



## **Summer Student Project Report**

# **TMVA SOFIE**

**Enhancing the Machine Learning Inference Engine**

**By**

*Sanjiban Sengupta, IIIT Bhubaneswar*

**under the supervision of**

*Lorenzo Moneta, CERN*



# Abstract

---

ROOT<sup>[4]</sup> is primarily used for high-scale data processing and analysis required for the research of high-energy physics. It provides various tools, utilities, and facilities for statistics, algebra, visualization, multi-varied methods, data serialization, parallelized data computation, etc.

With the advent of machine learning, it is nowadays necessary to use regression and classification methods for the determination and analysis of physical events. Consequently, ROOT offers native support for various supervised learning techniques and easy interoperability with commonly used machine learning libraries through its module called the Toolkit of Multi-varied Analysis, or TMVA<sup>[5]</sup>.

In the recent developments in TMVA, research on machine learning inference was being made for developing a Fast Machine Learning Inference Engine called SOFIE<sup>[1]</sup>. SOFIE which stands for System for Optimized Fast Inference code Emit was launched in 2021, and since then active work has been going on to improve its capabilities. SOFIE is based on ONNX<sup>[7]</sup> standards and presents a system that generates inference code with the least latency and few dependencies, suitable for applications in high-energy physics. SOFIE has support for parsing models developed in ONNX, Keras, or PyTorch framework.

On this project of enhancing the inference engine, rigorous work was done on strengthening the Keras parser, designing and developing the support for a custom operator, and implementing the functionality for parsing and inference of graph neural networks in SOFIE.

# Contents

1. Introduction	4
a. Motivation	
b. TMVA SOFIE	
2. Enhancing the Keras Parser	10
a. Introduction	
b. Implemented Support	
3. SOFIE Custom Operator: Design & Development	14
a. Motivation	
b. Definition	
c. Interface	
d. Implementation	
4. SOFIE Graph Neural Network	18
a. Motivation	
b. Design	
i. Architecture	
ii. Intermediate Representation	
1. RModel_GNN	
2. RModel_GraphIndependent	
3. RModel_GNNStack	
4. RFunctions	
a. RFunction_Update	
b. RFunction_Aggregate	
iii. Graph Data	
c. Inference	
i. RModel_GNN	
ii. RModel_GraphIndependent	
d. Parsing	
5. Future Developments	28
6. Conclusion	29
7. Acknowledgment	30
8. References	31

# Chapter 1

---

## Introduction

## 1.a) Motivation

There has been a lot of research on Machine Learning modeling and training. Even today, the majority of machine-learning research is primarily focused on the development of sophisticated machine-learning models capable of handling complex data and finding intricate patterns. Research on Machine Learning inference is often neglected, and basic standards are still followed for the deployment and inference of models once it is trained.

We do have popular machine learning frameworks like TensorFlow and PyTorch which provides a great range of utilities for machine learning inference. However, these are only usable on models trained on their platforms and do not follow a universal standardized approach. Furthermore, their integration with a C++ build is difficult, hard to maintain, and has a lot of dependencies that harm portability.

Originally authored by Facebook and Microsoft, ONNX<sup>[7]</sup> which stands for Open Neural Network Exchange is an open-sourced artificial intelligence ecosystem that provides standards for representing a machine learning model and algorithms for their inference in a universally standardized way. Microsoft also developed ONNXRuntime<sup>[9]</sup> which provides the environment required for the inference of an ONNX model of a particular machine learning method. However, it is not very beneficial in the case of high-energy physics applications. To begin with, it has a high dependency on different packages. Furthermore, it is difficult to integrate with high-energy physics applications because of the supposed control of libraries and threads, and the inability to handle single-event evaluation.

TMVA SOFIE thus tries to solve the problems by developing a sophisticated yet simple-to-use architecture, which is based on ONNX standards and supports C++ and Python, for fast machine learning inference. It provides a standard utility to parse an ONNX model, a Keras model, or a PyTorch model and translate it to its own internal representation and then generates the

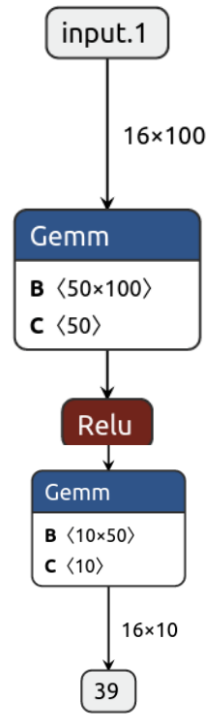
inference code with few dependencies, and can be easily used and maintained.

## 1.b) TMVA SOFIE

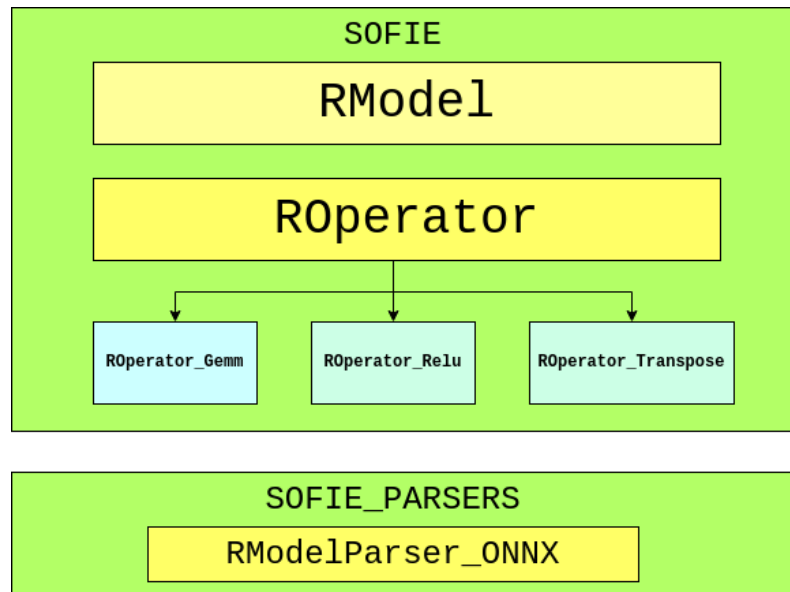
System for Fast Inference code Emit, or SOFIE, is a fast machine-learning inference engine, which provides:

