第1章 PyTorch程序的基本结构

主要内容

PyTorch的发展历史

参考 https://mp.weixin.qq.com/s/JrutTVvFtx3xZoagy661LQ

- 相关的人： Soumith Chintala

Torch7

PyTorch的启动

加入Apache基金会

###

下面是一个非常简单的PyTorch训练代码

```
import os
import time

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.autograd import Variable

from torch.utils.data import DataLoader
from torchvision import datasets, transforms

from collections import OrderedDict
import torch.utils.model_zoo as model_zoo
from torchvision import models

def get_dataset(batch_size, data_root='/tmp/public_dataset/pytorch', train=True, val=True,
    data_root = os.path.expanduser(os.path.join(data_root, 'mnist-data'))

    ds = []
    if train:
        train_loader = torch.utils.data.DataLoader(
            datasets.MNIST(root=data_root, train=True, download=True,
                           transform=transforms.Compose([
                                transforms.Resize((224, 224)),
                                transforms.Grayscale(3),
                                transforms.ToTensor(),
                                transforms.Normalize((0.1307,), (0.3081,))
                            ])),
            batch_size=batch_size, shuffle=True, **kwargs)
        ds.append(train_loader)
    if val:
        test_loader = torch.utils.data.DataLoader(
            datasets.MNIST(root=data_root, train=False, download=True,
                           transform=transforms.Compose([
                                transforms.Resize((224, 224)),
                                transforms.Grayscale(3),
                                transforms.ToTensor(),
                                transforms.Normalize((0.1307,), (0.3081,))
                            ])),
            batch_size=batch_size, shuffle=True, **kwargs)
        ds.append(test_loader)
    ds = ds[0] if len(ds) == 1 else ds
    return ds


epochs = 10
test_interval = 1
data_root = 'data'

use_cuda = torch.cuda.is_available()

# data loader
train_loader, test_loader = get_dataset(batch_size=200, data_root='./data', num_workers=1)

# model
model = models.resnet18(pretrained=True)
in_features = model.fc.in_features
model.fc = nn.Linear(in_features, 10)
if use_cuda:
    model.cuda()

# optimizer
optimizer = optim.SGD(model.parameters(), lr=0.01, weight_decay=0.0001, momentum=0.9)

t_begin = time.time()

for epoch in range(epochs):
    model.train()
    total = 0
    for batch_idx, (data, target) in enumerate(train_loader):
        indx_target = target.clone()
        if use_cuda:
            data, target = data.cuda(), target.cuda()

        optimizer.zero_grad()
        output = model(data)
        loss = F.cross_entropy(output, target)
        loss.backward()
        optimizer.step()

        total += len(data)
        elapse_time = time.time() - t_begin
        t_begin = elapse_time
        print("samples {}, time {}s".format(total, int(elapse_time)))

    if epoch % test_interval == 0:
        model.eval()
        test_loss = 0
        correct = 0
        for data, target in test_loader:
            indx_target = target.clone()
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            output = model(data)
            test_loss += F.cross_entropy(output, target).data
            pred = output.data.max(1)[1]  # get the index of the max log-probability
            correct += pred.cpu().eq(indx_target).sum()

        test_loss = test_loss / len(test_loader) # average over number of mini-batch
        acc = 100. * correct / len(test_loader.dataset)
        print('Test set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)'.format(
            test_loss, correct, len(test_loader.dataset), acc))
```

从这段代码可以看到，一般模型训练的代码包括几个部分： * 数据集的处理和加载 * 神经网络结构的构建、初始化 * 优化器的配置 * 损失函数的选择，见line 79，这里用的是交叉熵 * 迭代训练并定期在验证集上测试验证其准确率 * 保存合适的模型文件，这里没有做这一步

PyTorch的源代码结构

PyTorch的整体架构

PyTorch的源代码结构

```
pytorch
|--- android        # PyTorch for Android
|--- aten           #   C++ Tensor
|--- benchamarks    #  PyTorch Benchmarking
|--- binaries       #
|--- c10            #  Tensor
|--- caffe2         #  Caffe2
|--- cmake          # PyTorch
|--- docs           # PyTorch     Python C++
|--- ios            # PyTorch for iOS
|--- modules        #
|--- mypy_plugins   #
|--- scripts        #
|--- submodules     #
|--- test           #
|--- third_party    #
|--- tools          #
|--- torch          # PyTorch Python
|--- torchgen       #

torch
|--- csrc       # Torch C++
      |--- module.cpp       # Torch C++
```

C10

C10，来自于Caffe Tensor Library的缩写。这里存放的都是最基础的Tensor库的代码，可以运行在服务端和移动端。

C10目前最具代表性的一个class就是TensorImpl了，它实现了Tensor的最基础框架。继承者和使用者有：

```
Variable Variable::Impl
SparseTensorImpl
detail::make_tensor<TensorImpl>(storage_impl, CUDATensorId(), false)
Tensor(c10::intrusive_ptr<TensorImpl, UndefinedTensorImpl> tensor_impl)
c10::make_intrusive<at::TensorImpl, at::UndefinedTensorImpl>
```

