From keras to SOFIE: summer internship report

Uri Stern

September 8, 2023

1 Introduction

1.1 Problem Presentation

SOFIE is a system designed to allow ROOT users to run machine learning models, including neural networks, using an automatically created fast c++ code (instead of the regular and slower python option). In principle, for neural networks SOFIE works in the following manner:

1. Given an input model, SOFIE parses the model into an intermediate class called "RModel". The RModel is meant to hold all the information required to run the model: the input/output shape, all the operations done on the input (and their order), and the different layers' weights. The operations are saved in unique ROperator classes that hold all the information required for the different operations.

2. Using the RModel and the ROperator, SOFIE automatically generates the c++ inference code that can simply be compiled, either from c++ or python, and run by the user without any changes (or dependencies, other than BLAS for matrix multiplication).

A schema of this workflow is presented in Fig 1. In general, SOFIE is designed to be used with models saved in a generic format of ONNX, which makes parsing of models trained with ROOT very easy (and also for PyTorch, as the conversion between the two is relatively simple). However, this is not the case for models trained with keras library, as (i) there is no simple converter between keras and onnx, and (ii) keras supports some functions that onnx does not (for example the Swish activation function, which is used in the Atlas experiment).

Due to this problem, to enable users who train their models with keras to use SOFIE, a new parser is needed, other than the existing parsers for onnx format models. Such a parser exists in SOFIE, but it is very limited, as (i) it only supports dense and convolutional layers (along with their activation functions), but not pooling and recurrent layers, which are required for advanced models as RNN and CNN; and (ii) it is written in C++, while most of the parsing is actually done in python (due to the model being written in python), which makes future developments of the parser quite complex.

1.2 My Project

To answer the challenges described earlier, my project was to write a parser from neural networks trained with keras to RModel, that will be (i) written in python (ii) able to parse complex layers (iii)

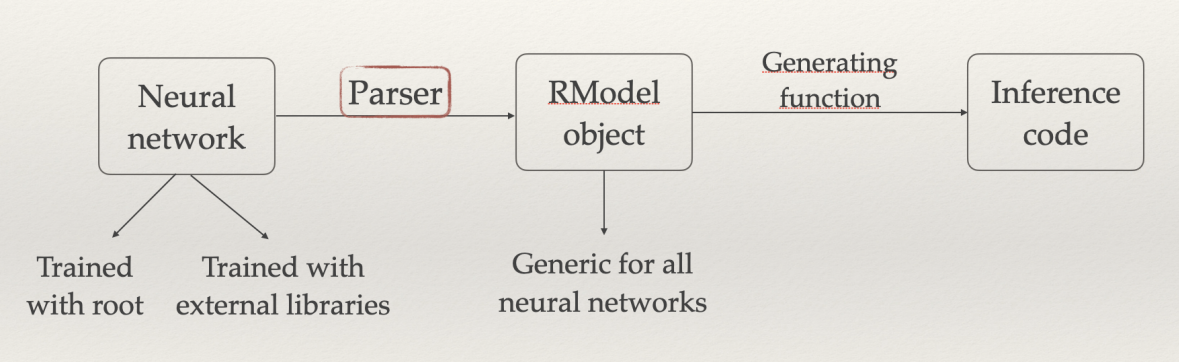

Figure 1: the workflow in SOFIE

layer type	supported layers
linear functions	mul, add, sub
non linear functions	sigmoid, softamx, tanh, relu, selu, leaky relu, swish
hidden layers	dense, batch-normalization, convolution, pooling, simple-rnn, gru, lstm

Table 1: An example table.

```
modelFile = "CNNtest.h5"
rmodel = Keras_Parser_into_RModel(modelFile)
generatedHeaderFile = modelFile.replace(".h5",".hxx")
print("Generating inference code for the Keras model from ",modelFile,"in the header ", generatedHeaderFile)
rmodel.Generate()
rmodel.OutputGenerated(generatedHeaderFile)
print("compiling SOFIE model ", modelName)
ret = ROOT.gInterpreter.Declare('#include "' + generatedHeaderFile + '"')
if not ret:
    print("Error compiling header file ", generatedHeaderFile)
    exit()
session = ROOT.TMVA_SOFIE_CNNtest.Session()
input_test = np.ones((1,256), dtype = 'float32')
result = session.infer(input_test)
keras_model = keras.models.load_model(modelFile)
keras_model.load_weights(modelFile)
print("fraction of equal elements in the results vector = {}%".format(100*accuracy))
```

