Extending TorchScript with Custom C++ Operators

使用自定义C++ 运算符扩展 TorchScript

> 作者：PyTorch
>
> 译者：ApacheCN

PyTorch1.0版本向 PyTorch 引入了一个名为 [TorchScript](https://pytorch.org/docs/master/jit.html）的新编程模型。TorchScript是Python编程语言的一个子集，可以通过TorchScript编译器进行解析，编译和优化。此外，已编译的TorchScript模型可以选择序列化为磁盘文件格式，您可以随后从纯C++(以及Python）程序进行加载和运行以进行推理。

TorchScript支持由 `torch` 包提供的大量操作，允许您将多种复杂模型纯粹表示为PyTorch的“标准库”中的一系列张量运算。然而，有时您可能会发现需要使用自定义C ++或CUDA函数扩展TorchScript。虽然我们建议您使用此这个选项时只在您的想法无法(足够有效）表达为一个简单的Python函数时，我们确实提供了一个非常友好和简单的界面，使用ATen 定义自定义C++ 和 CUDA 内核，PyTorch 的高性能C ++张量库。一旦绑定到TorchScript，您可以将这些自定义内核(或“ops”)嵌入到您的TorchScript模型中，并以Python和c++的序列化形式直接执行它们。

以下段落给出了一个编写TorchScript自定义操作的示例，以调用[OpenCV](https://www.opencv.org），这是一个用C ++编写的计算机视觉库。我们将讨论如何在C ++中使用张量，如何有效地将它们转换为第三方张量格式(在这种情况下，OpenCV [`](#id1）Mat` s），如何在TorchScript运行时注册运算符，最后如何编译运算符并在Python和C ++中使用它。

本教程假设您通过 `pip` 或 `conda` 安装了PyTorch 1.0的*preview release*。有关获取最新版PyTorch 1.0 \的说明，请参阅 [https://pytorch.org/get-started/locally](https://pytorch.org/get-started/locally）。或者，您可以从源代码编译PyTorch。[此文件](https://github.com/pytorch/pytorch/blob/master/CONTRIBUTING.md）中的文档将为您提供帮助。

在C ++中实现自定义运算符

对于本教程，我们将公开[warpPerspective](https://docs.opencv.org/2.4/modules/imgproc/doc/geometric_transformations.html#warpperspective）函数，该函数将透视变换应用于图像，OpenCV to TorchScript作为自定义运算符。第一步是用C ++编写自定义运算符的实现。让我们调用这个实现 `op.cpp` 的文件，并使它看起来像这样：

```
torch::Tensor warp_perspective(torch::Tensor image, torch::Tensor warp) {
  cv::Mat image_mat(/*rows=*/image.size(0),
                    /*cols=*/image.size(1),
                    /*type=*/CV_32FC1,
                    /*data=*/image.data<float>());
  cv::Mat warp_mat(/*rows=*/warp.size(0),
                   /*cols=*/warp.size(1),
                   /*type=*/CV_32FC1,
                   /*data=*/warp.data<float>());

  cv::Mat output_mat;
  cv::warpPerspective(image_mat, output_mat, warp_mat, /*dsize=*/{8, 8});

  torch::Tensor output = torch::from_blob(output_mat.ptr<float>(), /*sizes=*/{8, 8});
  return output.clone();
}
```

该运算符的代码很短。在文件的顶部，我们包含OpenCV头文件 `opencv2/opencv.hpp` ，以及 `torch/script.h` 头文件，它展示了我们需要编写自定义TorchScript运算符的PyTorch C++ API所需的所有好东西.我们的函数 `warp_perspective` 有两个参数：输入 `image` 和我们希望应用于图像的 `warp` 变换矩阵。这些输入的类型是 `torch::Tensor` ，PyTorch在C++中的张量类型(它也是Python中所有张量的基础类型）。我们的 `warp_perspective` 函数的返回类型也将是 `torch::Tensor` 。

提示

有关ATen的更多信息，请参阅[本说明](https://pytorch.org/cppdocs/notes/tensor_basics.html），ATen是为PyTorch提供`Tensor`类的库。此外，[本教程](https://pytorch.org/cppdocs/notes/tensor_creation.html）描述了如何在C ++中分配和初始化新的张量对象(此运算符不需要）。

注意

TorchScript编译器了解固定数量的类型。只有这些类型可以用作自定义运算符的参数。目前这些类型是： `torch::Tensor` ， `torch::Scalar` ， `double` ， `int64_t` 和 `std::vector` 这些类型。注意**only** `double` 和**not** `float` 和**only** `int64_t` 和**not**支持其他整数类型， `int` ， `short` 或 `long` 。

在我们的函数内部，我们需要做的第一件事是将PyTorch张量转换为OpenCV矩阵，因为OpenCV的 `warpPerspective` 期望 `cv::Mat` 对象作为输入。幸运的是，有一种方法可以做到这一点**而无需复制任何**数据。在前几行中，

```
cv::Mat image_mat(/*rows=*/image.size(0),
                  /*cols=*/image.size(1),
                  /*type=*/CV_32FC1,
                  /*data=*/image.data<float>());
```