- i. An intermediate representation based on ONNX standards, tailored for specific requirements of various models.
- ii. Inference Code Generator, which generates C++ header files containing easily invocable infer functions with the least latency and few dependencies. The infer function can be used in a plug-and-play format by just including the header file in the code under development.
- iii. A serialization system for saving the model architecture and weights data in `.root` files, which uses compression based on gzip algorithm, thus saving a lot of space in comparison to Keras, PyTorch, or even ONNX.
- iv. Parsers for translating ONNX, Keras, and PyTorch models into their internal representation, which can then be used for generating the inference code. Thus supporting three of the most popularly used machine learning frameworks.

The first edition of SOFIE comprised an `RModel` class, capable of storing the model architecture, and weights. Along with various `ROperators` which defined the operations needed to be performed on any input data to produce a transformed output. Model structure in SOFIE basically comprises a model graph, where tensors flow through various operators which transform them and in the end, are released as the output of the model. Operators are defined by the ONNX standards, and has the required attributes on data orientation, augmentation, and updating, and may have weights if required (for example, the `Gemm` operator which has weights & biases for its matrix transformations).

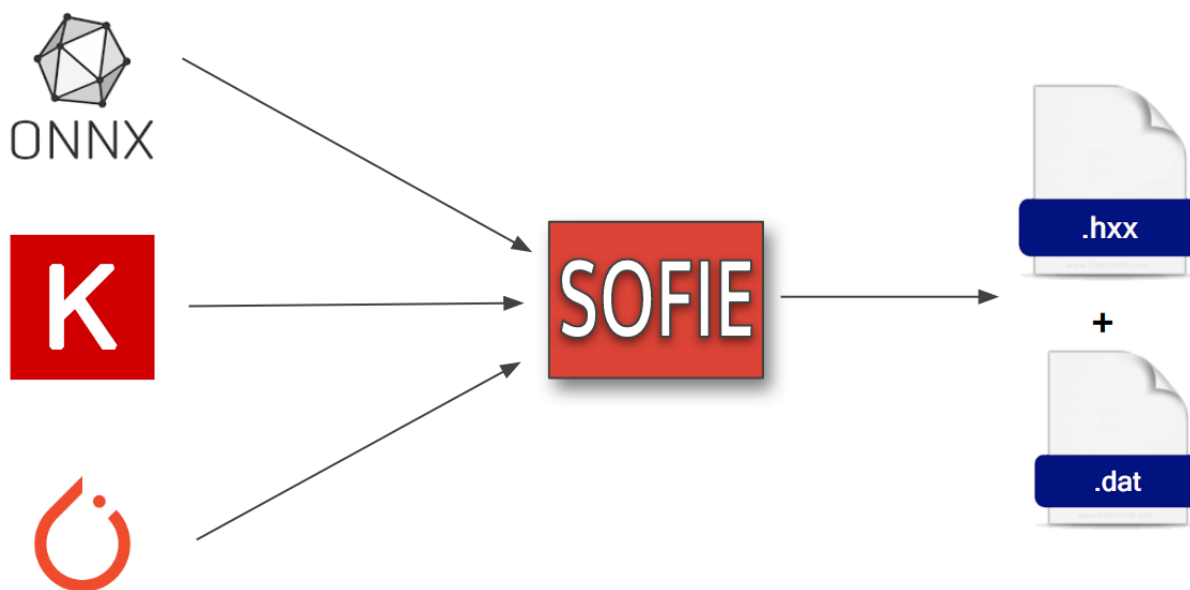


*Fig. 1: A Model graph representing the SOFIE internal representation. The graph is necessarily directed, where inference starts with the input tensors. They flow through the operators as intermediate tensors and are then released as output tensors from the last operator.*



*Fig. 2: The first architecture of SOFIE containing an RModel class, ROperators for Gemm, Relu, and Transpose operations, and an ONNX parser.*

The SOFIE project started with only 3 operators and the ONNX parser, and now it has support for over 30 operators, with new operators already under development and planned. SOFIE now has a Keras and PyTorch parser capable of translating the `.h5` and `.pt` models. Frequent contributions of SOFIE are keeping it updated with the recent changes in the external frameworks, with rigorous development efforts of adding support for new and complex operators.



*Fig. 3: Current architecture of SOFIE. It now can parse models from Keras and PyTorch and produces a .hxx C++ header file with a .dat file for the weights of the machine learning model*

SOFIE's RModel can be instantiated using the parsers after translating an ONNX, a Keras, or a PyTorch file.

```

using namespace TMVA::Experimental::SOFIE;
RModelParser_ONNX parser;
RModel model = parser.Parse("model.onnx");
  
```

And then it can be used to generate the inference code,

```

// generate inference code
model.Generate();
// output header file and weight file
model.OutputGenerated();
  
