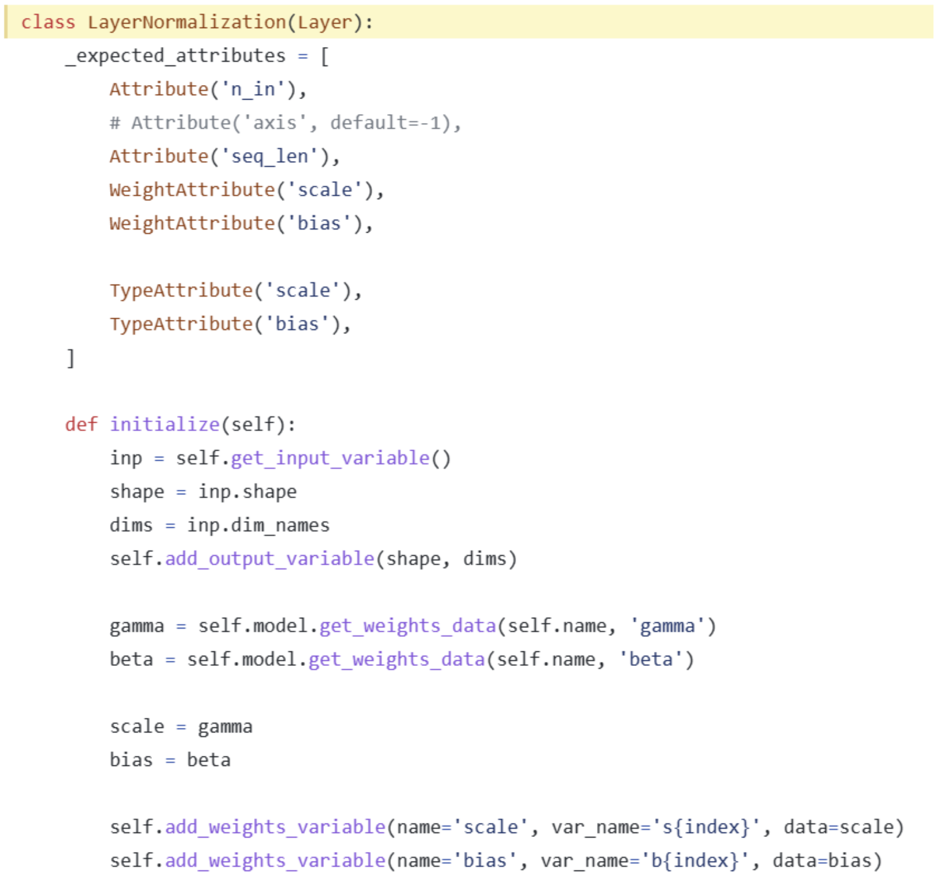


```python
@keras_handler('LayerNormalization')
def parse_layernorm_layer(keras_layer, input_names, input_shapes, data_reader, config):
    assert('LayerNormalization' in keras_layer['class_name'])

    layer = parse_default_keras_layer(keras_layer, input_names)

    in_size = 1
    for dim in input_shapes[0][1:]:
        in_size *= dim

    layer['axis'] = keras_layer['config']['axis'] if (keras_layer['config']['axis'][0]==2) else False
    if layer['axis'] is False:
        raise Exception('assigning the axis is not currently supported by hls4ml, only axis 2 is supported')

    if not((len(input_shapes[0])) == 3 ):
        raise Exception('input size is not currently supported by hls4ml, only dim3 is supported')
    if len(input_shapes[0])==3:
        layer['seq_len'] = input_shapes[0][-2]
    else: layer['seq_len'] = 1
    layer['n_in'] = in_size
    layer['n_out'] = layer['n_in']

    return layer, [shape for shape in input_shapes[0]]
```

When implementing the parser, we need to think about what parameters are needed for the layer. First, we need the number of the input for the layer. For the 1-d model, the input size would be the dimension of the last axis. However, when the model is 2-d or 3d, the input dimension would be the product of each axis' dimension (except batch). Second, We need to know which axis will be used to calculate the normalized value. If the axis is not to be considered supported in hls4ml, we need to raise an assertion. For the current hls4ml normalization layer, we only supported 2-d inputs with normalization on the second axis, so I made two assertions. Third, we need to find the number of output and sequence length.