我们正在调用OpenCV `Mat` 类的 这个构造函数 将我们的张量转换为 `Mat` 对象。我们传递原始 `image` tensor的行数和列数，数据类型(我们将在本例中将其定义为 `float32` ），最后是一个指向底层数据的原始指针 - 一个 `float *` 。 `Mat` 类的这个构造函数的特殊之处在于它不复制输入数据。相反，它将简单地为在“Mat”上执行的所有操作引用该内存。如果对 `image_mat` 执行in-place操作，则这将反映在原始 `image` 张量中(反之亦然）。这允许我们使用库的本机矩阵类型调用后续的OpenCV例程，即使我们实际上将数据存储在PyTorch张量中。我们重复此过程将 `warp` PyTorch张量转换为 `warp_mat` OpenCV矩阵：

```
cv::Mat warp_mat(/*rows=*/warp.size(0),
                 /*cols=*/warp.size(1),
                 /*type=*/CV_32FC1,
                 /*data=*/warp.data<float>());
```

接下来，我们准备调用我们非常渴望在TorchScript中使用的OpenCV函数： `warpPerspective` 。为此，我们传递OpenCV函数 `image_mat` 和 `warp_mat` 矩阵，以及一个名为 `output_mat` 的空输出矩阵。我们还指定了我们想要输出矩阵(图像）的大小 `dsize` 。对于此示例，它被硬编码为 `8 x 8` ：

```
cv::Mat output_mat;
cv::warpPerspective(image_mat, output_mat, warp_mat, /*dsize=*/{8, 8});
```

我们的自定义运算符实现的最后一步是将 `output_mat` 转换回PyTorch张量，以便我们可以在PyTorch中进一步使用它。这与我们之前在另一个方向转换时所做的惊人相似。PyTorch提供了一个 `torch::from_blob` 方法。在本例中，*blob*意为指向内存的一些不透明的平面指针，我们想将其解释为PyTorch张量。对 `torch::from_blob` 的调用如下所示：

```
torch::from_blob(output_mat.ptr<float>(), /*sizes=*/{8, 8})
```

我们在OpenCV `Mat` 类上使用 `.ptr&lt;float&gt;()` 方法来获取指向底层数据的原始指针(就像之前的PyTorch张量的 `.data&lt;float&gt;()` 一样）。我们还指定了张量的输出形状，我们将其硬编码为 `8 x 8` 。然后 `torch::from_blob` 的输出为 `torch::Tensor` ，指向OpenCV矩阵拥有的内存。

在从运算符实现返回此张量之前，我们必须在张量上调用 `.clone()` 来执行基础数据的内存复制。原因是 `torch::from_blob` 返回不拥有其数据的张量。此时，数据仍归OpenCV矩阵所有。但是，此OpenCV矩阵将超出范围并在函数末尾取消分配。如果我们按原样返回 `output` 张量，那么当我们在函数外部使用它时，它将指向无效的内存。调用 `.clone()` 返回一个新的张量，其中包含新张量所拥有的原始数据的副本。因此返回外部世界是安全的。

使用TorchScript注册自定义运算符

现在已经在C ++中实现了我们的自定义运算符，我们需要*使用TorchScript运行时和编译器注册它*。这将允许TorchScript编译器在TorchScript代码中解析对自定义运算符的引用。注册非常简单。对于我们的情况，我们需要写：

```
static auto registry =
  torch::jit::RegisterOperators("my_ops::warp_perspective", &warp_perspective);
```

在我们的 `op.cpp` 文件的全局范围内的某个地方。这将创建一个全局变量 `registry` ，它将在其构造函数中使用TorchScript注册我们的运算符(即每个程序只注册一次）。我们指定运算符的名称，以及指向其实现的指针(我们之前编写的函数）。该名称由两部分组成：*namespace*(`my_ops` ）和我们正在注册的特定运算符的名称(`warp_perspective` ）。命名空间和运算符名称由两个冒号(`::` ）分隔。

注意

如果要注册多个运算符，可以在构造函数之后将调用链接到 `.op()` ：

```
static auto registry =
  torch::jit::RegisterOperators("my_ops::warp_perspective", &warp_perspective)
  .op("my_ops::another_op", &another_op)
  .op("my_ops::and_another_op", &and_another_op);
```

在后台， `RegisterOperators` 将执行一些相当复杂的C++模板元编程魔术技巧来推断我们传递它的函数指针的参数和返回值类型(`&warp_perspective` ）。该信息用于为我们的运营商形成*function schema*。函数模式是运算符的结构化表示 - 一种“签名”或“原型” - 由TorchScript 编译器用于验证 TorchScript程序中的正确性。

构建自定义运算符

现在我们已经用C++ 实现了我们的自定义运算符并编写了它的注册代码，现在是时候将运算符构建到一个(共享）库中，我们可以将它加载到Python中进行研究和实验，或者加载到C ++中以便在非Python中进行推理环境。使用纯CMake或像 `setuptools` 这样的Python替代方法，存在多种构建运算符的方法。为简洁起见，以下段落仅讨论CMake方法。本教程的附录深入研究了基于Python的替代方案。

