Using pytorch models in c++ projects

Georgy Derevyanko

Plan

Pytorch C++ API intro:

- TH library: THTensor, THStorage
- ATen: Tensor, Storage, Array …

Converting model:

- JIT
- Parsing JIT in C++
- Compiling using CMake and SConstruct

Showing that it works:

- Demo

TH Library: moved to c10 namespace

Old code:

https://github.com/pytorch/pytorch/tree/master/aten/src/TH/generic

Code like this (C interface):

THTensor_(nDimension)(const THTensor *self);

is converted to:

THFloatTensor_nDimension(const THTensor *self);

THDoubleTensor_nDimension(const THTensor *self);

THByteTensor_nDimension(const THTensor *self);

...

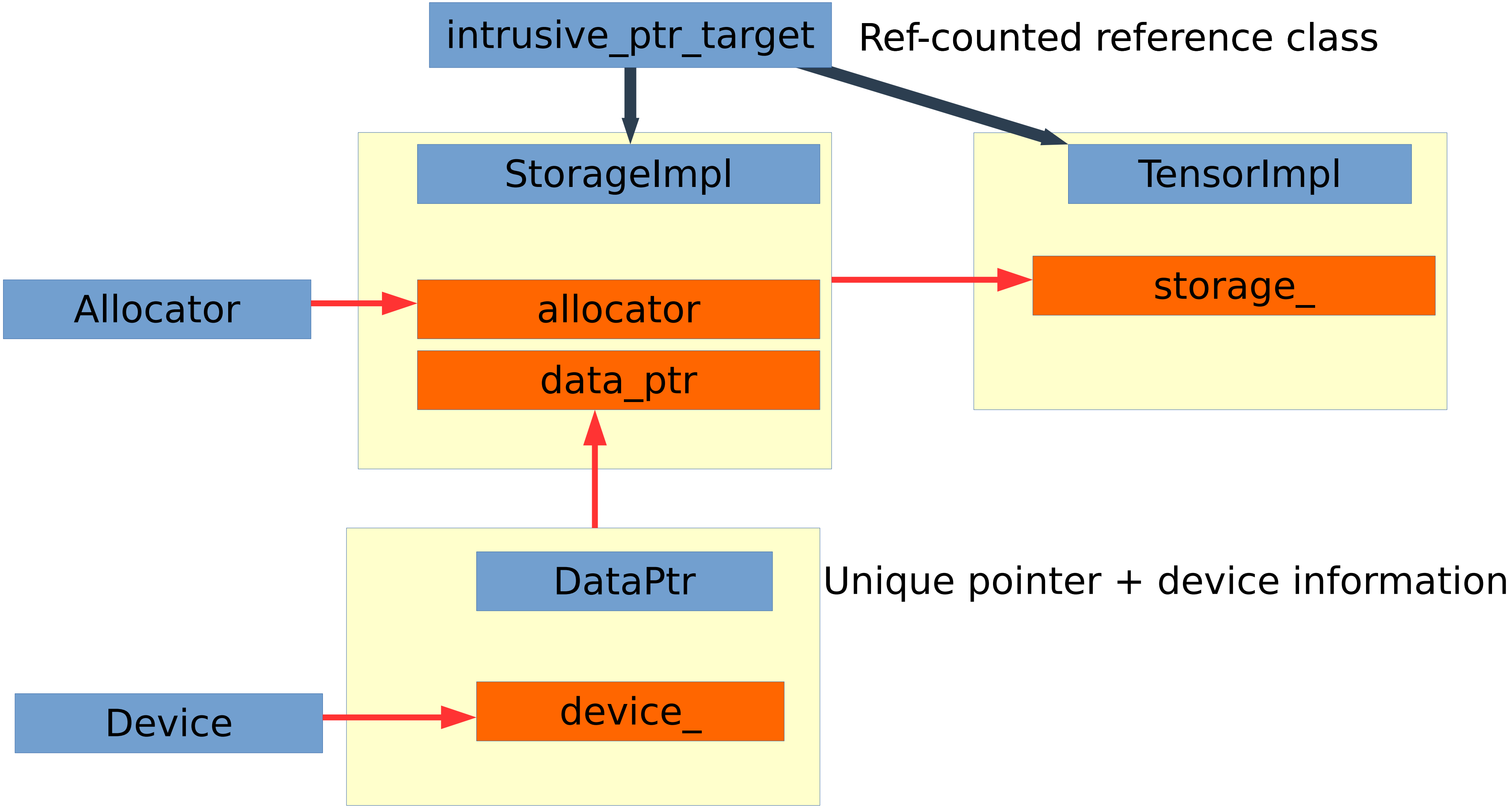
Tensor properties, creation, manipulation: THTensor.h, THTensor.cpp