```

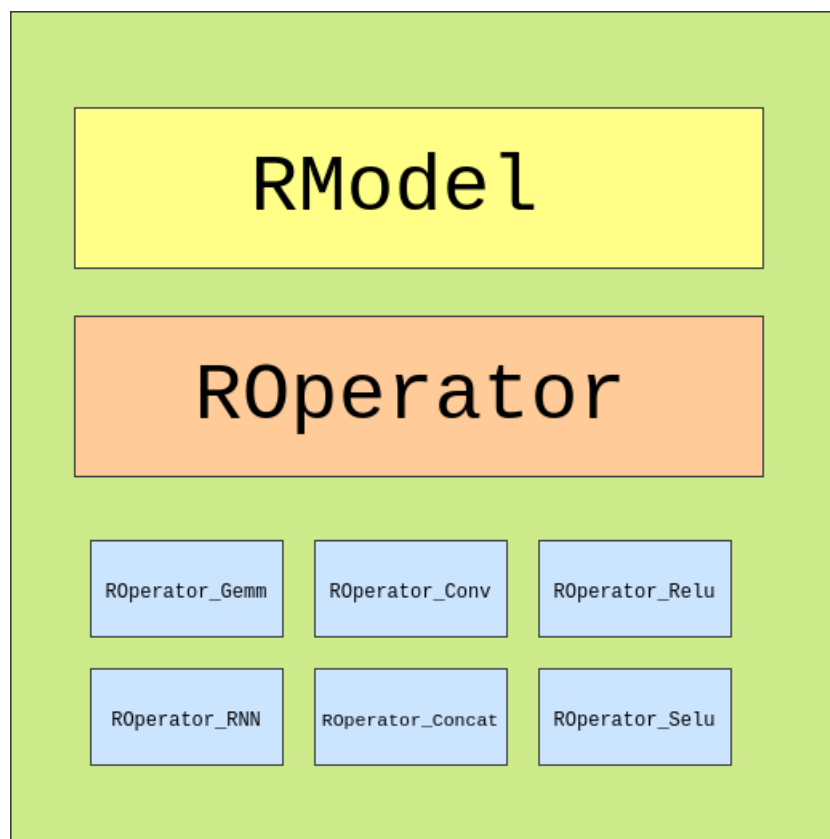


Or, can be stored in memory for future use,

```
TFile file("model.root","CREATE");  
SOFIE::RModel model = Parser.Parse("./example_model.onnx");  
model.Write("model");  
file.Close();
```

And can be similarly read later to generate inference code if required using ROOT I/O mechanisms,

```
TFile file("model.root","READ");  
using namespace TMVA::Experimental;  
SOFIE::RModel *model;  
file.GetObject("model",model);  
file.Close();
```



*Fig 4: SOFIE currently supports over 30 operators on Linear transformations, Convolution, Recurrent Neural Networks, etc.*

# Chapter 2

---

## **Enhancing the Keras Parser**

## 2.a) Introduction

Keras<sup>[10]</sup> is an open-source framework providing an environment for building, training, testing, and evaluating artificial neural networks by acting as an interface for TensorFlow. Keras is capable of modeling sequential and modular neural network objects and serializing them into `.h5` files. Being a very particular framework, it only supports its own format and modeling for inference. Unlike PyTorch, Keras does not support the translation to ONNX format natively thus not supporting the standardization of the model graph and its underlying computations.

ONNX launched a TensorFlow-to-ONNX converter, called TF2ONNX<sup>[11]</sup>, which is built for converting a TensorFlow model to an ONNX model. Even with this support, the Keras models are not easily translatable to ONNX files. For instance, to use TF2ONNX, the user is required to specify the opset version, input and output tensor names, etc.

SOFIE Keras Parser is an easy-to-use converter that is capable of translating a Keras `.h5` file to the internal representation of SOFIE. It does not require any user-supplied information about the opset version, information about the input or output tensor, etc. It only requires the path to the Keras file in memory and is capable to parse it by reading it and extracting data through a runtime python interface initialized by C-Python API.

```
using TMVA::Experimental::SOFIE;  
  
//Parser returns an RModel object  
RModel model = PyKeras::Parse("trained_model_dense.h5");
```

Once the model is parsed, the inference code can be subsequently generated and used by including the generated header file.

Considering the complexity of TF2ONNX on converting a Keras file to ONNX, SOFIE Keras Parser proves to be a powerful tool that is simple to use, develop and maintain. As the parser is built on the ONNX standards, it

extracts the information of various layers in a model and maps them with their analogous to the ONNX standards. It uses a simple yet effective extraction and mapping algorithm to combine ONNX operators for representing Keras layers.

## **2.b) Implemented support**

The enhancement plan included adding support for more Keras layers. With the successful completion of the plan, the parser now supports Batch Normalization, Convolution 2D, Basic Binary Operators (Add, Subtract, and Multiply), Reshape, Activation functions (Softmax, LeakyRelu, Tanh), and Concat layers. Previously, it only had the support for the Dense and Permute layer, and a few activations (ReLU, Selu, and Sigmoid).

The parser already has a well-implemented infrastructure for reading a Keras file, extracting the model information, and calling individual `make` functions for the layers which maps them with similar ONNX operators or a combination of them. The required attributes or layer information is then extracted and appropriately transformed as per the requirements of the ONNX operators.

To begin with, support for activation functions was added by developing simple functions for extracting the input and output tensors and calling their equivalent ROperators in the RModel object. For the basic binary layers, the ROperator\_BasicBinary class is used which is a template class that can be used to represent the said binary operations. For the concatenate and reshape layers, along with the information on input and output tensors, the axis about which the operation should be performed is extracted and supplied for instantiating the similar ROperator class.

For parsing the batch normalization layer, along with the information about the input and output tensors, some attributes are also extracted that includes gamma, beta, moving mean, moving variance, epsilon, and momentum, which are essential values required in the batch normalization method.