Figure 2: comparing between keras model at inference time and the c++ generated code for a convolutional neural network with various operators (conv, pool, dense, flatten, reshape, activation functions).

easy to expand in the future. The full project can be found in:
https://github.com/uristern123/root/tree/parser/bindings/
pyroot/pythonizations/python/ROOT/_pythonization/_tmva/_rmodel_keras.py

2 Parser Design and Implementation

2.1 Parser Workflow

Our parse works in three stages:

1. **Create the operators list**: for each layer of the model, we extract the relevant information of the layer - input/output names, attributes of the layer, data type, etc. After extracting all the relevant information, we create a c++ ROperator object from the relevant class, which is returned to the main function of the parser and added to the RModel using a call to the c++ function *AddOperator* of the RModel class.

2. **Create the weights list**: for each layer that has weights, we add the weights and their shape to the weights list of the model via the c++ function *AddInitilializedTensor* of the RModel class, called from our python code.

3. **extracting the input and output info**: lastly, we pass to the RModel the input and output info (name, shape) of the inference using the c++ functions *AddInputTensorInfo*, *AddInputTensorName* and *AddOutputTensorNameList* of the RModel class.

2.2 Tests and Results

Currently, the parser can parse several activation functions, linear operations, and most importantly, dense, convolutional, pooling and recurrent layers. A full list can be found in Table 1.

To test our parser, I created many networks with different layers and compared the outputs given by keras and the generated c++ code by SOFIE. In all of the tests, the outputs are the same, which ensures the safety of using the new parser.

The tests can be found in https://github.com/uristern123/root/tree/parser/tmva/sofie/test/KerasParserTest/. An example code for such test can be seen in Fig 2

```
def MakeKerasConv(layer):
    finput = layer['layerInput']
    foutput = layer['layerOutput']
    fLayerDType = layer['layerDType']
    fLayerInputName = finput[0]
    fLayerOutputName = foutput[0]
    attributes = layer['layerAttributes']
    fWeightNames = layer["layerWeight"]
    fKernelName = fWeightNames[0]
    fBiasName = fWeightNames[1]
    fAttrDilations = attributes["dilation_rate"]
    fAttrGroup = int(attributes["groups"])
    fAttrKernelShape = attributes["kernel_size"]
    fKerasPadding = str(attributes["padding"])
    fAttrStrides = attributes["strides"]

    if fKerasPadding == 'valid':
        fAttrAutopad = 'VALID'
    elif fKerasPadding == 'same':
        fAttrAutopad = 'NOTSET'
        fInputShape = attributes['_build_input_shape']
        inputHeight = fInputShape[1]
        inputWidth = fInputShape[2]
        outputHeight = math.ceil(float(inputHeight) / float(fAttrStrides[0]))
        outputWidth = math.ceil(float(inputWidth) / float(fAttrStrides[1]))
        padding_height = max((outputHeight - 1) * fAttrStrides[0] + fAttrKernelShape[0] - inputHeight, 0)
        padding_width = max((outputWidth - 1) * fAttrStrides[1] + fAttrKernelShape[1] - inputWidth, 0)
        padding_top = math.floor(padding_height / 2)
        padding_bottom = padding_height - padding_top
        padding_left = math.floor(padding_width / 2)
        padding_right = padding_width - padding_left
        fAttrPads = [padding_top, padding_bottom, padding_left, padding_right]
    else:
        raise RuntimeError(
            "TMVA::SOFIE - RModel Keras Parser doesn't yet supports Convolution layer with padding " + fKerasPadding
        )
    if TMVA.Experimental.SOFIE.ConvertStringToType(fLayerDType) == ROOT.TMVA.Experimental.SOFIE.ETensorType.FLOAT:
        op = ROOT.TMVA.Experimental.SOFIE.ROperator_Conv['float'](fAttrAutopad, fAttrDilations, fAttrGroup,
                                                                  fAttrKernelShape, fAttrPads, fAttrStrides,
                                                                  fLayerInputName, fKernelName, fBiasName,
                                                                  fLayerOutputName)

        return op
    else:
        raise RuntimeError(
            "TMVA::SOFIE - Unsupported - Operator Gemm does not yet support input type " + fLayerDType
        )
```

Figure 3: function for parsing convolutional layers.

