Redesign of the TMVA application interface

TMVA::Transformation
TMVA::Inference
TMVA::Utility

ROOT
Data Analysis Framework

https://root.cern

Overview

- Topical PPP meeting about TMVA: https://indico.cern.ch/event/731642/
- **Proposal for a reorientation of TMVA adapted to the modern machine-learning landscape**. Let's focus on:
  - Data-loading from ROOT files
  - Robust, persistent and efficient implementations of selected machine-learning methods
  - Application runtime performance in Python and C++
  - Tight integration with `RDataFrame`
- **Todays topic:** Discussion of the C++ interface for the "serving layer" (aka "inference", "application") in TMVA

Evolution of HEP x ML Engineering

	HEP (Circa 2013)	HEP (Circa 2018)	**HEP (Circa 2019)**
Data Layer	ROOT Files	ROOT Files	ROOT Files
Loading Layer	Ad hoc ROOT ETL logic	Numpy / HDF5 Converters / Loaders	**RDataFrame + TMVA**
Training Layer	TMVA	Keras, TensorFlow, PyTorch, XGBoost, scikit-learn, …	Keras, TensorFlow, PyTorch, XGBoost, scikit-learn, TMVA, ...
Serving Layer	Deployment Target (TMVA)	Deployment Target (lwtnn, TensorFlow, TMVA wrappers)	**RDataFrame + TMVA**

Current application interface

```cpp
// 1) Create reader
auto reader = TMVA::Reader("option_str");

// 2) Book input variables
float var1, var2, var3, var4;
reader.AddVariable("var1", &var1);
reader.AddVariable("var2", &var2);
reader.AddVariable("var3", &var3);
reader.AddVariable("var4", &var4);

// 3) Book MVA method
reader.BookMVA("name", "weights.xml");

// 4) Fill input variables and perform application
var1 = 1; var2 = 2; var3 = 3; var4 = 4;
float prediction = reader.EvaluateMVA("name");
```

- Variable transformation and MVA method combined in a closed system, restricts interaction with external libraries
- Workflow highly specialized on event-by-event application with flat inputs, complicated to feed multi-dimensional inputs (image data) and perform batch processing (more efficient)
- Complicated feeding of user data, minor support of STL containers

New application interface

```cpp
// 1) Gather data
vector<float> inputs = {1, 2, 3, 4};

// 2) Transform data with some preprocessing method
auto preprocessing = TMVA::Transformation::StandardScaler<float>("parameters.pickle");
vector<float> inputs_preprocessed = preprocessing.Transform(inputs);

// 3) Feed inputs to the ML model
auto model = TMVA::Inference::Keras<float>("model.h5");
vector<float> prediction = model.Predict(inputs_preprocessed);

// 4) Postprocess prediction
auto postprocessing = TMVA::Transformation::StandardScaler<float>("parameters.pickle");
vector<float> prediction_postprocessed = postprocessing.InverseTransform(prediction);
```

- Inspired by the interface of popular high-level ML packages (sklearn, Keras, xgboost, ...)
- Accepts user data as `std::vector` and `numpy.array` in Python
- Enables to load serialized parameters from TMVA methods and external libraries (sklearn preprocessing module, Keras models, ...)
- How to handle multi-dimensional inputs such as images? → See next slide!

Batch processing

```cpp
// 1) Load preprocessing and ML model
auto preprocessing = TMVA::Transformation::StandardScaler<float>("parameters.pickle");
auto model = TMVA::Inference::Keras<float>("model.h5");

// 2) Gather data in a multi-dimensional container
template <typename T>
using RTensor = typename std::vector<std::vector<T>>;

RTensor<float> inputs = {{1, 2, 3, 4}, {-1, -2, -3, -4}};

// 3) Perform application on all inputs efficiently in one computation
RTensor<float> prediction = model.Predict(preprocessing.Transform(inputs));
```

- Need for a container representing multi-dimensional arrays (`RTensor`/`RMultiVec`).
- Same problem (and solution) as for the support of multi-dimensional inputs (e.g. images)
- Allows to adopt application interface in Python with `numpy.array` as container
- Container with shape information as well needed for future redesign of training interface (passing of datasets/batches), `RDataFrame.MultiTake`, ...

Integration with RDataFrame

```cpp
// 1) Load preprocessing and ML model
auto preprocessing = TMVA::Transformation::StandardScaler<float>("parameters.pickle");
auto model = TMVA::Inference::Keras<float>("model.h5");

// 2) Set up a dataframe
ROOT::EnableImplicitMT();
auto df = ROOT::RDataFrame("TreeS", "tmva_class_example.root");

// 3) Gather inputs as std::vector
auto df1 = df.Define("inputs", TMVA::Utility::MakeVector<4, float>(), {"var1", "var2", "var3", "var4"});

// 4) Perform application in the dataframe event-loop
// 4.1) Only the model, without preprocessing
auto df2 = df1.Define("predictions", model, {"inputs"});
// 4.2) Apply model with preprocessed inputs
auto df2 = df1.Define("predictions", TMVA::Utility::Chain<float>(preprocessing, model), {"inputs"});
```

- ML methods and preprocessing needs to be thread-safe to support multi-threading?
- Provide copy constructor for the methods and use a copy in each thread?

Summary

- **Features supported by the proposed application interface:**
  - Separated variable transformation and ML inference
  - Data ingestion via STL container
  - Efficient event-by-event application
  - Application on batches of events
  - Tight integration with `RDataFrame` supporting implicit multi-threading

- **To be discussed:**
  - Proposed workflow
  - Need for container in ROOT representing multi-dimensional arrays (`RTensor`/`RMultiVec`).
  - Handling of multi-threaded application in `RDataFrame` (thread safety, copy constructors).