用CMake建设

要使用 CMake 构建系统将自定义运算符构建到共享库中，我们需要编写一个简短的 `CMakeLists.txt` 文件并将其与我们之前的 `op.cpp` 文件放在一起。为此，让我们商定一个如下所示的目录结构：

```
warp-perspective/
  op.cpp
  CMakeLists.txt
```

此外，请确保从 pytorch.org 获取最新版本的LibTorch发行版，该发行版包含PyTorch的C++ 库和 CMake 构建文件。将解压缩的分发放在文件系统中可访问的位置。以下段落将该位置称为 `/path/to/libtorch` 。我们的 `CMakeLists.txt` 文件的内容应该如下：

```
cmake_minimum_required(VERSION 3.1 FATAL_ERROR)
project(warp_perspective)

find_package(Torch REQUIRED)
find_package(OpenCV REQUIRED)

# Define our library target
add_library(warp_perspective SHARED op.cpp)
# Enable C++11
target_compile_features(warp_perspective PRIVATE cxx_range_for)
# Link against LibTorch
target_link_libraries(warp_perspective "${TORCH_LIBRARIES}")
# Link against OpenCV
target_link_libraries(warp_perspective opencv_core opencv_imgproc)
```

警告

此设置对构建环境做出一些假设，特别是与OpenCV的安装有关的内容。上面的 `CMakeLists.txt` 文件在运行Ubuntu Xenial的Docker容器内进行了测试，并通过 `apt` 安装了 `libopencv-dev` 。如果它不适合您并且您感到卡住，请使用随附教程库中的 `Dockerfile` 构建一个隔离的，可重现的环境，在其中使用本教程中的代码。如果您遇到进一步的麻烦，请在教程库中提出问题或在我们的论坛中发帖提问。

要现在构建我们的运算符，我们可以从 `warp_perspective` 文件夹运行以下命令：

```
$ mkdir build
$ cd build
$ cmake -DCMAKE_PREFIX_PATH=/path/to/libtorch ..
-- The C compiler identification is GNU 5.4.0
-- The CXX compiler identification is GNU 5.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Looking for pthread.h
-- Looking for pthread.h - found
-- Looking for pthread_create
-- Looking for pthread_create - not found
-- Looking for pthread_create in pthreads
-- Looking for pthread_create in pthreads - not found
-- Looking for pthread_create in pthread
-- Looking for pthread_create in pthread - found
-- Found Threads: TRUE
-- Found torch: /libtorch/lib/libtorch.so
-- Configuring done
-- Generating done
-- Build files have been written to: /warp_perspective/build
$ make -j
Scanning dependencies of target warp_perspective
[ 50%] Building CXX object CMakeFiles/warp_perspective.dir/op.cpp.o
[100%] Linking CXX shared library libwarp_perspective.so
[100%] Built target warp_perspective
```

这将在 `build` 文件夹中放置一个 `libwarp_perspective.so` 共享库文件。在上面的 `cmake` 命令中，您应该将 `/path/to/libtorch` 替换为解压缩的LibTorch分发的路径。

我们将在下面详细介绍如何使用和调用我们的运算符，但为了尽早获得成功感，我们可以尝试在Python中运行以下代码：

```
>>> import torch
>>> torch.ops.load_library("/path/to/libwarp_perspective.so")
>>> print(torch.ops.my_ops.warp_perspective)
```

这里，`/path/to/libwarp_perspective.so` 应该是我们刚刚构建的 `libwarp_perspective.so` 共享库的相对路径或绝对路径。如果一切顺利，这应该打印出类似的东西

```
<built-in method my_ops::warp_perspective of PyCapsule object at 0x7f618fc6fa50>
```

这是我们稍后用来调用我们的自定义运算符的Python函数。

在Python中使用TorchScript自定义运算符

一旦我们的自定义运算符内置到共享库中，我们就可以在Python的TorchScript模型中使用此运算符。这有两个部分：首先将运算符加载到Python中，然后在TorchScript代码中使用运算符。

您已经了解了如何将运算符导入Python： `torch.ops.load_library()` 。此函数获取包含自定义运算符的共享库的路径，并将其加载到当前进程中。加载共享库还将执行我们放入自定义运算符实现文件的全局 `RegisterOperators` 对象的构造函数。这将使用TorchScript编译器注册我们的自定义运算符，并允许我们在TorchScript代码中使用该运算符。

您可以将加载的运算符称为 `torch.ops.<namespace>.<function>` ，其中 `<namespace>` 是运算符名称的名称空间部分， `<function>` 是运算符的函数名称。对于我们上面写的运算符，命名空间是 `my_ops` 和函数名 `warp_perspective` ，这意味着我们的运算符可用作 `torch.ops.my_ops.warp_perspective` 。虽然此函数可用于脚本或跟踪的TorchScript模块，但我们也可以在vanilla eager PyTorch中使用它并将其传递给常规的PyTorch张量：

```
>>> import torch
>>> torch.ops.load_library("libwarp_perspective.so")
>>> torch.ops.my_ops.warp_perspective(torch.randn(32, 32), torch.rand(3, 3))
tensor([[0.0000, 0.3218, 0.4611,  ..., 0.4636, 0.4636, 0.4636],
        [0.3746, 0.0978, 0.5005,  ..., 0.4636, 0.4636, 0.4636],
        [0.3245, 0.0169, 0.0000,  ..., 0.4458, 0.4458, 0.4458],
        ...,
        [0.1862, 0.1862, 0.1692,  ..., 0.0000, 0.0000, 0.0000],
        [0.1862, 0.1862, 0.1692,  ..., 0.0000, 0.0000, 0.0000],
        [0.1862, 0.1862, 0.1692,  ..., 0.0000, 0.0000, 0.0000]])
```

