- PyTorch C++ API

These pages provide the documentation for the public portions of the PyTorch C++ API. This API can roughly be divided into five parts:

- **ATen**: The foundational tensor and mathematical operation library on which all else is built.
- **Autograd**: Augments ATen with automatic differentiation.
- **C++ Frontend**: High level constructs for training and evaluation of machine learning models.
- **TorchScript**: An interface to the TorchScript JIT compiler and interpreter.
- **C++ Extensions**: A means of extending the Python API with custom C++ and CUDA routines.

Combining, these building blocks form a research and production ready C++ library for tensor computation and dynamic neural networks with strong emphasis on GPU acceleration as well as fast CPU performance. It is currently in use at Facebook in research and production; we are looking forward to welcome more users of the PyTorch C++ API.

Warning

At the moment, the C++ API should be considered “beta” stability; we may make major breaking changes to the backend in order to improve the API, or in service of providing the Python interface to PyTorch, which is our most stable and best supported interface.

ATen

ATen is fundamentally a tensor library, on top of which almost all other Python and C++ interfaces in PyTorch are built. It provides a core `Tensor` class, on which many hundreds of operations are defined. Most of these operations have both CPU and GPU implementations, to which the `Tensor` class will dynamically dispatch based on its type. A small example of using ATen could look as follows:

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

at::Tensor a = at::ones({2, 2}, at::kInt);
at::Tensor b = at::randn({2, 2});
auto c = a + b.to(at::kInt);
```

This `Tensor` class and all other symbols in ATen are found in the `at::` namespace, documented here.

Autograd

What we term *autograd* are the portions of PyTorch's C++ API that augment the ATen `Tensor` class with capabilities concerning automatic differentiation. The autograd system records operations on tensors to form an *autograd graph*. Calling `backwards()` on a leaf variable in this graph performs reverse mode differentiation through the network of functions and tensors spanning the autograd graph, ultimately yielding gradients. The following example provides a taste of this interface:

```
#include <torch/csrc/autograd/variable.h>
#include <torch/csrc/autograd/function.h>

torch::Tensor a = torch::ones({2, 2}, torch::requires_grad());
torch::Tensor b = torch::randn({2, 2});
auto c = a + b;
c.backward(); // a.grad() will now hold the gradient of c w.r.t. a.
```

The `at::Tensor` class in ATen is not differentiable by default. To add the differentiability of tensors the autograd API provides, you must use tensor factory functions from the *torch::* namespace instead of the *at::* namespace. For example, while a tensor created with *at::ones* will not be differentiable, a tensor created with *torch::ones* will be.

C++ Frontend

The PyTorch C++ frontend provides a high level, pure C++ modeling interface for neural network and general ML(Machine Learning) research and production use cases, largely following the Python API in design and provided functionality. The C++ frontend includes the following:

- An interface for defining machine learning models through a hierarchical module system (like `torch.nn.Module`);

- A “standard library” of pre-existing modules for the most common modeling purposes (e.g. convolutions, RNNs, batch normalization etc.);
- An optimization API, including implementations of popular optimizers such as SGD, Adam, RMSprop and others;
- A means of representing datasets and data pipelines, including functionality to load data in parallel over many CPU cores;
- A serialization format for storing and loading checkpoints of a training session (like `torch.utils.data.DataLoader`);
- Automatic parallelization of models onto multiple GPUs (like `torch.nn.parallel.DataParallel`);
- Support code to easily bind C++ models into Python using pybind11;
- Entry points to the TorchScript JIT compiler;
- Helpful utilities to facilitate interfacing with the ATen and Autograd APIs.

See this document for a more detailed description of the C++ frontend. Relevant sections of the *torch::* namespace related to the C++ Frontend include torch::nn, torch::optim, torch::data, torch::serialize, torch::jit and torch::python. Examples of the C++ frontend can be found in this repository which is being expanded on a continuous and active basis.

Note

Unless you have a particular reason to constrain yourself exclusively to ATen or the Autograd API, the C++ frontend is the recommended entry point to the PyTorch C++ ecosystem. While it is still in beta as we collect user feedback (from you!), it provides both more functionality and better stability guarantees than the ATen and Autograd APIs.

TorchScript

TorchScript is a representation of a PyTorch model that can be understood, compiled and serialized by the TorchScript compiler. Fundamentally, TorchScript is a programming language in its own right. It is a subset of Python using the PyTorch API. The C++ interface to TorchScript encompasses three primary pieces of functionality:

- A mechanism for loading and executing serialized TorchScript models defined in Python;
- An API for defining custom operators that extend the TorchScript standard library of operations;
- Just-in-time compilation of TorchScript programs from C++.

The first mechanism may be of great interest to you if you would like to define your models in Python as much as possible, but subsequently export them to C++ for production environments and no-Python inference. You can find out more about this by following this link. The second API concerns itself with scenarios in which you would like to extend TorchScript with custom operators, which can similarly be serialized and invoked from C++ during inference. Lastly, the torch::jit::compile function may be used to access the TorchScript compiler directly from C++.

C++ Extensions

C++ Extensions offer a simple yet powerful way of accessing all of the above interfaces for the purpose of extending regular Python use-cases of PyTorch. C++ extensions are most commonly used to implement custom operators in C++ or CUDA to accelerate research in vanilla PyTorch setups. The C++ extension API does not add any new functionality to the PyTorch C++ API. Instead, it provides integration with Python setuptools as well as JIT compilation mechanisms that allow access to ATen, the autograd and other C++ APIs from Python. To learn more about the C++ extension API, go through this tutorial.

Contents

Indices and tables

- Index
- Module Index
- Search Page

Acknowledgements

This documentation website for the PyTorch C++ universe has been enabled by the Exhale project and generous investment of time and effort by its maintainer, svenevs. We thank Stephen for his work and his efforts providing help with the PyTorch C++ documentation.