Storage: THStorage.h, THStorage.cpp

...

THLibrary is only an interface to c10:TensorImpl

c10: general architecture



Ref-counted reference class

Unique pointer + device information

ATen: general architecture

c10::Storage wraps c10::StorageImpl

...

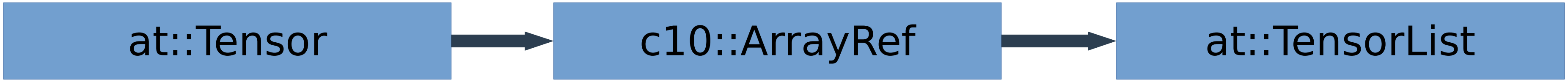
C10Tensor wraps c10::TensorImpl

at::Tensor wraps pointer to the tensor implementation

c10 + at: Arrays and TensorList

```
std::vector<torch::Tensor> cc2({x, torch::zeros({batch_size, x.size(1), 1, 1}, this_device)});

torch::Tensor y2 = at::cat(at::TensorList(cc2), 2);
```



https://github.com/pytorch/pytorch/blob/master/c10/util/ArrayRef.h

in https://github.com/pytorch/pytorch/blob/master/aten/src/ATen/core/Tensor.h:

using TensorList = ArrayRef<Tensor>;

JIT

Let’s say you’ve trained your model in python.

```
model.eval()

dummy_input = torch.ones(1,1024, dtype=torch.float, device='cpu')
output = model(dummy_input)

trace = torch.jit.trace(model, dummy_input)
trace.save(output_filename)
```

Documentation (running traced script):

https://pytorch.org/docs/stable/jit.html

https://pytorch.org/tutorials/advanced/cpp_export.html

Reimplementing the model in C++

```
struct CREPE : torch::nn::Module{
  torch::Tensor pad1;
  int batch_size;
  at::TensorOptions this_device;
  CREPE(at::TensorOptions device, int capacity=32)
    : conv1(torch::nn::Conv2dOptions(1, capacity*32, /*kernel_size=*/{512, 1}).stride({4,1}).padding({254,0})),
    conv1_bn(torch::nn::BatchNormOptions(capacity*32)),
    conv2(torch::nn::Conv2dOptions(capacity*32, capacity*4, {64, 1}).padding({31,0})),
    conv2_bn(torch::nn::BatchNormOptions(capacity*4)),
    conv3(torch::nn::Conv2dOptions(capacity*4, capacity*4, {64, 1}).padding({31,0})),
    conv3_bn(torch::nn::BatchNormOptions(capacity*4)),
    conv4(torch::nn::Conv2dOptions(capacity*4, capacity*4, {64, 1}).padding({31,0})),
    conv4_bn(torch::nn::BatchNormOptions(capacity*4)),
    conv5(torch::nn::Conv2dOptions(capacity*4, capacity*8, {64, 1}).padding({31,0})),
    conv5_bn(torch::nn::BatchNormOptions(capacity*8)),
    conv6(torch::nn::Conv2dOptions(capacity*8, capacity*16, {64, 1}).padding({31,0})),
    conv6_bn(torch::nn::BatchNormOptions(capacity*16)),
    fc(torch::nn::LinearOptions(capacity*64, 360)),
    this_device(device) {

      register_module("conv1", conv1);
      register_module("conv1_bn", conv1_bn);
      register_module("conv2", conv2);
      register_module("conv2_bn", conv2_bn);
      register_module("conv3", conv3);
      register_module("conv3_bn", conv3_bn);
      register_module("conv4", conv4);
      register_module("conv4_bn", conv4_bn);
      register_module("conv5", conv5);
      register_module("conv5_bn", conv5_bn);
      register_module("conv6", conv6);
      register_module("conv6_bn", conv6_bn);

      register_module("fc", fc);
      // to(this_device);
  }
```