注意

幕后发生的事情是，第一次在Python中访问 `torch.ops.namespace.function` 时，TorchScript编译器(在C ++版本中）将查看是否已注册函数 `namespace::function` ，如果已注册，则返回此函数的Python句柄，我们随后可以使用从Python调用我们的C ++运算符实现。这是TorchScript自定义运算符和C ++扩展之间的一个值得注意的区别：C ++扩展使用pybind11手动绑定，而TorchScript自定义ops由PyTorch本身绑定。 Pybind11为您提供了更多关于可以绑定到Python的类型和类的灵活性，因此建议用于纯粹的热切代码，但TorchScript操作不支持它。

从这里开始，您可以在脚本或跟踪代码中使用自定义运算符，就像使用 `torch` 包中的其他函数一样。实际上，像 `torch.matmul` 这样的“标准库”函数与自定义运算符的注册路径大致相同，这使得自定义运算符在TorchScript中的使用方式和位置方面确实是一流公民。

使用带有跟踪的自定义运算符

让我们首先将运算符嵌入到跟踪函数中。回想一下，对于跟踪，我们从一些香草Pytorch代码开始：

```
def compute(x, y, z):
    return x.matmul(y) + torch.relu(z)
```

然后在其上调用 `torch.jit.trace` 。我们进一步传递 `torch.jit.trace` 一些示例输入，它将转发到我们的实现，以记录输入流经它时发生的操作序列。这样做的结果实际上是热切的PyTorch程序的“冻结”版本，TorchScript编译器可以进一步分析，优化和序列化：

```
>>> inputs = [torch.randn(4, 8), torch.randn(8, 5), torch.randn(4, 5)]
>>> trace = torch.jit.trace(compute, inputs)
>>> print(trace.graph)
graph(%x : Float(4, 8)
 %y : Float(8, 5)
 %z : Float(4, 5)) {
 %3 : Float(4, 5) = aten::matmul(%x, %y)
 %4 : Float(4, 5) = aten::relu(%z)
 %5 : int = prim::Constant[value=1]()
 %6 : Float(4, 5) = aten::add(%3, %4, %5)
 return (%6);
}
```

现在，令人兴奋的启示是我们可以简单地将我们的自定义运算符放入我们的PyTorch跟踪中，就像它是 `torch.relu` 或任何其他 `torch` 函数一样：

```
torch.ops.load_library("libwarp_perspective.so")

def compute(x, y, z):
    x = torch.ops.my_ops.warp_perspective(x, torch.eye(3))
    return x.matmul(y) + torch.relu(z)
```

然后跟踪它：

```
>>> inputs = [torch.randn(4, 8), torch.randn(8, 5), torch.randn(8, 5)]
>>> trace = torch.jit.trace(compute, inputs)
>>> print(trace.graph)
graph(%x.1 : Float(4, 8)
 %y : Float(8, 5)
 %z : Float(8, 5)) {
 %3 : int = prim::Constant[value=3]()
 %4 : int = prim::Constant[value=6]()
 %5 : int = prim::Constant[value=0]()
 %6 : int[] = prim::Constant[value=[0, -1]]()
 %7 : Float(3, 3) = aten::eye(%3, %4, %5, %6)
 %x : Float(8, 8) = my_ops::warp_perspective(%x.1, %7)
 %11 : Float(8, 5) = aten::matmul(%x, %y)
 %12 : Float(8, 5) = aten::relu(%z)
 %13 : int = prim::Constant[value=1]()
 %14 : Float(8, 5) = aten::add(%11, %12, %13)
 return (%14);
 }
```

将TorchScript自定义操作集成到跟踪的PyTorch代码就像这样简单！

使用自定义操作符和脚本

除了跟踪之外，另一种获得PyTorch程序的TorchScript表示的方法是直接在 TorchScript 中编写代码_。 TorchScript在很大程度上是Python语言的一个子集，但有一些限制使得TorchScript编译器更容易推理程序。通过使用 `@torch.jit.script` 为自由函数和 `@torch.jit.script_method` 为类中的方法(必须也从 `torch.jit.ScriptModule` 派生）注释，将常规PyTorch代码转换为TorchScript。有关TorchScript注释的更多详细信息，请参见此处

使用TorchScript而不是跟踪的一个特殊原因是跟踪无法捕获PyTorch代码中的控制流。因此，让我们考虑一下这个使用控制流程的功能：

```
def compute(x, y):
  if bool(x[0][0] == 42):
      z = 5
  else:
      z = 10
  return x.matmul(y) + z
```

要将此函数从vanilla PyTorch转换为TorchScript，我们使用 `@torch.jit.script` 对其进行注释：

```
@torch.jit.script
def compute(x, y):
  if bool(x[0][0] == 42):
      z = 5
  else:
      z = 10
  return x.matmul(y) + z
```