2.3 Parsing Examples

Fig 3 presents an example of convolutional layer parsing. In general, the function extracts general attributes (such as input/output name), and then extracts unique attributes of such layer (for example, the padding dimensions). After all the necessary details are extracted from the layer, an object of class ROperator_Conv is created, which is later added to the RModel object (unseen in the figure). Fig 4 shows the resulting c++ generated inference function for a model with one dense layer and relu activation function.

3 Challenges and Solutions

3.1 Format Inconsistencies Between Keras and Onnx

A large challenge challenge in parsing keras models into RModel is that the RModel class expects, and work with, operators that have onnx format. For some layers, such as convolution, pooling and recurrent layers, this raises an issue - since keras and onnx differ in some of their attributes in those models, such as the input shape of the convolutional layers or the weights data structure of the recurrent layers. In the recurrent layers, the solution is simple: we simply transposed the weight matrix before adding its data to the RModel, and change the order of the weights when necessary. However, in the convolution and pooling layers the solution is more complex, and demands transposing the input in runtime before and after the layers to match the correct format.

```
std::vector<float> infer(float* tensor_dense10input){

//--------- Gemm
   char op_0_transA = 'n';
   char op_0_transB = 'n';
   int op_0_m = 1;
   int op_0_n = 64;
   int op_0_k = 7;
   float op_0_alpha = 1;
   float op_0_beta = 1;
   int op_0_lda = 7;
   int op_0_ldb = 64;
   std::copy(tensor_dense10bias0bcast, tensor_dense10bias0bcast + 64, tensor_dense10Dense);
   BLAS::sgemm_(&op_0_transB, &op_0_transA, &op_0_n, &op_0_m, &op_0_k, &op_0_alpha, tensor_dense10kernel0, &op_0_ldb,
tensor_dense10input, &op_0_lda, &op_0_beta, tensor_dense10Dense, &op_0_n);

//------ RELU
   for (int id = 0; id < 64 ; id++){
      tensor_dense10Relu0[id] = ((tensor_dense10Dense[id] > 0 )? tensor_dense10Dense[id] : 0);
   }
   std::vector<float> ret (tensor_dense10Relu0, tensor_dense10Relu0 + 64);
   return ret;
}
};
} //TMVA_SOFIE_Relutest
```

Figure 4: generated inference function in c++ for dense layer with relu activation function.

3.2 Missing Attributes in Keras Models

While keras implemented most of the attributes of the onnx operators, some attributes are not implemented (such as peepholes in lstm) or predetermined by the library, without giving the user option to change it. Such cases had to be found and values had to be defined for them, to properly create the ROperators (that expect those attributes).

3.3 Working With C++ From Python

As mentioned before, our code uses existing c++ classes and methods to create the RModel. This, however, can cause issues when special pointers (unique and shared pointers) are requested by the c++ method, as this translation cannot be easily done from python. To solve this issue, we simply created a c++ function that receives a regular pointer of the same object and translated it to a unique/shared pointer, followed by a call to the relevant function.

Additionally, while working with the c++ code we found some minor bugs in some functions, which were fixed.

4 Conclusion and Future Work

To conclude, In this project we created a python function that parses neural networks trained with keras library into the RModel class in SOFIE, which is used to automatically create c++ inference code for those models. The parser is tested and safe to use, as it provides the same results as inference done with the keras library.

In the future, the parser's easy structure and explainability allows future additions of additional layers (such as the attention layer), in a simple manner.

On a personal note, I would like to thank Lorenzo Moneta for his kind supervision of this project, and for giving me the opportunity to lead it.

For any question/comments, I am available at ustern@gmail.com.