Adding support for the Convolution layer was complicated as the convolution operation in Keras supports the NHWC format by default for representing 2D image data, whereas ONNX supports NCHW. Thus, for appropriate mapping of a Keras Convolution 2D layer in SOFIE, the model requires a transpose operation to alter the data format and make it suitable before a convolution operation. Apart from this, the attribute representation in Keras and ONNX are also quite different and thus require some internal computations for successful mapping. Keras supports the same and valid padding, whereas ONNX specifies padding by numerics across each dimension. Thus, the padding values are required to be computed based on the layer specification, the kernel values, strides, and the input tensor shape.

```
long outputHeight = std::ceil(float(inputHeight)/float(fAttrStrides[0]));
long outputWidth  = std::ceil(float(inputWidth) / float(fAttrStrides[1]));

long padding_height = std::max(long((outputHeight - 1) * fAttrStrides[0] +
                                     fAttrKernelShape[0] - inputHeight), 0L);
long padding_width  = std::max(long((outputWidth - 1) * fAttrStrides[1] +
                                     fAttrKernelShape[1] - inputWidth), 0L);

size_t padding_top = std::floor(padding_height/2);
size_t padding_bottom = padding_height - padding_top;
size_t padding_left = std::floor(padding_width/2);
size_t padding_right  = padding_width - padding_left;
```

Along with padding, attribute values like dilations, strides, group, and kernel shape, are extracted and are then used for instantiating an object of `ROperator_Conv` class.

Suitable tests were added for all the implemented layers which are developed and tested using Google's GTest<sup>[12]</sup> framework for C++ testing. All the implemented codes were thoroughly reviewed and consequently merged and are now available in the master branch of ROOT<sup>[6]</sup> under the Experimental namespace of TMVA.

# Chapter 3

---

## **SOFIE Custom Operator** Design & Development

### 3.a) Motivation

SOFIE currently supports 39 operators out of the 183 operators specified in ONNX standards. Often it may be a requirement for the users to define a custom function capable of accepting input tensor(s) and outputting a resultant tensor after a certain transformation that is not defined in the ONNX standards. Thus, SOFIE required support for defining a custom operator which should be simple to define, easy to test, debug and integrate with few overheads and dependencies.

### 3.b) Definition

With the completion of the project, SOFIE now supports adding a custom-defined operator in an RModel object. A custom operator can be defined with some attributes which include the name of the operator, names of the input & output tensors, output tensor shapes, and the name of a header file that shall contain the definition of the custom operator.

In the previous edition of SOFIE, whenever the inference code was generated, a `.hxx` header file containing the infer function and a `.dat` file containing the weights of the model were produced. With the addition of the custom operator, now the user needs to define a function named Compute which should take the input tensors, and output tensors by reference and do the necessary computations required. Essentially, the Compute function should be added into a separate header file, and information about the input & output tensors should be supplied while instantiating a ROperator\_Custom object.



*Fig 5: To integrate a custom operator, the user has to supply the compute function in an additional header file which will be included in the generated inference code.*

### 3.c) Interface

```
std::unique_ptr<SOFIE::ROperator> op;  
op.reset(new SOFIE::ROperator_Custom<float>("Exp", {"denseBiasAdd0"},  
                                             {"exp_out"}, {{1,4}}, "exp_compute.hxx"));
```

The above code instantiates a custom operator object by specifying a float-templated object with “*Exp*” as the operator name, which is followed by the collection of input and output tensor names. Furthermore, the shape of the output tensor: {1,4} is also passed along with the name of the header file that shall contain the definition of the compute function.

```
int main() {  
  
    //Parsing the saved Keras .h5 file into RModel object  
    RModel model = PyKeras::Parse("KerasModelForCustomOp.h5");  
    model.Generate();  
  
    std::unique_ptr<ROperator> op;  
    op.reset(new ROperator_Custom<float>(/*OpName*/ "Scale_by_2",  
                                         /*input tensor names*/model.GetOutputTensorNames(),  
                                         /*output tensor names*/{"output"},  
                                         /*output shapes*/{{1,4}},  
                                         /*header file name with the compute function*/  
                                         "scale_by_2_op.hxx"));  
  
    // adding the custom op in the model  
    model.AddOperator(std::move(op));  
  
    // Generating inference code again after adding the custom operator  
    model.Generate();  
    model.OutputGenerated("KerasModelWithCustomOp.hxx");  
  
    return 0;  
}
```

The above code instantiates an RModel object by parsing a Keras model. It then adds a Custom operator to scale the input tensor values by 2.



The definition of the custom operator should be present in the stated header file:

```
#include <vector>
#include <algorithm>
#include <iostream>
namespace Scale_by_2{
void Compute(const std::vector<float>& input, std::vector<float>& output){
    for(size_t i=0; i<input.size(); ++i){
        output[i]=input[i]*2;
    }
}
} //Scale_by_2 operator
```

When the inference code is generated for the model, the user-provided header file is included and the Compute function is invoked for computing the transformation required in the input tensor.

### 3.d) Implementation

Adding the support for the Custom operator in SOFIE required a generalized structure of an operator where the computation is outsourced to an external file. As the function is defined by the user, SOFIE currently does not guarantee any checks on the data type and shape of input and output tensors. It is the responsibility of the user to assure that these are correct based on their function definition. Thus, the ShapeInference and TypeInference methods are not defined. For the initialization, the output tensors are checked and added as intermediate tensors of the model. The Generate function only produces the code to call the compute function by passing the input and output tensors by reference. The compute function is expected to accept the input tensors as const reference to source the data and populate the output tensors with the transformed data.

# Chapter 4

---

## **SOFIE** **Graph Neural Networks**

## 4.a) Motivation

Recent advancements in Artificial Intelligence and specifically Machine Learning have led to numerous research in the applications of sophisticated models and multi-varied methods which are capable of understanding the hidden patterns in huge datasets and provide a predictive model for future forecasting and understanding of the internal dynamics.

High-energy physics research uses advanced machine learning methods to understand and predict the nature and behavior of particles, collisions, etc. For instance, the team at CMS co-developed ParticleNet<sup>[3]</sup>, a graph neural network architecture supporting graph convolution (edge convolution and dynamic graph CNN methods). It is majorly used in studying heavy flavor jet tagging, jet mass regression, etc. On the other hand, LHCb plans to use DeepMind's Graphnets<sup>[2]</sup> library for its research on full-event filtering, trigger interpretation, etc. Graphnets<sup>[8]</sup> is essentially a static graph neural network based on TensorFlow and Sonnet. With the growing demands for graph neural networks in the high-energy physics arena, it was deemed necessary to add its support for inference in SOFIE.