4. In step 3, we load the configuration parameters into the hls4ml, but how about the weight parameters? In `hls4ml/model/layers.py`, we will load the weight/bias parameter into hls4ml, and store the previous configuration parameters into attributes.

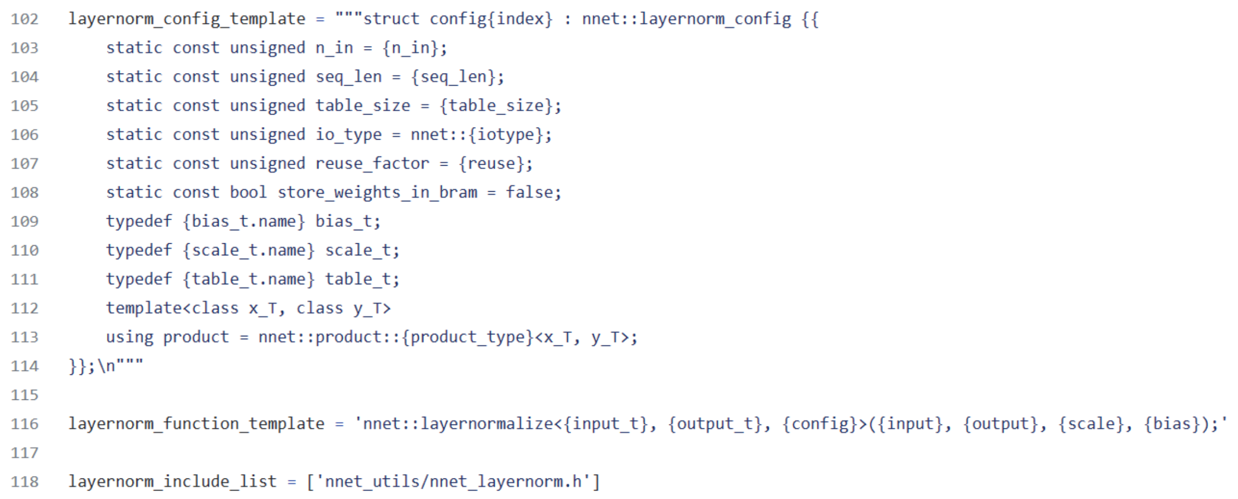
Normalization layer:



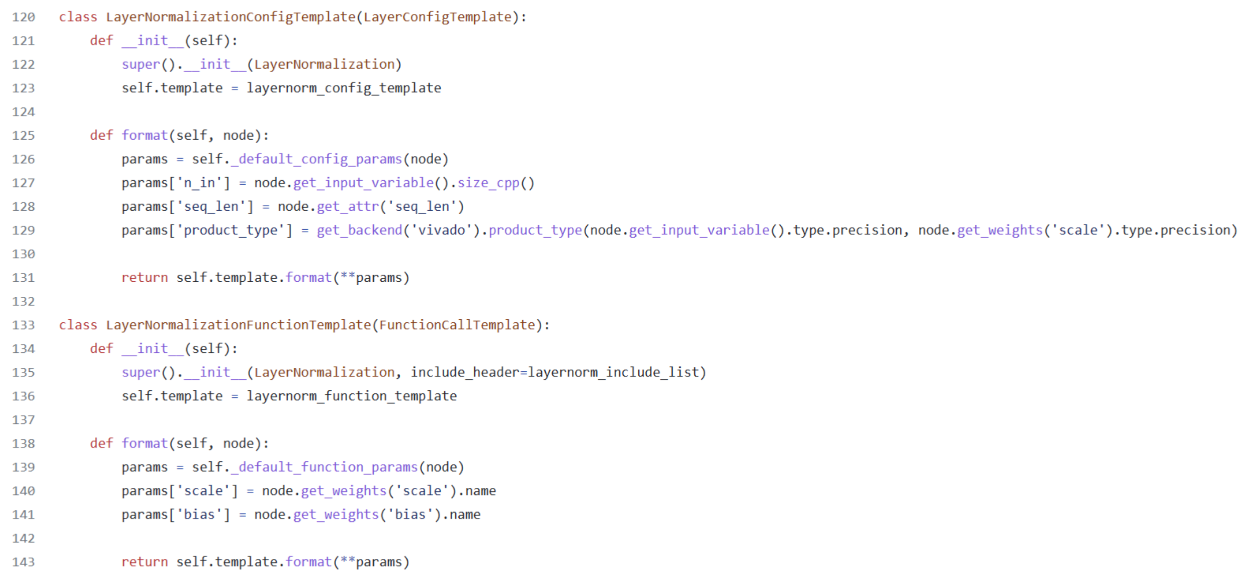
```
class LayerNormalization(Layer):
    _expected_attributes = [
        Attribute('n_in'),
        # Attribute('axis', default=-1),
        Attribute('seq_len'),
        WeightAttribute('scale'),
        WeightAttribute('bias'),

        TypeAttribute('scale'),
        TypeAttribute('bias'),
    ]

    def initialize(self):
        inp = self.get_input_variable()
        shape = inp.shape
        dims = inp.dim_names
        self.add_output_variable(shape, dims)

        gamma = self.model.get_weights_data(self.name, 'gamma')
        beta = self.model.get_weights_data(self.name, 'beta')

        scale = gamma
        bias = beta

        self.add_weights_variable(name='scale', var_name='s{index}', data=scale)
        self.add_weights_variable(name='bias', var_name='b{index}', data=bias)
```

In the class LayerNormalization(Layer), we create expected attributes for “number of input”, and “sequence length”. For the weights, we need to create “WeightAttributes”, the naming of the weight attribute needs to be well described for the data.
We then define a class function “initialize(self)” to initialize the weight parameters. To load the weight, we can use the function “`get_weights_data()`”, or “`add_weights_variable()`”. After we load the weight, we need to store the weight using the function “`add_weights_variable()`”. We also need to use “`self.add_output_variable(shape, dims)`” to add the output shape and name of each dimension.
The last part of this procedure is to add this new layer to the “`layer_map`” dictionary.

5. Add the new supported layer in `hls4ml/utils/config.py`. However, the most updated version of hls4ml may change this supported layer to other places.
6. Create a template configuration in hls4ml/hls4ml/backends/vivado/passes
   In the previous steps, we have defined the configuration parameters, and now we will create a file that can help us to convert the configuration into c++ code.



```
102  layernorm_config_template = """struct config{index} : nnet::layernorm_config {{
103      static const unsigned n_in = {n_in};
104      static const unsigned seq_len = {seq_len};
105      static const unsigned table_size = {table_size};
106      static const unsigned io_type = nnet::{iotype};
107      static const unsigned reuse_factor = {reuse};
108      static const bool store_weights_in_bram = false;
109      typedef {bias_t.name} bias_t;
110      typedef {scale_t.name} scale_t;
111      typedef {table_t.name} table_t;
112      template<class x_T, class y_T>
113      using product = nnet::product::{product_type}<x_T, y_T>;
114  }};\n"""
115
116  layernorm_function_template = 'nnet::layernormalize<{input_t}, {output_t}, {config}>({input}, {output}, {scale}, {bias});'
117
118  layernorm_include_list = ['nnet_utils/nnet_layernorm.h']
```



```
120  class LayerNormalizationConfigTemplate(LayerConfigTemplate):
121      def __init__(self):
122          super().__init__(LayerNormalization)
123          self.template = layernorm_config_template
124
125      def format(self, node):
126          params = self._default_config_params(node)
127          params['n_in'] = node.get_input_variable().size_cpp()
128          params['seq_len'] = node.get_attr('seq_len')
129          params['product_type'] = get_backend('vivado').product_type(node.get_input_variable().type.precision, node.get_weights('scale').type.precision)
130
131          return self.template.format(**params)
132
133  class LayerNormalizationFunctionTemplate(FunctionCallTemplate):
134      def __init__(self):
135          super().__init__(LayerNormalization, include_header=layernorm_include_list)
136          self.template = layernorm_function_template
137
138      def format(self, node):
139          params = self._default_function_params(node)
140          params['scale'] = node.get_weights('scale').name
141          params['bias'] = node.get_weights('bias').name
142
143          return self.template.format(**params)
```

