RTensor

Proposing a C++ container for multi-dimensional arrays

ROOT
Data Analysis Framework
https://root.cern

Why do we need this?

So-far identified use-cases:

- Input/output container for machine-learning methods manipulating high-dimensional data, e.g., images
- Internals for TMVA neural network implementation
- C++ representation of `numpy.array` supporting proper pythonizations
- `RDataFrame.MultiTake` return value
- `...`

Design decisions

- Container wrapping contiguous data with additional shape information
- No support for complex interaction with the data, e.g., matrix multiplication or broadcasting
- Interface very similar to `numpy.array` and `xtensor`

- **Most features shown in the next slides are implemented here in a proof of concept:** https://github.com/stwunsch/root/tree/dev-rtensor

Features

Constructors

C++

```
>>> using namespace TMVA::Experimental;
>>>
>>> // Initialize from data via memory adoption
>>> float data[] = {1, 2, 3, 4, 5, 6};
>>> auto x = RTensor<float>(data, {2, 3});
>>> std::cout << x << std::endl;
{ { 1, 2, 3 }
  { 4, 5, 6 } }
>>>
>>> // Initialize with owned data
>>> auto x = RTensor<float>({2, 3});
>>> std::cout << x << std::endl;
{ { 0, 0, 0 }
  { 0, 0, 0 } }
```

- Constructors supporting:
  - Memory adoption (mutable view)
  - Owning memory

Python

```
>>> import ROOT
>>> RTensor = ROOT.TMVA.Experimental.RTensor
>>>
>>> # Initialize from data via memory adoption
>>> data = ROOT.std.vector("float")((1, 2, 3, 4, 5, 6))
>>> shape = ROOT.std.vector("size_t")((2, 3))
>>> x = RTensor(data.data(), shape)
>>> print(x)
{ { 1, 2, 3 }
  { 4, 5, 6 } }
>>>
>>> # Initialize with owned data
>>> x = RTensor("float")(shape)
>>> print(x)
{ { 0, 0, 0 }
  { 0, 0, 0 } }
```

- Pretty printing in C++ and Python
- Interoperability with `numpy.array` shown on next slides

Container properties

C++

```
>>> // Initialize with owned data
>>> auto x = RTensor<float>({2, 3});
>>> std::cout << x << std::endl;
{ { 0, 0, 0 }
  { 0, 0, 0 } }
>>>
>>> // Print container properties
>>> std::cout << x.GetShape() << std::endl;
{ 2, 3 }
>>> std::cout << x.GetData() << std::endl;
0xc75f80
>>> std::cout << x.GetMemoryOrder() << std::endl;
'C'
>>> std::cout << x.HasOwnedData() << std::endl;
true
```

- Container properties:
  - Pointer to data
  - Shape
  - Ordering in memory (row vs column ordering)
  - Data ownership

Set and get elements

C++

```
>>> // Initialize tensor
>>> auto x = RTensor<float>({2, 3});
>>> std::cout << x << std::endl;
{ { 0, 0, 0 }
  { 0, 0, 0 } }
>>>
>>> // Set elements
>>> x.At(0,0) = 1;
>>> x(0,1) = 2;
>>> std::cout << x << std::endl;
{ { 1, 2, 0 }
  { 0, 0, 0 } }
>>>
>>> // Get elements
>>> std::cout << x.At(0,0) << ", " << x(0,1) << std::endl;
1, 2
```

- Set and get elements with `x.At(i,j,k,…)` or `x(i,j,k,…)`
- No definition of `operator[]` because `x[i,j]` not possible in C++

Python

```
>>> # Initialize tensor
>>> x = RTensor("float")(shape)
>>> print(x)
{ { 0, 0, 0 }
  { 0, 0, 0 } }
>>>
>>> # Set element
>>> x[0,0] = 1
>>> print(x)
{ { 1, 0, 0 }
  { 0, 0, 0 } }
>>>
>>> # Get element
>>> print(x[0,0])
1
```

- `x[i,j,...]` possible in Python due to `__getitem__` and `__setitem__` pythonizations

Reshape, expand dims and squeeze

C++

```
>>> // Initialize from data via memory adoption
>>> float data[] = {1, 2, 3, 4, 5, 6};
>>> auto x = RTensor<float>(data, {2, 3});
>>> std::cout << x << std::endl;
{ { 1, 2, 3 }
  { 4, 5, 6 } }
>>>
>>> // Reshape
>>> x.Reshape({3, 2});
>>> std::cout << x << std::endl;
{ { 1, 2 }
  { 3, 4 }
  { 5, 6 } }
>>>
>>> // Reshape again
>>> x.Reshape({6, 1});
>>> std::cout << x << std::endl;
{ { 1, 2, 3, 4, 5, 6 } }
```

```
>>> // Squeeze (remove dimensions of 1)
>>> x.Squeeze();
>>> std::cout << x << std::endl;
{ 1, 2, 3, 4, 5, 6 }
>>>
>>> // Expand dimensions again
>>> x.ExpandDims(1);
>>> std::cout << x << std::endl;
{ { 1, 2, 3, 4, 5, 6 } }
```

- Implements basic functionality known from numpy

Interoperability with numpy.array

RTensor → numpy.array

```
>>> # Initialize RTensor from data
>>> import ROOT
>>> x = ROOT.RTensor(data.data(), shape)
>>> print(x)
{ { 1, 2, 3 }
  { 4, 5, 6 } }
>>>
>>> # Adopt memory and create a view with a numpy.array
>>> import numpy
>>> x_numpy = numpy.asarray(x)
>>> print(x_numpy)
[[1. 2. 3.]
 [4. 5. 6.]]
```

numpy.array → RTensor

```
>>> # Create a numpy.array
>>> import numpy
>>> x_numpy = numpy.array([[1, 2, 3], [4, 5, 6]])
>>> print(x_numpy)
[[1. 2. 3.]
 [4. 5. 6.]]
>>>
>>> # Adopt memory and create a view with an RTensor
>>> import ROOT
>>> x = ROOT.AsTensor(x_numpy)
>>> print(x)
{ { 1, 2, 3 }
  { 4, 5, 6 } }
```

- Possibility to write C++ code processing `numpy.array`s without hard dependency on Python libraries
- `RTensor` → `numpy.array`: Memory adoption via the `__array_interface__` mechanism
- `numpy.array` → `RTensor`: Uses similar mechanism in the `ROOT.AsTensor` pythonization

Example use-case

Interoperability with numpy in TMVA

Python

```python
# Gather training data
x_numpy = numpy.array(some_input_variables)
y_numpy = numpy.array(some_targets)

x = ROOT.AsTensor(x_numpy)
y = ROOT.AsTensor(y_numpy)

# Train TMVA model
bdt = ROOT.TMVA.BDT(num_trees=800, depth=3)
bdt.Fit(x, y)

# Apply model on data
prediction = bdt.Predict(x)

# Evaluate prediction with numpy methods
prediction_numpy = numpy.asarray(prediction)
print(numpy.mean(prediction_numpy))
```

- Further possibility with pythonizations to accept `numpy.array` as input arguments of TMVA models →

  `prediction = bdt.Predict(x_numpy)`

What comes next?

Open questions

- Missing features?
  - Slicing?
  - STL iterator interface?
  - `RTensor.Apply` for elementwise modification of elements?

- Ideas for additional use-cases in ROOT?
- Shall we go for a proper implementation?