Rather well documented:

BatchNormOptions:
https://pytorch.org/cppdocs/api/structtorch_1_1nn_1_1_batch_norm_options.html#exhale-struct-structtorch-1-1nn-1-1-batch-norm-options

BatchNorm <-- BatchNormImpl:
https://pytorch.org/cppdocs/api/classtorch_1_1nn_1_1_batch_norm_impl.html#classtorch_1_1nn_1_1_batch_norm_impl

More on creating models:

https://pytorch.org/cppdocs/frontend.html#end-to-end-example

Forward pass

```
torch::Tensor forward(torch::Tensor x) {

batch_size = x.size(0);
x = x.reshape({1, 1, 1024, 1});

    x = torch::relu(conv1->forward(x));
    x = torch::max_pool2d(conv1_bn->forward(x), {2, 1});

    std::vector<torch::Tensor> cc1({x, torch::zeros({batch_size, x.size(1), 1, 1}, this_device)});
    torch::Tensor y1 = at::cat(at::TensorList(cc1), 2);

    x = torch::relu(conv2->forward(y1));
    x = torch::max_pool2d(conv2_bn->forward(x), {2, 1});

    std::vector<torch::Tensor> cc2({x, torch::zeros({batch_size, x.size(1), 1, 1}, this_device)});
    torch::Tensor y2 = at::cat(at::TensorList(cc2), 2);

    x = torch::relu(conv3->forward(y2));
    x = torch::max_pool2d(conv3_bn->forward(x), {2, 1});

    std::vector<torch::Tensor> cc3({x, torch::zeros({batch_size, x.size(1), 1, 1}, this_device)});
    torch::Tensor y3 = at::cat(at::TensorList(cc3), 2);

    x = torch::relu(conv4->forward(y3));
    x = torch::max_pool2d(conv4_bn->forward(x), {2, 1});

    std::vector<torch::Tensor> cc4({x, torch::zeros({batch_size, x.size(1), 1, 1}, this_device)});
    torch::Tensor y4 = at::cat(at::TensorList(cc4), 2);

    x = torch::relu(conv5->forward(y4));
    x = torch::max_pool2d(conv5_bn->forward(x), {2, 1});

    std::vector<torch::Tensor> cc5({x, torch::zeros({batch_size, x.size(1), 1, 1}, this_device)});
    torch::Tensor y5 = at::cat(at::TensorList(cc5), 2);

    x = torch::relu(conv6->forward(y5));
    x = torch::max_pool2d(conv6_bn->forward(x), {2, 1});

    x = x.squeeze(3);
    x = x.permute({0, 2, 1});
    x = x.flatten(1);

    x = torch::sigmoid(fc->forward(x));
    return x;
  }
```

These are documented poorly, but are essential for complicated models

Parsing JIT module to our model

In model definition:

```
torch::nn::Conv2d conv1{nullptr} ...
```

Modules do not contain implementations:

https://pytorch.org/cppdocs/api/classtorch_1_1nn_1_1_batch_norm.html#exhale-class-classtorch-1-1nn-1-1-batch-norm

They are derived from ModuleHolder class

Module-specific loading procedure:

```
void copy_conv2d(torch::nn::Conv2d &module, std::shared_ptr<torch::jit::script::Module> jit_module){
    module->weight.set_requires_grad(false);
    module->weight.copy_(jit_module->get_parameter("weight"));
    module->bias.set_requires_grad(false);
    module->bias.copy_(jit_module->get_parameter("bias"));
}
```

Accessing module implementation

Accessing jit module parameters, poorly documented:

https://pytorch.org/cppdocs/api/structtorch_1_1jit_1_1script_1_1_module.html#exhale-struct-structtorch-1-1jit-1-1script-1-1-module

Loading from jit script:

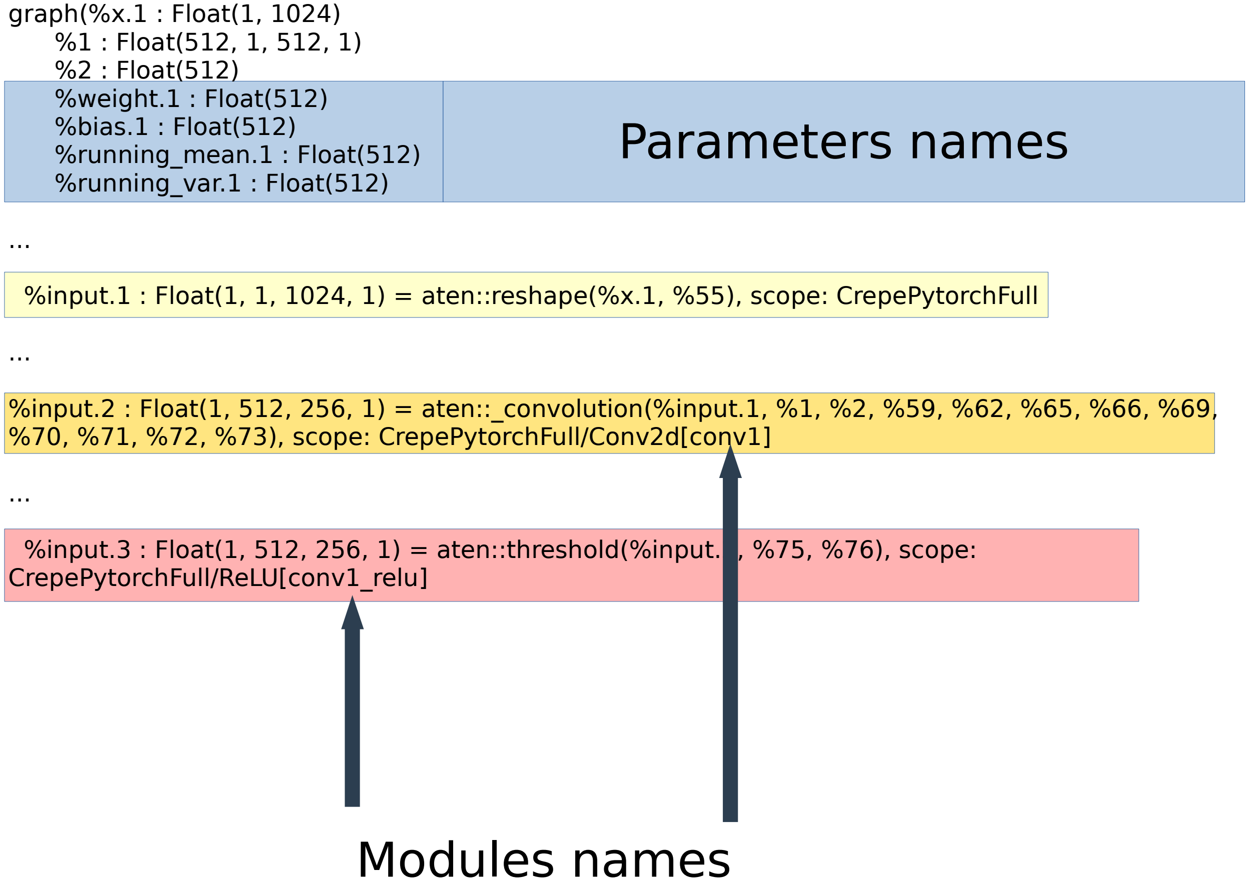
```
std::shared_ptr<torch::jit::script::Module> module = torch::jit::load(module_path);
copy_conv2d(net->conv1, module->get_module("conv1"));
```

JIT where do module names come from

Python code:

```
class CrepePytorchFull(nn.Module):
    def __init__(self, capacity=32):
        super(CrepePytorchFull, self).__init__()
        self.conv1 = nn.Conv2d(1, 32*capacity, ...
        self.conv1_relu = nn.ReLU()
        self.conv1_BN = nn.BatchNorm2d(32*capacity)
        self.conv1_maxpool = nn.MaxPool2d(...
        self.conv1_dropout = nn.Dropout(0.25)
    def forward(self, x):
        batch_size = x.size(0)
        sample_size = x.size(1)
        x = x.reshape(batch_size, 1, 1024, 1)
        x = self.conv1(x)
        x = self.conv1_relu(x)
        x = self.conv1_BN(x)
        x = self.conv1_maxpool(x)
        x = self.conv1_dropout(x)
```

