The C++ Frontend

The PyTorch C++ frontend is a C++14 library for CPU and GPU tensor computation, with automatic differentiation and high level building blocks for state of the art machine learning applications.

Description

The PyTorch C++ frontend can be thought of as a C++ version of the PyTorch Python frontend, providing automatic differentiation and various higher level abstractions for machine learning and neural networks. Specifically, it consists of the following components:

Component	Description
`torch::Tensor`	Automatically differentiable, efficient CPU and GPU enabled tensors
`torch::nn`	A collection of composable modules for neural network modeling
`torch::optim`	Optimization algorithms like SGD, Adam or RMSprop to train your models
`torch::data`	Datasets, data pipelines and multi-threaded, asynchronous data loader
`torch::serialize`	A serialization API for storing and loading model checkpoints
`torch::python`	Glue to bind your C++ models into Python
`torch::jit`	Pure C++ access to the TorchScript JIT compiler

End-to-end example

Here is a simple, end-to-end example of defining and training a simple neural network on the MNIST dataset:

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

// Define a new Module.
struct Net : torch::nn::Module {
  Net() {
    // Construct and register two Linear submodules.
    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));
  }

  // Implement the Net's algorithm.
  torch::Tensor forward(torch::Tensor x) {
    // Use one of many tensor manipulation functions.
    x = torch::relu(fc1->forward(x.reshape({x.size(0), 784})));
    x = torch::dropout(x, /*p=*/0.5, /*train=*/is_training());
    x = torch::relu(fc2->forward(x));
    x = torch::log_softmax(fc3->forward(x), /*dim=*/1);
    return x;
  }

  // Use one of many "standard library" modules.
  torch::nn::Linear fc1{nullptr}, fc2{nullptr}, fc3{nullptr};
};

int main() {
  // Create a new Net.
  auto net = std::make_shared<Net>();

  // Create a multi-threaded data loader for the MNIST dataset.
  auto data_loader = torch::data::make_data_loader(
      torch::data::datasets::MNIST("./data").map(
          torch::data::transforms::Stack<>()),
      /*batch_size=*/64);

  // Instantiate an SGD optimization algorithm to update our Net's parameters.
  torch::optim::SGD optimizer(net->parameters(), /*lr=*/0.01);

  for (size_t epoch = 1; epoch <= 10; ++epoch) {
    size_t batch_index = 0;
    // Iterate the data loader to yield batches from the dataset.
    for (auto& batch : *data_loader) {
      // Reset gradients.
      optimizer.zero_grad();
      // Execute the model on the input data.
      torch::Tensor prediction = net->forward(batch.data);
      // Compute a loss value to judge the prediction of our model.
      torch::Tensor loss = torch::nll_loss(prediction, batch.target);
      // Compute gradients of the loss w.r.t. the parameters of our model.
      loss.backward();
```

```
      // Update the parameters based on the calculated gradients.
      optimizer.step();
      // Output the loss and checkpoint every 100 batches.
      if (++batch_index % 100 == 0) {
        std::cout << "Epoch: " << epoch << " | Batch: " << batch_index
                  << " | Loss: " << loss.item<float>() << std::endl;
        // Serialize your model periodically as a checkpoint.
        torch::save(net, "net.pt");
      }
    }
  }
}
```

To see more complete examples of using the PyTorch C++ frontend, see the example repository.

Philosophy

PyTorch's C++ frontend was designed with the idea that the Python frontend is great, and should be used when possible; but in some settings, performance and portability requirements make the use of the Python interpreter infeasible. For example, Python is a poor choice for low latency, high performance or multithreaded environments, such as video games or production servers. The goal of the C++ frontend is to address these use cases, while not sacrificing the user experience of the Python frontend.

As such, the C++ frontend has been written with a few philosophical goals in mind:

- **Closely model the Python frontend in its design**, naming, conventions and functionality. While there may be occasional differences between the two frontends (e.g., where we have dropped deprecated features or fixed "warts" in the Python frontend), we guarantee that the effort in porting a Python model to C++ should lie exclusively in **translating language features**, not modifying functionality or behavior.
- **Prioritize flexibility and user-friendliness over micro-optimization.** In C++, you can often get optimal code, but at the cost of an extremely unfriendly user experience. Flexibility and dynamism is at the heart of PyTorch, and the C++ frontend seeks to preserve this experience, in some cases sacrificing performance (or "hiding" performance knobs) to keep APIs simple and explicable. We want researchers who don't write C++ for a living to be able to use our APIs.

A word of warning: Python is not necessarily slower than C++! The Python frontend calls into C++ for almost anything computationally expensive (especially any kind of numeric operation), and these operations will take up the bulk of time spent in a program. If you would prefer to write Python, and can afford to write Python, we recommend using the Python interface to PyTorch. However, if you would prefer to write C++, or need to write C++ (because of multithreading, latency or deployment requirements), the C++ frontend to PyTorch provides an API that is approximately as convenient, flexible, friendly and intuitive as its Python counterpart. The two frontends serve different use cases, work hand in hand, and neither is meant to unconditionally replace the other.

Installation

Instructions on how to install the C++ frontend library distribution, including an example for how to build a minimal application depending on LibTorch, may be found by following this link.