## 4.b) Design

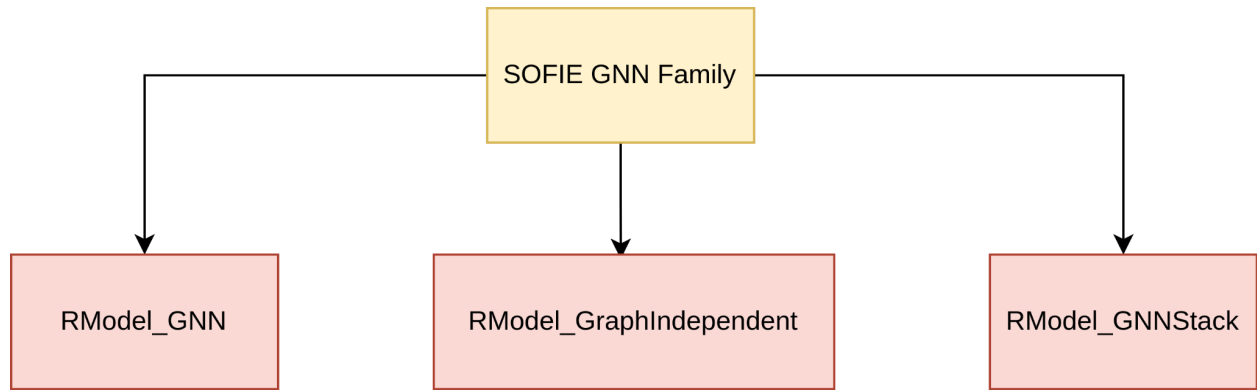
SOFIE was created based on the ONNX standards. ONNX doesn't directly support the inference of a graph neural network natively. Thus, it was difficult to design the architecture by using mere combinations of ROperators and RModels. After a lot of discussions, and research it was decided that SOFIE's support for GNN should begin with adding support for DeepMind's Graphnets.

### 4.b.i) Architecture

Graphnets' architecture comprises three major data elements: the edges, nodes, and global data. Along with the data elements, they have update and aggregate functions that operate on the elements to modify an input graph. The model primarily takes an input graph with assigned values for nodes,

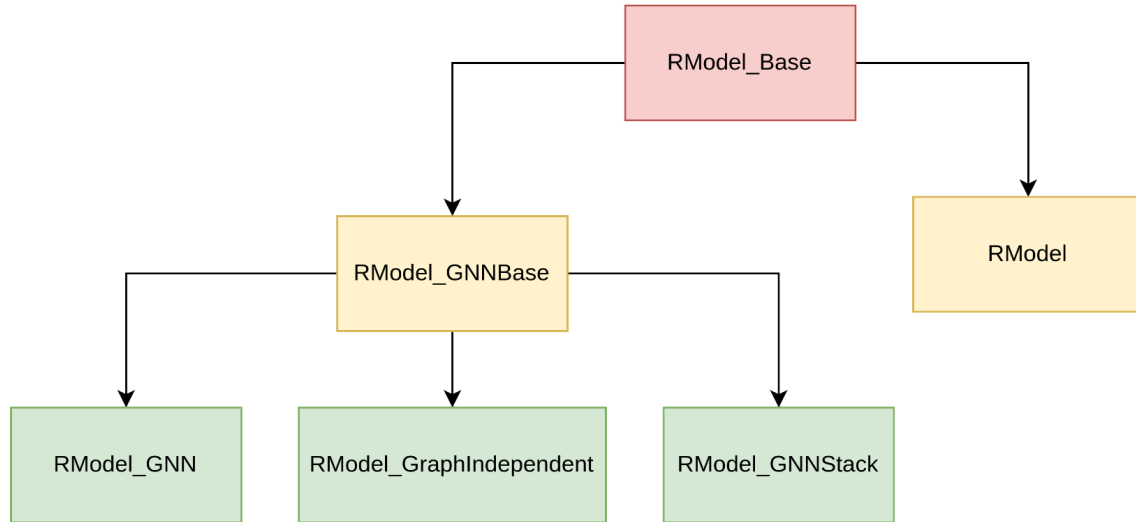
edges, globals, senders & receivers, and follows an algorithm to modify the values and result into the output graph data. The model, however, does not change the number of attributes for the data elements or the number of nodes or edges in the graph system, thus making it a static transformation.

Graphnets proposes two different architectures for the Graph modules. First, a usual graph module that has both updation and aggregate functions, and, secondly, a GraphIndependent module which is responsible for only updation of the nodes, edges, and globals, thus acting as an independent transformation of the data elements. To incorporate this, SOFIE's GNN support was developed with both architectures and a stacking architecture where combinations of Graph and GraphIndependent modules can be implemented.