https://github.com/Ethan0Jiang/hls4ml/blob/66003e7d1b49888dfbaccc4d3399a8100012477d/hls4ml/backends/vivado/passes/core_templates.py#L120
From line 102 to 114, we create the config of the normalization layer that will be written into a config.h file. In this config template, the element in the {curly brackets} will be replaced by the configuration parameter, using the class `LayerNormalization ConfigTemplate()`. The `LayerNormalizationConfigTemplate` class has a `format` function to replace the argument in the template, which is a function that you need to implement. We also need to create a "`LayerNormalizationFunctionTemplate()`" class to define how we can call the normalization layer in the top .cpp file.
*Don't forget to add your new layer into the import line at the top of the file.

7. (Optional) `hls4ml/backends/vivado/vivado_backend.py`
   Sometimes, you might need to implement an optimizer for your layer.
   In the optimizer, you can try to reshape, or transpose your weight, or you can also set a default value for the parameters.

```python
@layer_optimizer(LayerNormalization)
def init_layernormalization(self, layer):
    if 'table_t' not in layer.attributes:
        layer.set_attr('table_t', NamedType(name=layer.name + '_table_t', precision=FixedPrecisionType(width=32, integer=5)))
    if 'table_size' not in layer.attributes:
        layer.set_attr('table_size', 2048)
```

Here, I create an attribute of table_size and table_t with some default value for the lookup table of nonlinear functions.
*don't forget to add your layer into import at the top of the file.