JIT trace:



```
graph(%x.1 : Float(1, 1024)
      %1 : Float(512, 1, 512, 1)
      %2 : Float(512)
      %weight.1 : Float(512)
      %bias.1 : Float(512)
      %running_mean.1 : Float(512)
      %running_var.1 : Float(512)
...
 %input.1 : Float(1, 1, 1024, 1) = aten::reshape(%x.1, %55), scope: CrepePytorchFull
...
%input.2 : Float(1, 512, 256, 1) = aten::_convolution(%input.1, %1, %2, %59, %62, %65, %66, %69,
%70, %71, %72, %73), scope: CrepePytorchFull/Conv2d[conv1]
...
 %input.3 : Float(1, 512, 256, 1) = aten::threshold(%input.2, %75, %76), scope:
CrepePytorchFull/ReLU[conv1_relu]
```

Parameters names

Modules names

Compiling with CMake

CMake:

```
cmake_minimum_required(VERSION 3.0 FATAL_ERROR)
project(custom_ops)
find_package(Torch REQUIRED)
add_executable(example-app example-app.cpp)
target_link_libraries(example-app "${TORCH_LIBRARIES}")
set_property(TARGET example-app PROPERTY CXX_STANDARD 11)
```

```
mkdir build
cd build
cmake -DCMAKE_PREFIX_PATH=/path/to/libtorch ..
make
```

Example from https://pytorch.org/tutorials/advanced/cpp_export.html

When compiling without CUDA it uses -Wl –as-needed option for CUDA-dependent libraries.

Compiling using SConstruct

```
# SCsub
import os
Import('env')

torch_rlibdirs = [
    "/home/lupoglaz/Projects/godot/modules/pitchdetector/libtorch/lib",
    "/usr/local/cuda-10.0/lib64"
]
```

Runtime library directories

```
torch_libdirs = [
    "/home/lupoglaz/Projects/godot/modules/pitchdetector/libtorch/lib",
    "/usr/local/cuda-10.0/lib64",
    "/home/lupoglaz/Projects/godot/modules/pitchdetector/build/Audio",
    "/usr/local/cuda-10.0/lib64",
    "/usr/lib/x86_64-linux-gnu",
    "/usr/local/cuda/lib64",
]

torch_libnames = [  "PITCH", "asound", "m", "torch", "caffe2", "caffe2_gpu", "c10", "c10_cuda",
                    "cufft", "curand", "cudnn", "culibos", "cublas", "cuda", "nvrtc",
                    "nvToolsExt", "cudart_static", "pthread", "dl", "rt", "X11"]
```

Libraries that we need

```
env_sum = env.Clone()
env_sum.Append(LINKFLAGS=['-Wl,--no-undefined', '-Wl,--no-as-needed'])
env_sum.Append(CPPPATH=torch_include)
env_sum.Append(LIBPATH=torch_libdirs)
env_sum.Append(RPATH=torch_rlibdirs)
env_sum.Append(LIBS=torch_libnames)
env_sum.Append(CPPDEFINES={'-D_GLIBCXX_USE_CXX11_ABI' : 0 })
env_sum.Append(CXXFLAGS=['-O2', '-Wall', '-std=c++11'])
```

Important option

```
src_list = ["pitchdetector.cpp", "register_types.cpp", "Audio/cAlsaIO.cpp", "Audio/cNNPitchDetector.cpp"]
env_sum.add_source_files(env.modules_sources, src_list)

env.Append(LINKFLAGS=['-Wl,--no-undefined', '-Wl,--no-as-needed'])
env.Append(LIBPATH=torch_libdirs)
env.Append(RPATH=torch_rlibdirs)
env.Append(LIBS=torch_libnames)
```

Linking libraries to the global environment

Demo

Thanks