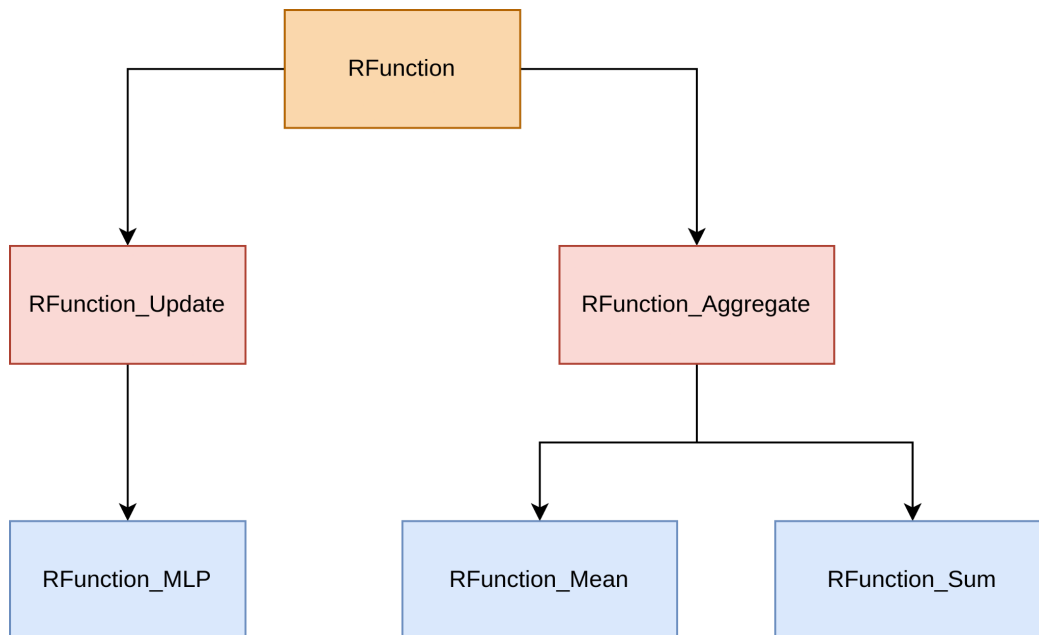
*Fig 6: SOFIE GNN Family having different class implementations for each of the usual Graph module, the GraphIndependent module, and the GNNStack for combinations of the previous two.*

To adapt the Graphnets model into SOFIE, the architecture of SOFIE was modified to make it more modular and comprehensible. Currently, SOFIE has an RModel class that stores the model configuration, essentially the input tensor data, output tensor data, operators, and initialized tensors. With the addition of the GNN family, significant changes were made for improving inheritance and code reusability.



*Fig 7: With the addition of the GNN support, the RModel\_Base now acts as the parent class which has RModel\_GNNBase and RModel as children. RModel\_GNNBase further acts as the parent class for all the GNN modules.*

The graph modules in SOFIE contain the data members, and functions required for the computations on them. Computing functions are one among the RFunctions, responsible for the update or aggregate of graph data elements.



*Fig 8: Graph modules in SOFIE have functions to operate on the data members. The functions include update functions like MLP or aggregate functions like ReduceSum or ReduceMean.*

## 4.b.ii) Intermediate Representation

Graphnets' representation consists of a GraphModule object containing the data for the nodes, edges, and global members, the sender's list, the receiver's list, the update functions, and the aggregate functions. Adapting this, SOFIE's Graph family has a similar representation for the data internally but differs in their configurations.

### 4.b.ii.1) RModel\_GNN

The basic yet most comprehensive Graph module contains the following data members for the model configuration.

```
std::unique_ptr<RFunction_Update> edge_update;  
std::unique_ptr<RFunction_Update> node_update;  
std::unique_ptr<RFunction_Update> global_update;  
  
std::unique_ptr<RFunction_Aggregate> node_edge_agg;  
std::unique_ptr<RFunction_Aggregate> node_global_agg;  
std::unique_ptr<RFunction_Aggregate> edge_global_agg;  
  
int num_nodes, num_edges;  
int num_node_features, num_edge_features, num_global_features;  
std::vector<int> senders, receivers;
```

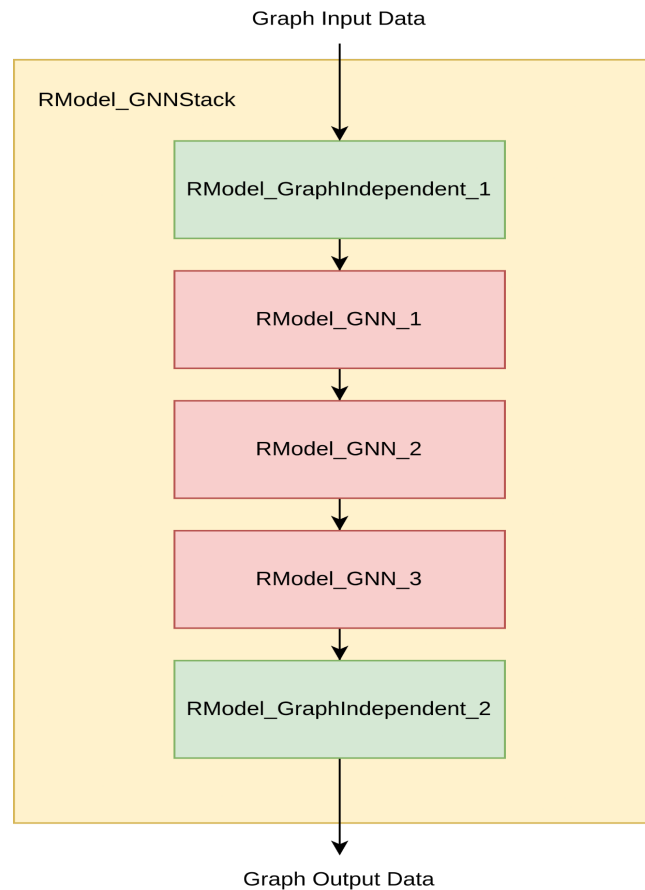
The functions are represented as unique\_ptr to particular RFunction objects depending on their usage. The number of edges and nodes and the number of their features are stored along with the number of global features. To represent the relative edges, their indices are used in the senders and receivers vectors.

### 4.b.ii.2) RModel\_GraphIndependent

Unlike the GNN module, the GraphIndependent module was designed for individual transformation of the data members in graph data without considering any relation between them. Thus, they do not require aggregate functions. The remaining data members from the GNN module constitute the internals of the GraphIndependent module.

### 4.b.ii.3) RModel\_GNNStack

Most modern graph networks may require collections of GNN and GraphIndependent modules for their architecture. RModel\_GNNStack facilitates the combination of the said modules and produces the inference code where the graph input data moves from the constituent modules and results to the output graph data at the end.