这将及时将 `compute` 函数编译成图表示，我们可以在 `compute.graph` 属性中检查它：

```
>>> compute.graph
graph(%x : Dynamic
 %y : Dynamic) {
 %14 : int = prim::Constant[value=1]()
 %2 : int = prim::Constant[value=0]()
 %7 : int = prim::Constant[value=42]()
 %z.1 : int = prim::Constant[value=5]()
 %z.2 : int = prim::Constant[value=10]()
 %4 : Dynamic = aten::select(%x, %2, %2)
 %6 : Dynamic = aten::select(%4, %2, %2)
 %8 : Dynamic = aten::eq(%6, %7)
 %9 : bool = prim::TensorToBool(%8)
 %z : int = prim::If(%9)
 block0() {
 -> (%z.1)
 }
 block1() {
 -> (%z.2)
 }
 %13 : Dynamic = aten::matmul(%x, %y)
 %15 : Dynamic = aten::add(%13, %z, %14)
 return (%15);
}
```

现在，就像以前一样，我们可以使用我们的自定义运算符，就像我们脚本代码中的任何其他函数一样：

```
torch.ops.load_library("libwarp_perspective.so")

@torch.jit.script
def compute(x, y):
  if bool(x[0] == 42):
     z = 5
  else:
     z = 10
  x = torch.ops.my_ops.warp_perspective(x, torch.eye(3))
  return x.matmul(y) + z
```

当TorchScript编译器看到对 `torch.ops.my_ops.warp_perspective` 的引用时，它将找到我们通过C++ 中的 `RegisterOperators` 对象注册的实现，并将其编译为其图表示：

```
>>> compute.graph
graph(%x.1 : Dynamic
 %y : Dynamic) {
 %20 : int = prim::Constant[value=1]()
 %16 : int[] = prim::Constant[value=[0, -1]]()
 %14 : int = prim::Constant[value=6]()
 %2 : int = prim::Constant[value=0]()
 %7 : int = prim::Constant[value=42]()
 %z.1 : int = prim::Constant[value=5]()
 %z.2 : int = prim::Constant[value=10]()
 %13 : int = prim::Constant[value=3]()
 %4 : Dynamic = aten::select(%x.1, %2, %2)
 %6 : Dynamic = aten::select(%4, %2, %2)
 %8 : Dynamic = aten::eq(%6, %7)
 %9 : bool = prim::TensorToBool(%8)
 %z : int = prim::If(%9)
 block0() {
 -> (%z.1)
 }
 block1() {
 -> (%z.2)
 }
 %17 : Dynamic = aten::eye(%13, %14, %2, %16)
 %x : Dynamic = my_ops::warp_perspective(%x.1, %17)
 %19 : Dynamic = aten::matmul(%x, %y)
 %21 : Dynamic = aten::add(%19, %z, %20)
 return (%21);
 }
```

请特别注意图表末尾对 `my_ops::warp_perspective` 的引用。

注意 TorchScript图表表示仍有可能发生变化。不要依赖它看起来像这样。

当在Python中使用我们的自定义运算符时，这就是它。简而言之，您使用 `torch.ops.load_library` 导入包含操作符的库，并像跟踪或脚本化的TorchScript代码一样调用您的自定义操作，就像任何其他 `torch` 操作符一样。

在C ++中使用TorchScript自定义运算符

TorchScript的一个有用功能是能够将模型序列化为磁盘文件。该文件可以通过线路发送，存储在文件系统中，更重要的是，可以动态反序列化和执行，而无需保留原始源代码。这可以在Python中实现，也可以在C ++中实现。为此，PyTorch为提供了一个纯C ++ API ，用于反序列化以及执行TorchScript模型。如果您还没有，请阅读关于在C ++中加载和运行序列化TorchScript模型的教程，接下来的几段将构建。

简而言之，即使从文件反序列化并在C ++中运行，自定义运算符也可以像常规 `torch` 运算符一样执行。对此的唯一要求是将我们之前构建的自定义操作符共享库与我们执行模型的C ++应用程序链接起来。在Python中，这只是调用 `torch.ops.load_library` 。在C ++中，您需要在您使用的任何构建系统中将共享库与主应用程序链接。以下示例将使用CMake展示此内容。

注意

从技术上讲，您也可以在运行时将共享库动态加载到C ++应用程序中，就像在Python中一样。在Linux上，你可以用dlopen 来做到这一点。在其他平台上存在等价物。

在上面链接的C ++执行教程的基础上，让我们从一个文件中的最小C ++应用程序开始， `main.cpp` 在我们的自定义操作符的不同文件夹中，加载并执行序列化的TorchScript模型：

```
#include <torch/script.h> // One-stop header.

#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;
  }
//使用torch::jit::load()从文件反序列化ScriptModule。
  std::shared_ptr<torch::jit::script::Module> module = torch::jit::load(argv[1]);
  std::vector<torch::jit::IValue> inputs;
  inputs.push_back(torch::randn({4, 8}));
  inputs.push_back(torch::randn({8, 5}));

  torch::Tensor output = module->forward(std::move(inputs)).toTensor();

  std::cout << output << std::endl;
}
```

