Why use of C++ in a python dominated field?

Why C++?

• Personal higher affinity to C++
• Accessing data from ROOT format in python not optimized
  → For loop ‘for event in tree’ quite slow
  → Usage of work-around like TTree::Draw
  → Usage of external package like uproot
• TTree manipulation in python not optimized for attaching the inference of the network

→ Installation guide for PyTorch C++ API in the back up
Python syntax

```python
import torch
import torch.nn.functional as F

class Net(torch.nn.Module):
    def __init__(self):
        self.fc1 = torch.nn.Linear(784, 64)
        self.fc2 = torch.nn.Linear(64, 32)
        self.fc3 = torch.nn.Linear(32, 10)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.dropout(x, p=0.5)
        x = F.relu(self.fc2(x))
        x = F.log_softmax(self.fc3(x), dim=1)
        return x
```
C++ syntax

```cpp
#include <torch/torch.h>

struct Net : torch::nn::Module {
    Net() {
        fc1 = register_module("fc1", torch::nn::Linear(784, 64));
        fc2 = register_module("fc2", torch::nn::Linear(64, 32));
        fc3 = register_module("fc3", torch::nn::Linear(32, 10));
    }

    torch::Tensor forward(torch::Tensor x) {
        x = torch::relu(fc1->forward(x));
        x = torch::dropout(x, /* p */ 0.5);
        x = torch::relu(fc2->forward(x));
        x = torch::log_softmax(fc3->forward(x), /* dim */ 1);
        return x;
    }

    torch::nn::Linear fc1{nullptr}, fc2{nullptr}, fc3{nullptr};
};
```
Python syntax

```python
import torch
import torchvision

net = Net()

trainset = torchvision.datasets.MNIST(root='./data', train=True,
                                     download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64,
                                          shuffle=True, num_workers=2)

optimizer = optim.SGD(net.parameters(), lr=0.01)
criterion = nn.NLLLoss()

for epoch in range(2):
    for batch in trainloader:
        optimizer.zero_grad();
        inputs, labels = batch

        outputs = net(inputs)
        loss = criterion(outputs, labels)

        loss.backward()
        optimizer.step()
```

4/8 | PyTorch C++ API | David Brunner | Thematic CERN School of Computing | 16.6.2021 |
C++ syntax

```cpp
#include <torch/torch.h>
#include <memory>

int main() {
    std::shared_ptr<Net> net = std::make_shared<Net>();

    auto trainset = torch::data::datasets::MNIST("./data").map(
        torch::data::transforms::Stack<>();
    auto trainloader = torch::data::make_data_loader(
        std::move(trainSet),
        /*batch_size=*/64);

    torch::optim::SGD optimizer(net->parameters(), /*lr=*/0.01);
    torch::nn::NLLLoss criterion();

    for (std::size_t epoch = 1; epoch <= 2; ++epoch) {
        for (auto& batch : *trainloader) {
            optimizer.zero_grad();

            torch::Tensor prediction = net->forward(batch.data);
            torch::Tensor loss = criterion->forward(prediction, batch.target);

            loss.backward();
            optimizer.step();
        }
    }
}
```
Python vs C++ syntax: In general

For most cases: Python and C++ similar in logic

For some functionalities there are differences:

- Tensor slicing, see C++ documentation

Python:

```python
tensor[:,:,:0]
```

C++:

```cpp
using namespace torch::Indexing;
tensor.index({Slice(), Slice(), 0});
```

- Dataloader functionality
- Padded packed sequences
Synergy of ROOT and PyTorch C++

Using PyTorch C++ for data preprocessing

- Write custom PyTorch dataset class, see python example [here](#).
- With C++ API, using the custom dataset class with the PyTorch dataloader class, a ROOT TTree can be processed directly and returned in batched format.
  → See example in the back up.

Using PyTorch C++ to attach model inference to TTree

- Evaluate score for analysis with trained model in C++ and simply append a branch in existing TTree, see in the ROOT [documentation](#).
- Also possible, but not tried myself: Train model in python and load trained model only for evaluation in C++ application.
• C++ interfacing machine learning application in 2021 is possible and usable
• PyTorch C++ API quite developed and almost as user friendly as python version
• ROOT and PyTorch C++ API are a feasible option in physics analysis
• But in the end it is personal preference, because C++ has its disadvantage (syntax, not interpreted, compiling)
How to get?
For full documentation see [here](#)

- Using package installer like anaconda/pip, provides full library including python
- Download compiled C++ library, called libtorch, without python interface
- Hardcore way: Install from source, not recommended if not really necessary
How to use/compile? (tested on Centos7)

- Only one header to include: `#include <torch/torch.h>`
- Compiling using CMake, see [here](#)
- Using gcc, little more involved:
  
  Include flags:
  
  ```
  -l{torchPath}/torch/include/torch/csrc/api/include/
  -l{torchPath}/torch/include/torch
  -l{torchPath}/torch/include
  ```

  Linker flags:
  
  ```
  -Wl,-rpath,{torchPath}/torch/lib -L{torchPath}/torch/lib
  -ltorch -lc10 -Wl,–no-as-needed, -ltorch_cpu
  ```
#include <torch/torch.h>
#include <TTree.h>
#include <TBranch.h>

struct MyTensor{
  torch::Tensor data, label;
};

class MyDataSet : public torch::data::datasets::Dataset<HTagDataset, MyTensor>{
  private:
    std::shared_ptr<TTree> tree;
    int label;

    std::vector<float> values;
    std::vector<TBranch*> branches;

  public:
    MyDataSet(std::shared_ptr<TTree>& tree, label) : tree(tree), label(label){
      for(std::string& bname : {"Electron_pT", "Electron_eta"}(
        values.push_back(0.);
        tree->SetBranchAddress(bname.c_str(), &values.back());
      )
    }

    torch::optional<size_t> size() const { return tree->GetEntries();}

    MyTensor get(size_t index){
      tree->GetEntry(index);
      return {torch::from_blob(values.data(), {1, 2}).clone(),
        torch::tensor({label})};
    }
};