*Fig 9: Structure of an RModel\_GNNStack object*

Unlike GNN and GraphIndependent modules, the GNNStack contains its constituents in a vector: `std::vector<std::unique_ptr<RModel_GNNBase>> fGraphs`. The GNNStack model essentially produces the inference code for each individual Graph module in separate header files. It includes all those header files into one and calls the respective infer functions sequentially with the output of one as the input of the next.

#### **4.b.ii.4) RFunctions**

Analogous to ROperators, RFunctions are responsible for computations in the graph data in a graph module.

RFunctions can be of 2 types: RFunction\_Update and RFunction\_Aggregate. Instead of tensor data as input in ROperators, input data of RFunctions depends on their type (update or aggregate) and target (node, edge, global).

##### **4.b.ii.4.a) RFunction\_Update**

RFunction\_Update is responsible for updating the data members in a graph structure. They essentially contain an RModel object and use its generated infer function for updating the data members. In the first edition of SOFIE GNN, the RFunction\_MLP was implemented as the only update function. For updating the edges in a graph module structure, the update function requires the values for the current edge, the receiver node, the sender node, and the global data. For node updation, it requires the aggregate values of the connected edges, the current node value, and the global value. Global updation requires the aggregate value of all edges, the aggregate value of all nodes, and the current global value. The input tensors significantly change in the update functions for the GraphIndependent model which just needs the individual nodes, edges, and global values for their updation respectively with no relation to other data members whatsoever.

##### **4.b.ii.4.b) RFunction\_Aggregate**

Aggregate functions are basically responsible for reduction operations in a collection of tensors. Edges for every node are required to be reduced to an aggregate value for updating the nodes' value. Similarly, all the edges and nodes are required to be reduced for updating the global values in a graph module. Unlike RFunction\_Update, RFunction\_Aggregate has the same form of input data for different relations. Currently, the RFunction\_Sum and RFunction\_Mean are implemented. They only require the collection of input tensors and they result in their reduced form by doing the required computation. As of now, RFunction\_Aggregate are not designed to have an



RModel object, rather their inference code is generated by simple tensor calculations which seemed simpler, modular, and non-repetitive since aggregate functions do not usually contain any initialized tensors for their computation. However, in the future, if the need arises for the Aggregate function to contain an RModel object, suitable changes can be done in their architecture to implement the support.

### 4.b.iii) Graph Data

A GNN\_Data structure was implemented to represent graph data values. An object of the structure is implemented to contain the graph data and consequently, is moved through various graph modules for the required updation and aggregation to result in the output graph data.

```
struct GNN_Data {  
    std::vector<float> node_data;  
    std::vector<float> edge_data;  
    std::vector<float> global_data;  
};
```

The above structure includes vectors of float containing the node, edge, and global data values. All the data values for the nodes, edges, and globals are concatenated and passed as a single entity respectively in the structure object. Depending on the number of nodes in a model and the number of features each node has, the node data vector is indexed accordingly while passing the data to the update or the aggregate function.

### 4.c) Inference

The inference code generated for the graph modules varies distinctively depending on their model configuration and usage. For the usual GNN model, a header file is generated along with a weight file containing the initialized tensors of all the update functions indexed sequentially, and are so extracted during inference. All the update function's RModel objects' inference codes are generated for their function definition and thus used

during the update of nodes, edges, or globals. Aggregate functions operate on relations on edge-node, global-node, and global-edge. For their definition, it is first checked whether they differ in terms of operation. If they do, then different implementations are defined, otherwise, they are defined once and are called for aggregation.

#### 4.c.i) RModel\_GNN

The inference algorithm basically involves the updation of edges, nodes, and global data in consecutive order.

```
function GNN(E,N,G): # E:edges, N:nodes, G:globals
    for k in range(1,len(E)):
        E[k] = edge_update(E[k],N[E[k].receiver],N[E[k].sender],G)

    for i in range(1,len(N)):
        Elist = list()
        for k in range(1,len(E)):
            if E[k].receiver == i:
                Elist.append({E[k],N[E[k].receiver],N[E[k].sender]})
        EN_Agg = edge_node_agg(Elist) if len(Elist) else {0,0,0}
        N[i] = node_update(EN_Agg,N[i],G)

    Nlist= list()
    Elist = list()
    for i in range(1,len(N)):
        Nlist.append(N[i])
    for i in range(1,len(E)):
        Elist.append({E[i],N[E[i].receiver],N[E[i].sender]})
    EG_Agg = edge_global_agg(Elist)
    NG_Agg = node_global_agg(Vlist)
    G = global_update(EG_Agg, NG_Agg, G)

    return {E,N,G}
```

The above pseudo-code shows the order of operation for the inference of the GNN model. The edges are at first updated with their previous data, their receiver and sender's node data, and the global data. For updating the node values, firstly, if there is any connected edge present for which the node acts as a receiver, are reduced. If no such edge exists, then the node update function uses zero values as a placeholder. The aggregates, along with the

previous node values and the global values are then used to update the nodes. For global updates, all the edges and nodes are reduced and are then passed along with the previous global data in the global update function.

#### 4.c.ii) RModel\_GraphIndependent

The GraphIndependent model is similar in configuration to the GNN, but it doesn't have any aggregate functions. They are only for individual data transformation, thus they follow the below-mentioned algorithm:

```
function GraphIndependent(E,N,G):# E:edges, N:nodes, G:globals
    for k in range(1,len(E)):
        E[k] = edge_update(E[k])

    for i in range(1,len(N)):
        N[i] = node_update(N[k])

    G = global_update(G)

    return {E,N,G}
```

Inference of GraphIndependent object also starts with updating the edges. In this case, however, only the previous values of the edges are used for the updation, with no regard to the receiver's or the sender's node data. Similarly, the node and globals update is also on their past values and no other relationship is considered.