连同一个小的 `CMakeLists.txt` 文件：

```
cmake_minimum_required(VERSION 3.1 FATAL_ERROR)
project(example_app)

find_package(Torch REQUIRED)

add_executable(example_app main.cpp)
target_link_libraries(example_app "${TORCH_LIBRARIES}")
target_compile_features(example_app PRIVATE cxx_range_for)
```

此时，我们应该能够构建应用程序：

```
$ mkdir build
$ cd build
$ cmake -DCMAKE_PREFIX_PATH=/path/to/libtorch ..
-- The C compiler identification is GNU 5.4.0
-- The CXX compiler identification is GNU 5.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Looking for pthread.h
-- Looking for pthread.h - found
-- Looking for pthread_create
-- Looking for pthread_create - not found
-- Looking for pthread_create in pthreads
-- Looking for pthread_create in pthreads - not found
-- Looking for pthread_create in pthread
-- Looking for pthread_create in pthread - found
-- Found Threads: TRUE
-- Found torch: /libtorch/lib/libtorch.so
-- Configuring done
-- Generating done
-- Build files have been written to: /example_app/build
$ make -j
Scanning dependencies of target example_app
[ 50%] Building CXX object CMakeFiles/example_app.dir/main.cpp.o
[100%] Linking CXX executable example_app
[100%] Built target example_app
```

并且在没有通过模型的情况下运行它：

```
$ ./example_app
usage: example_app <path-to-exported-script-module>
```

接下来，让我们序列化我们之前编写的使用自定义运算符的脚本函数：

```
torch.ops.load_library("libwarp_perspective.so")

@torch.jit.script
def compute(x, y):
  if bool(x[0][0] == 42):
      z = 5
  else:
      z = 10
  x = torch.ops.my_ops.warp_perspective(x, torch.eye(3))
  return x.matmul(y) + z

compute.save("example.pt")
```

最后一行将脚本函数序列化为名为“example.pt”的文件。如果我们将这个序列化模型传递给我们的C ++应用程序，我们可以立即运行它：

```
$ ./example_app example.pt
terminate called after throwing an instance of 'torch::jit::script::ErrorReport'
what():
Schema not found for node. File a bug report.
Node: %16 : Dynamic = my_ops::warp_perspective(%0, %19)
```

或者可能不是。也许还没有。当然！我们尚未将自定义运算符库与我们的应用程序相关联。我们现在就这样做，为了正确地执行此操作，让我们稍微更新一下我们的文件组织，如下所示：

```
example_app/
  CMakeLists.txt
  main.cpp
  warp_perspective/
    CMakeLists.txt
    op.cpp
```

这将允许我们将 `warp_perspective` 库CMake目标添加为我们的应用程序目标的子目录。 `example_app` 文件夹中的最上层 `CMakeLists.txt` 应如下所示：

```
cmake_minimum_required(VERSION 3.1 FATAL_ERROR)
project(example_app)

find_package(Torch REQUIRED)

add_subdirectory(warp_perspective)

add_executable(example_app main.cpp)
target_link_libraries(example_app "${TORCH_LIBRARIES}")
target_link_libraries(example_app -Wl,--no-as-needed warp_perspective)
target_compile_features(example_app PRIVATE cxx_range_for)
```

这个基本的CMake配置看起来很像以前，除了我们将 `warp_perspective` CMake构建添加为子目录。一旦其CMake代码运行，我们将 `example_app` 应用程序与 `warp_perspective` 共享库链接。

注意

上面的例子中嵌入了一个关键细节：`warp_perspective` 链接线的 `-Wl,--no-as-needed` 前缀。这是必需的，因为我们实际上不会在应用程序代码中从 `warp_perspective` 共享库中调用任何函数。我们只需要运行全局 `RegisterOperators` 对象的构造函数。不方便的是，这会使链接器混乱并使其认为它可以完全跳过与库的链接。在Linux上，`-Wl,--no-as-needed` 标志强制链接发生(注意：此标志特定于Linux！）。还有其他解决方法。最简单的是在运算符库中定义*某些函数*，您需要从主应用程序调用它。这可以像在某个头中声明的函数 `void init();` 一样简单，然后在运算符库中将其定义为 `void init() { }`。在主应用程序中调用此 `init()` 函数将使链接器感觉这是一个值得链接的库。不幸的是，这超出了我们的控制范围，我们宁愿让您知道原因和简单的解决方法，而不是将一些不透明的宏交给您的代码中的plop。

现在，由于我们现在在最上层找到 `Torch` 包，`warp_perspective` 子目录中的 `CMakeLists.txt` 文件可以缩短一点。它应该如下所示：

```
find_package(OpenCV REQUIRED)
add_library(warp_perspective SHARED op.cpp)
target_compile_features(warp_perspective PRIVATE cxx_range_for)
target_link_libraries(warp_perspective PRIVATE "${TORCH_LIBRARIES}")
target_link_libraries(warp_perspective PRIVATE opencv_core opencv_photo)
```