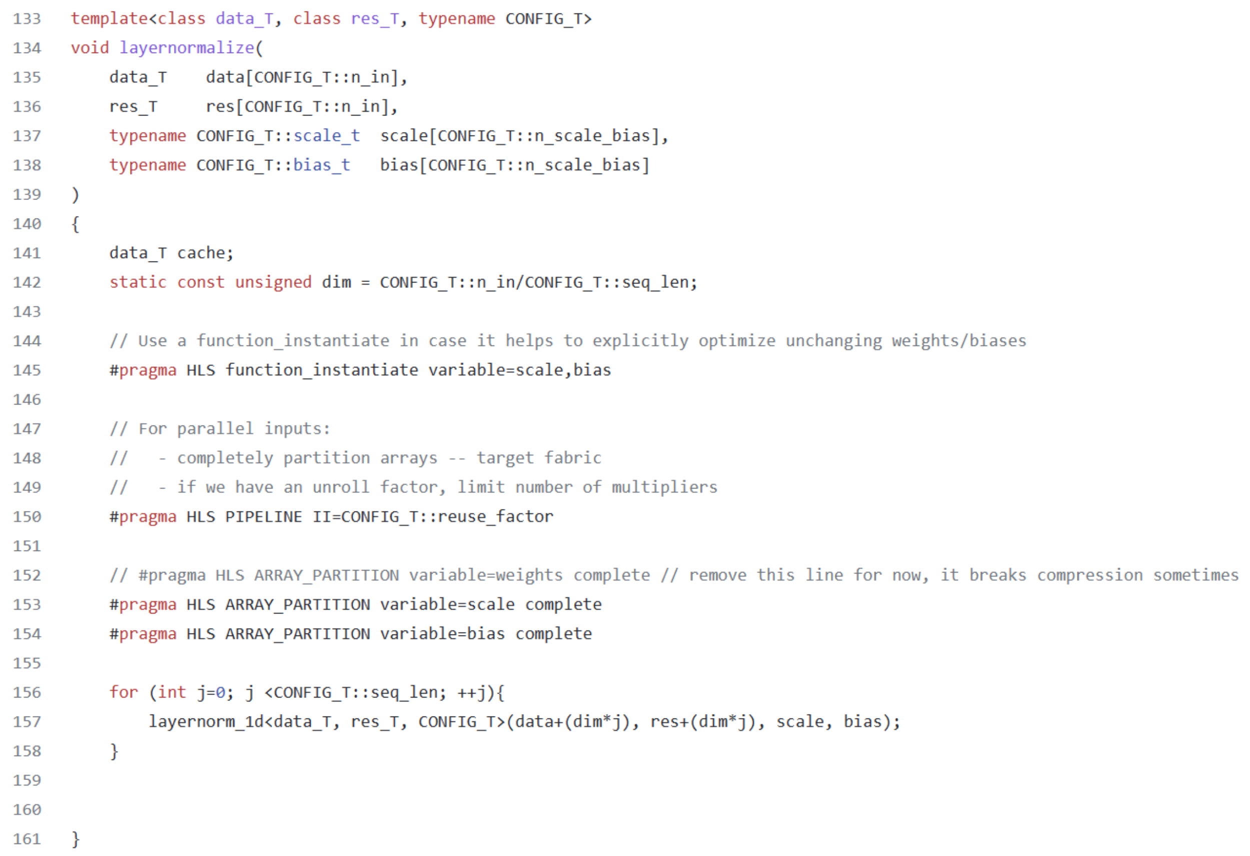
8. (Optional) resource_strategy
   If you are implementing the resource strategy, you have to transpose the weight shape of the dense layer in `hls4ml/backends/vivado/passes/resource_strategy.py`

```python
def transform(self, model, node):
    if isinstance(node, Dense):
        node.weights['weight'].data = np.transpose(node.weights['weight'].data)
    elif isinstance(node, Conv1D):
        node.weights['weight'].data = np.transpose(node.weights['weight'].data, axes=[2, 0, 1]) #(W,C,F) => (F,W,C)
    elif isinstance(node, SeparableConv1D):
        node.weights['depthwise'].data = np.transpose(node.weights['depthwise'].data, axes=[2, 0, 1]) #(W,C,F) => (F,W,C)
        node.weights['pointwise'].data = np.transpose(node.weights['pointwise'].data, axes=[2, 0, 1]) #(W,C,F) => (F,W,C)
    elif isinstance(node, Conv2D):
        node.weights['weight'].data = np.transpose(node.weights['weight'].data, axes=[3, 0, 1, 2]) #(H,W,C,F) => (F,H,W,C)
    elif isinstance(node, SeparableConv2D):
        node.weights['depthwise'].data = np.transpose(node.weights['depthwise'].data, axes=[3, 0, 1, 2]) #(H,W,C,F) => (F,H,W,C)
        node.weights['pointwise'].data = np.transpose(node.weights['pointwise'].data, axes=[3, 0, 1, 2]) #(H,W,C,F) => (F,H,W,C)
    elif isinstance(node, (LSTM, GRU)):
        node.weights['weight'].data = np.transpose(node.weights['weight'].data)
        node.weights['recurrent_weight'].data = np.transpose(node.weights['recurrent_weight'].data)
    elif isinstance(node, (MultiHeadAttention)):
        node.weights['key_weight'].data   = np.transpose(node.weights['key_weight'].data,   axes=[0, 2, 1])
        node.weights['query_weight'].data = np.transpose(node.weights['query_weight'].data, axes=[0, 2, 1])
        node.weights['value_weight'].data = np.transpose(node.weights['value_weight'].data, axes=[0, 2, 1])
        node.weights['attention_output_weight'].data = np.transpose(node.weights['attention_output_weight'].data, axes=[2, 0, 1])
    else:
        raise Exception('Unexpected layer {} with resource strategy'.format(node.class_name))

    node.set_attr('_weights_transposed', True)
```

   *don't forget to add your layer into import at the top of the file.

