torch::deploy

`torch::deploy` is a system that allows you to run multiple embedded Python interpreters in a C++ process without a shared global interpreter lock. For more information on how `torch::deploy` works internally, please see the related arXiv paper.

> **Warning**
>
> This is a prototype feature. Only Linux x86 is supported, and the API may change without warning.

Getting Started

Installing `torch::deploy`

`torch::deploy` is not yet built by default in our binary releases, so to get a copy of libtorch with `torch::deploy` enabled, follow the instructions for building PyTorch from source.

When running `setup.py`, you will need to specify `USE_DEPLOY=1`, like:

```
export CMAKE_PREFIX_PATH=${CONDA_PREFIX:-"$(dirname $(which conda))/../"}
export USE_DEPLOY=1
python setup.py bdist_wheel
python -mpip install dist/*.whl
```

Creating a model package in Python

`torch::deploy` can load and run Python models that are packaged with `torch.package`. You can learn more about `torch.package` in the `torch.package` documentation.

For now, let's create a simple model that we can load and run in `torch::deploy`.

```
from torch.package import PackageExporter
import torchvision

# Instantiate some model
model = torchvision.models.resnet.resnet18()

# Package and export it.
with PackageExporter("my_package.pt") as e:
    e.intern("torchvision.**")
    e.extern("sys")
    e.save_pickle("model", "model.pkl", model)
```

Now, there should be a file named `my_package.pt` in your working directory.

> **Note**
>
> Currently, `torch::deploy` supports only the Python standard library and `torch` as `extern` modules in `torch.package`. In the future we plan to transparently support any Conda environment you point us to.

Loading and running the model in C++

Let's create a minimal C++ program to that loads the model.

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

#include <iostream>
#include <memory>

int main(int argc, const char* argv[]) {
    if (argc != 2) {
        std::cerr << "usage: example-app <path-to-exported-script-module>\n";
        return -1;
    }

    // Start an interpreter manager governing 4 embedded interpreters.
    torch::deploy::InterpreterManager manager(4);

    try {
        // Load the model from the torch.package.
        torch::deploy::Package package = manager.loadPackage(argv[1]);
        torch::deploy::ReplicatedObj model = package.loadPickle("model", "model.pkl");
    } catch (const c10::Error& e) {
        std::cerr << "error loading the model\n";
        return -1;
    }

    std::cout << "ok\n";
}
```

This small program introduces many of the core concepts of `torch::deploy`.

An `InterpreterManager` abstracts over a collection of independent Python interpreters, allowing you to load balance across them when running your code.

Using the `InterpreterManager::loadPackage` method, you can load a `torch.package` from disk and make it available to all interpreters.

`Package::loadPickle` allows you to retrieve specific Python objects from the package, like the ResNet model we saved earlier.

Finally, the model itself is a `ReplicatedObj`. This is an abstract handle to an object that is replicated across multiple interpreters. When you interact with a `ReplicatedObj` (for example, by calling `forward`), it will select an free interpreter to execute that interaction.

Building and running the application

Assuming the above C++ program was stored in a file called, *example-app.cpp*, a minimal CMakeLists.txt file would look like:

```
cmake_minimum_required(VERSION 3.0 FATAL_ERROR)
project(deploy_tutorial)

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 14)
```

The last step is configuring and building the project. Assuming that our code directory is laid out like this:

System Message: WARNING/2 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)deploy.rst`, line 141)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

    example-app/
        CMakeLists.txt
        example-app.cpp
```

We can now run the following commands to build the application from within the `example-app/` folder:

```
mkdir build
cd build
# Point CMake at the built version of PyTorch we just installed.
SITE_PACKAGES="$(python -c 'from distutils.sysconfig import get_python_lib; print(get_python_lib())')"
cmake -DCMAKE_PREFIX_PATH="$SITE_PACKAGES/torch" ..
cmake --build . --config Release
```

Now we can run our app:

```
./example-app /path/to/my_package.pt
```

Executing `forward` in C++

One you have your model loaded in C++, it is easy to execute it:

```
// Create a vector of inputs.
std::vector<torch::jit::IValue> inputs;
inputs.push_back(torch::ones({1, 3, 224, 224}));

// Execute the model and turn its output into a tensor.
at::Tensor output = model(inputs).toTensor();
std::cout << output.slice(/*dim=*/1, /*start=*/0, /*end=*/5) << '\n';
```

Notably, the model's forward function is executing in Python, in an embedded CPython interpreter. Note that the model is a `ReplicatedObj`, which means that you can call `model()` from multiple threads and the forward method will be executed on multiple independent interpreters, with no global interpreter lock.