让我们重新构建我们的示例应用程序，它还将与自定义运算符库链接。在最上层 `example_app` 目录中：

```
$ mkdir build
$ cd build
$ cmake -DCMAKE_PREFIX_PATH=/path/to/libtorch ..
-- The C compiler identification is GNU 5.4.0
-- The CXX compiler identification is GNU 5.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Looking for pthread.h
-- Looking for pthread.h - found
-- Looking for pthread_create
-- Looking for pthread_create - not found
-- Looking for pthread_create in pthreads
-- Looking for pthread_create in pthreads - not found
-- Looking for pthread_create in pthread
-- Looking for pthread_create in pthread - found
-- Found Threads: TRUE
-- Found torch: /libtorch/lib/libtorch.so
-- Configuring done
-- Generating done
-- Build files have been written to: /warp_perspective/example_app/build
$ make -j
Scanning dependencies of target warp_perspective
[ 25%] Building CXX object warp_perspective/CMakeFiles/warp_perspective.dir/op.cpp.o
[ 50%] Linking CXX shared library libwarp_perspective.so
[ 50%] Built target warp_perspective
Scanning dependencies of target example_app
[ 75%] Building CXX object CMakeFiles/example_app.dir/main.cpp.o
[100%] Linking CXX executable example_app
[100%] Built target example_app
```

如果我们现在运行 `example_app` 二进制文件并将其交给我们的序列化模型，我们应该得到一个圆满的结局：

```
$ ./example_app example.pt
11.4125   5.8262   9.5345   8.6111  12.3997
 7.4683  13.5969   9.0850  11.0698   9.4008
 7.4597  15.0926  12.5727   8.9319   9.0666
 9.4834  11.1747   9.0162  10.9521   8.6269
10.0000  10.0000  10.0000  10.0000  10.0000
10.0000  10.0000  10.0000  10.0000  10.0000
10.0000  10.0000  10.0000  10.0000  10.0000
10.0000  10.0000  10.0000  10.0000  10.0000
[ Variable[CPUFloatType]{8,5} ]
```

成功！你现在准备推断了。

结论

本教程向您介绍了如何在C ++中实现自定义TorchScript运算符，如何将其构建到共享库中，如何在Python中使用它来定义TorchScript模型，以及最后如何将其加载到C ++应用程序中以进行推理工作负载。现在，您已准备好使用与第三方C ++库连接的C ++运算符扩展TorchScript模型，编写自定义高性能CUDA内核，或实现需要Python，TorchScript和C ++之间的界线平滑混合的任何其他用例。

与往常一样，如果您遇到任何问题或有疑问，您可以使用我们的论坛或 GitHub问题取得联系。此外，我们的常见问题解答(FAQ）页面可能会提供有用的信息。

附录A：构建自定义运算符的更多方法

“构建自定义运算符”部分介绍了如何使用CMake将自定义运算符构建到共享库中。本附录概述了另外两种编译方法。它们都使用Python作为编译过程的“驱动程序”或“接口”。此外，两者都重复使用，这是相当于TorchScript自定义运算符的香草(渴望）PyTorch，它依赖于 pybind11 来实现从C ++到Python的“显式”绑定。

第一种方法使用C ++扩展'方便的即时(JIT）编译接口，在第一次运行时在PyTorch脚本的后台编译代码。第二种方法依赖于古老的 `setuptools` 包，并涉及编写单独的 `setup.py` 文件。这允许更高级的配置以及与其他基于 `setuptools` 的项目的集成。我们将在下面详细探讨这两种方法。

使用JIT编译构建

PyTorch C ++扩展工具包提供的JIT编译功能允许将自定义运算符的编译直接嵌入到Python代码中，例如：在训练脚本的顶部。

注意

这里的“JIT编译”与TorchScript编译器中的JIT编译无关，以优化您的程序。它只是意味着您的自定义操作符C ++代码将在您第一次导入时在系统的 `/tmp` 目录下的文件夹中编译，就像您事先已经自己编译它一样。

这个JIT编译功能有两种形式。首先，您仍然将运算符实现保存在单独的文件(`op.cpp` ）中，然后使用 `torch.utils.cpp_extension.load()` 编译扩展。通常，此函数将返回公开C ++扩展的Python模块。但是，由于我们没有将自定义运算符编译到自己的Python模块中，因此我们只想编译普通的共享库。幸运的是， `torch.utils.cpp_extension.load()` 有一个参数 `is_python_module` 我们可以设置为 `False` 来表示我们只对构建共享库而不是Python模块感兴趣。然后 `torch.utils.cpp_extension.load()` 将编译并将共享库加载到当前进程中，就像之前 `torch.ops.load_library` 所做的那样：

```
import torch.utils.cpp_extension

torch.utils.cpp_extension.load(
    name="warp_perspective",
    sources=["op.cpp"],
    extra_ldflags=["-lopencv_core", "-lopencv_imgproc"],
    is_python_module=False,
    verbose=True
)

print(torch.ops.my_ops.warp_perspective)
```

这应该大致打印：

```
<built-in method my_ops::warp_perspective of PyCapsule object at 0x7f3e0f840b10>
```