9. Header file of your layer.
   1) Take a look at the HLS code of different layers in “hls4ml/templates/vivado/nnet_utils/” , such as dense layer, activation layer, mutiheadAttention layer, etc…
   2) Do some research about C++ templates online.

3) Structure of a layer:



```
template<class data_T, class res_T, typename CONFIG_T>
void layernormalize(
    data_T    data[CONFIG_T::n_in],
    res_T     res[CONFIG_T::n_in],
    typename CONFIG_T::scale_t  scale[CONFIG_T::n_scale_bias],
    typename CONFIG_T::bias_t   bias[CONFIG_T::n_scale_bias]
)
{
    data_T cache;
    static const unsigned dim = CONFIG_T::n_in/CONFIG_T::seq_len;

    // Use a function_instantiate in case it helps to explicitly optimize unchanging weights/biases
    #pragma HLS function_instantiate variable=scale,bias

    // For parallel inputs:
    //   - completely partition arrays -- target fabric
    //   - if we have an unroll factor, limit number of multipliers
    #pragma HLS PIPELINE II=CONFIG_T::reuse_factor

    // #pragma HLS ARRAY_PARTITION variable=weights complete // remove this line for now, it breaks compression sometimes
    #pragma HLS ARRAY_PARTITION variable=scale complete
    #pragma HLS ARRAY_PARTITION variable=bias complete

    for (int j=0; j <CONFIG_T::seq_len; ++j){
        layernorm_1d<data_T, res_T, CONFIG_T>(data+(dim*j), res+(dim*j), scale, bias);
    }


}
```

We will define a c++ template for the layer. The template has three type parameter,
`<class data_T, class res_T, typename CONFIG_T>`
The data_T is the input data type, by default it would be ap_fix<16,6>, the res_T is the output dataset, and the typename CONFIG_T is a struct that we defined in step6.
Usually there is one input and one output for the layer, and two weights. The input/output variables have been defined at step6, the FunctionTemplet.
First, we do not have to implement the detailed structure of the layer, but make sure the code from step1 to step8 can pass through the hls4ml workflow. To test that, write some very simple operation in the HLS code, for example, result = input +1. Then run the hls.compile() and hls.build() to make sure that there are no bugs and errors. If bugs occur, check the report in the prompt and try to fix it.
Once we verify the code can pass through the flow with no error, we can start to implement the layer in HLS code.
A few good resource and hint of writing the HLS code:

1) HLS Pragmas (xilinx.com) Understand Pragma Reshape, `ARRAY_PARTITION` , unroll, pipeline, dataFlow, instantiate, those are the most common Pragma to use.
2) Check Hans' tutorial of how to write good HLS code. Vivado HLS – Hans Giesen (upenn.edu)
3) Understand two different i/o strategy in HLS, parallel vs. hls_stream

Once finish implementing the layer, we can start to debug the layer

Custom Layer Using Extension API

In pull request 528 (https://github.com/fastmachinelearning/hls4ml/pull/528), HLS4ML adds an extension API that allows users to add custom layers into the package. The example can be found in hls4ml/test/pytest/test_extensions.py
https://github.com/fastmachinelearning/hls4ml/blob/main/test/pytest/test_extensions.py

If you want to add special layers such gaussian sampling layer, you might need to encapsulate it as a Keras layer first, an example can be found here
https://keras.io/guides/making_new_layers_and_models_via_subclassing/