#### 4.d) Parsing

An important part of adding support for graph neural networks is implementing the ability to parse a GNN model from a different architecture or format and translate it to SOFIE's GNN representation. DeepMind's Graphnets currently does not support serializing a model into memory, thus a usual I/O parser cannot be developed for this case. For parsing a model developed in Graphnets, an in-memory parser was required to accept the Graphnets model from the PyROOT interface and build SOFIE's GNN model.

# Future Developments

---

Future developments of the project focuses on various enhancements in the Keras parser and the newly added Graph Neural Network support.

For the Keras parser, support for various other layers is planned for implementation, which includes RNN, LSTM, GRU, Conv1D, Conv3D, etc. Adding the support for the stated layers will enhance the parser's capability and will expand its range of usage. Furthermore, attributing to Pull Request #11518<sup>[13]</sup>, the Keras parser can be modified to have a registration-based architecture for the layers it supports, which shall make it more modular, efficient, and easier to maintain.

For the SOFIE GNN implementation, a wide range of new update and aggregate functions can be developed to broaden its usage. Furthermore, as the architecture was built from scratch, it is currently in its first edition, and general optimizations on inference and architecture can be performed. As it currently supports GNN architectures based on DeepMind's Graphnets, enhancements can be implemented to add support for Particlenet by CMS, or PyTorch Geometric. Particlenet is essentially a combination of various ONNX operators to work as a graph neural network, thus its support will require few modifications on the current architecture. On the contrary, GNN models built on PyTorch Geometric have dynamic graph convolution operations, and thus its implementation in SOFIE's GNN Family will require the implementation of a dynamic graph data structure and algorithms supporting that.

# Conclusion

---

This report summarises the work implemented over the course of the Summer Student Program at CERN on enhancing the TMVA SOFIE: Machine Learning Inference Engine. The work for 13 weeks comprised improving the Keras parser, adding support for the Custom operator, and implementing support for Graph Neural Networks in SOFIE.

Improvements on the Keras parser were implemented and integrated into SOFIE in the master branch of ROOT<sup>[6]</sup>. Similarly, the custom operator was also integrated with the required tests in the master branch. GNN Support was prototyped and has its first version which is currently under extensive review with validations using GNN data from LHCb.

The improved Keras parser now has extended support for layers, and thus will be beneficial for users who want to use SOFIE to infer their Keras models. With the custom operator, user-defined operations can now be added to an RModel object, thus users have the flexibility to design and integrate their own computation functions. SOFIE's new GNN implementation is highly beneficial for usage in high-energy physics research because of the huge demand for graph neural networks for understanding complex data, and having its support in the engine will facilitate fast inference of models built in various GNN frameworks in a simplified manner.

# Acknowledgement

---

The CERN Summer Student Program is undoubtedly the best scientific endeavor I have experienced to date. The 13 weeks I spent at the CERN site were excellent and filled with numerous unique experiences. First and foremost, I would like to thank my supervisor, Dr. Lorenzo Moneta, to provide me with the opportunity to work on the project, and for all the guidance throughout the application procedure and the project duration. I am highly grateful for all the support and learning I have received. Furthermore, I am thankful to the entire team of ROOT and the EP-SFT group for having me. The discussions and interactions I had with the members of the team were very insightful and I learned a lot from them. I also thank all the summer students for their rigorous dedication to their projects to make the program a grand success. I gratefully acknowledge the support of the organizers of the summer student program, Adriana Bejaoui, and Hannah Luther, and thank them for helping the summer students throughout the program with administrative support. Lastly, I convey my sincere gratitude to CERN for creating and organizing this program of learning and networking, and for giving us the opportunity to experience the largest particle physics laboratory in the world.

# References

---

- [1] Sitong An, Lorenzo Moneta, “*C++ Code Generation for Fast Inference of Deep Learning Models in ROOT/TMVA*”, 25th International Conference on Computing in High-Energy and Nuclear Physics, 17th-21st May 2021.
- [2] Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Caglar Gulcehre, Francis Song, Andrew Ballard, Justin Gilmer, George Dahl, Ashish Vaswani, Kelsey Allen, Charles Nash, Victoria Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matt Botvinic, “*Relational inductive biases, deep learning, and graph networks*”, June 4, 2018.
- [3] Huilin Qu, Loukas Gouskos, “*Jet Tagging via Particle Clouds*”, Physics Rev. D 101, 056019, March 26, 2020
- [4] “*The ROOT Manual*”, <https://root.cern/manual/>
- [5] “*TMVA 4 Toolkit for Multivariate Data Analysis with ROOT Users Guide*”, <https://root.cern.ch/download/doc/tmva/TMVAUsersGuide.pdf>
- [6] “*root-project/root: The official repository for ROOT: analyzing, storing and visualizing big data, scientifically*”, GitHub repository for ROOT, the data analysis tool by CERN.
- [7] “*onnx/onnx: Open standard for machine learning interoperability*”, GitHub repository on the specifications and architecture of ONNX standards
- [8] “*deepmind/graph\_nets: Build Graph Nets in Tensorflow*”, GitHub repository of DeepMind’s open-sourced GraphNets implementation
- [9] “*microsoft/onnxruntime: ONNX Runtime: cross-platform, high-performance ML inferencing and training accelerator*”, GitHub repository of Microsoft’s runtime engine for the inference of onnx models
- [10] “*keras-team/keras: Deep Learning for humans*”, Github repository of the Keras framework
- [11] “*onnx/tensorflow-onnx: Convert TensorFlow, Keras, Tensorflow.js and Tflite models to ONNX*”, Github repository of the TF2ONNX utility by ONNX ecosystem
- [12] “*GoogleTest User’s Guide*”, <https://google.github.io/googletest/>
- [13] “*[TMVA][SOFIE] ONNX parser with registry like pattern*”, Pull request by Ahmat Hamdan on adding a registry interface for the ONNX parser