JIT编译的第二种风格允许您将自定义TorchScript运算符的源代码作为字符串传递。为此，请使用`torch.utils.cpp_extension.load_inline`：

```
import torch
import torch.utils.cpp_extension

op_source = """
#include <opencv2/opencv.hpp>
#include <torch/script.h>

torch::Tensor warp_perspective(torch::Tensor image, torch::Tensor warp) {
 cv::Mat image_mat(/*rows=*/image.size(0),
 /*cols=*/image.size(1),
 /*type=*/CV_32FC1,
 /*data=*/image.data<float>());
 cv::Mat warp_mat(/*rows=*/warp.size(0),
 /*cols=*/warp.size(1),
 /*type=*/CV_32FC1,
 /*data=*/warp.data<float>());

 cv::Mat output_mat;
 cv::warpPerspective(image_mat, output_mat, warp_mat, /*dsize=*/{64, 64});

 torch::Tensor output =
 torch::from_blob(output_mat.ptr<float>(), /*sizes=*/{64, 64});
 return output.clone();
}

static auto registry =
 torch::jit::RegisterOperators("my_ops::warp_perspective", &warp_perspective);
"""
torch.utils.cpp_extension.load_inline(
    name="warp_perspective",
    cpp_sources=op_source,
    extra_ldflags=["-lopencv_core", "-lopencv_imgproc"],
    is_python_module=False,
    verbose=True,
)

print(torch.ops.my_ops.warp_perspective)
```

当然，如果源代码相当短，最好只使用 `torch.utils.cpp_extension.load_inline` 。

使用Setuptools构建

从Python独家构建自定义运算符的第二种方法是使用 `setuptools` 。这样做的好处是 `setuptools` 具有非常强大和广泛的接口，用于构建用C ++编写的Python模块。但是，由于 `setuptools` 实际上是用于构建Python模块而不是普通的共享库(它没有Python期望从模块中获得的必要入口点），因此这条路线可能有点古怪。也就是说，你只需要一个 `setup.py` 文件代替 `CMakeLists.txt` ，如下所示：

请注意，我们在底部的 `BuildExtension` 中启用了 `no_python_abi_suffix` 选项。这指示 `setuptools` 在生成的共享库的名称中省略任何Python-3特定的ABI后缀。否则，在Python 3.7上，例如，库可能被称为 `warp_perspective.cpython-37m-x86_64-linux-`

`gnu.so`，其中`cpython-37m-x86_64-linux-gnu`是ABI标签，但我们真的只是想要它被称为`warp_perspective.so`

如果我们现在在`setup.py`所在的文件夹中的终端中运行`python setup.py build develop`，我们应该看到类似的内容：

```
$ python setup.py build develop
running build
running build_ext
building 'warp_perspective' extension
creating build
creating build/temp.linux-x86_64-3.7
gcc -pthread -B /root/local/miniconda/compiler_compat -Wl,--sysroot=/ -Wsign-compare -
DNDEBUG -g -fwrapv -O3 -Wall -Wstrict-prototypes -fPIC -
I/root/local/miniconda/lib/python3.7/site-packages/torch/lib/include -
I/root/local/miniconda/lib/python3.7/site-packages/torch/lib/include/torch/csrc/api/include -
I/root/local/miniconda/lib/python3.7/site-packages/torch/lib/include/TH -
I/root/local/miniconda/lib/python3.7/site-packages/torch/lib/include/THC -
I/root/local/miniconda/include/python3.7m -c op.cpp -o build/temp.linux-x86_64-3.7/op.o -
DTORCH_API_INCLUDE_EXTENSION_H -DTORCH_EXTENSION_NAME=warp_perspective -
D_GLIBCXX_USE_CXX11_ABI=0 -std=c++11
cc1plus: warning: command line option '-Wstrict-prototypes' is valid for C/ObjC but not for
C++
creating build/lib.linux-x86_64-3.7
g++ -pthread -shared -B /root/local/miniconda/compiler_compat -L/root/local/miniconda/lib -
Wl,-rpath=/root/local/miniconda/lib -Wl,--no-as-needed -Wl,--sysroot=/ build/temp.linux-
x86_64-3.7/op.o -lopencv_core -lopencv_imgproc -o build/lib.linux-x86_64-
3.7/warp_perspective.so
running develop
running egg_info
creating warp_perspective.egg-info
writing warp_perspective.egg-info/PKG-INFO
writing dependency_links to warp_perspective.egg-info/dependency_links.txt
writing top-level names to warp_perspective.egg-info/top_level.txt
writing manifest file 'warp_perspective.egg-info/SOURCES.txt'
reading manifest file 'warp_perspective.egg-info/SOURCES.txt'
writing manifest file 'warp_perspective.egg-info/SOURCES.txt'
running build_ext
copying build/lib.linux-x86_64-3.7/warp_perspective.so ->
Creating /root/local/miniconda/lib/python3.7/site-packages/warp-perspective.egg-link (link to .)
Adding warp-perspective 0.0.0 to easy-install.pth file

Installed /warp_perspective
Processing dependencies for warp-perspective==0.0.0
Finished processing dependencies for warp-perspective==0.0.0
```

这将生成一个名为`warp_perspective.so`的共享库，我们可以像之前那样将其传递给`torch.ops.load_library`，以使我们的操作符对TorchScript可见：

未找到相关的 Issues 进行评论

请联系 @jiangzhonglian @wizardforcel 初始化创建