值得一提的是，C10中还使用/修改了来自llvm的SmallVector，在vector元素比较少的时候用以代替std::vector，

ATen

ATen，来自于 A TENsor library for C++11的缩写；PyTorch的C++ tensor li-
brary。ATen部分有大量的代码是来声明和定义Tensor运算相关的逻辑的，除此之外，PyTorch还使用了aten/src/AT

Caffe2

为了复用，2018年4月Facebook宣布将Caffe2的仓库合并到了PyTorch的仓库,从用户层面来复用包含了代码、CI、部
37m-x86_64-linux-gnu.so（caffe2 CPU Python 绑定）、caffe2_pybind11_state_gpu.cpython-
37m-x86_64-linux-gnu.so（caffe2 CUDA Python 绑定），基本上来自旧的caffe2项目)

Torch

Torch，部分代码仍然在使用以前的快要进入历史博物馆的Torch开源项目，比如具有下面这些文件名格式的文件：

```
TH* = TorcH
THC* = TorcH Cuda
THCS* = TorcH Cuda Sparse (now defunct)
THCUNN* = TorcH CUda Neural Network (see cunn)
THD* = TorcH Distributed
THNN* = TorcH Neural Network
THS* = TorcH Sparse (now defunct)
THP* = TorcH Python
```

PyTorch会使用tools/setup_helpers/generate_code.py来动态生成Torch层面相关的一些代码，这部分动态生成的逻

参考

- PyTorch ATen代码的动态生成 https://zhuanlan.zhihu.com/p/55966063
- Pytorch1.3源码解析-第一篇 https://www.cnblogs.com/jeshy/p/11751253.html

第四章 PyTorch的编译

主要内容

- PyTorch的编译过程
- setup.py的结构
- 代码生成过程
- 生成的二进制包

环境准备

大多数情况下我们只需要安装PyTorch的二进制版本即可，即可进行普通的模型开发训练了，但如果要深入了解PyTo

根据官方文档，建议安葬Python 3.7或以上的环境，而且需要C++14的编译器，比如clang，一开始我在ubuntu中装了

Python的环境我也根据建议安装了Anaconda，一方面Anaconda会自动安装很多库，包括PyTorch所依赖的mkl这样的加

如果我们需要编译支持GPU的PyTorch，需要安装cuda、cudnn，其中cuda建议安装10.2以上，cuDNN建议v7以上版本。

另外，为了不影响本机环境，建议基于容器环境进行编译。

本机环境准备

笔者的开发环境是在一台比较老的PC机上，主机操作系统是Ubuntu18.04，配置了GPU卡GTX1660Ti。如果读者记不清

```
lspci |grep VGA
01:00.0 VGA compatible controller: NVIDIA Corporation Device 2182 (rev a1)
```

如果输出中没有GPU型号，如上面的输出，可以在以下网站查询得到： http://pci-ids.ucw.cz/read/PC/10de/2182

在确定GPU卡型号之后，可以在NVIDIA的网站上查找对应的驱动，网址为：
https://www.nvidia.com/Download/index.aspx?lang=en-us。 比如笔者的1660Ti的驱动信息如下：
> > Linux x64 (AMD64/EM64T) Display Driver >
> Version: 515.76 > Release Date: 2022.9.20 > Operating System: Linux 64-bit > Language: English (US) > File Size: 347.96 MB >

下载对应的驱动之后，安装即可。一般的电脑都有核心网卡，在安装的过程中可以考虑将核心显卡用于显示，独立显

如果是在主机环境编译，需要安装CUDA和Cudnn，根据NVIDIA官网的提示进行安装即可。

如果使用容器环境进行编译，本机还需要安装nvidia-container-runtime。

```
curl -s -L https://nvidia.github.io/nvidia-docker/gpgkey | sudo apt-key add -
distribution=$(. /etc/os-release;echo $ID$VERSION_ID)
echo $distribution
curl -s -L https://nvidia.github.io/nvidia-docker/$distribution/nvidia-docker.list | sudo te
#wget https://nvidia.github.io/nvidia-container-runtime/ubuntu14.04/amd64/./nvidia-containe
sudo apt-get -y update
sudo apt-get install -y nvidia-container-toolkit
sudo apt-get install -y nvidia-container-runtime
sudo systemctl restart docker
```

之后需要安装docker，并将当前用户加入到docker的用户组里。

```
$ apt install docker.io
$ groupadd docker
$ usermod -ag docker <user>
```

在主机环境准备好后，我们开始准备基于ubuntu18.04的开放编译环境。

为了简便起见，建议直接使用NVIDIA预先准备好的容器环境，从这里可以找到对应本机操作系统和CUDA版本的容器：https://hub.docker.com/r/nvidia/cuda。

比如笔者所使用的环境是Ubuntu18.04+CUDA11.7，因此应该使用的容器环境是：nvidia/cuda:11.7.0-cudnn8-devel-ubuntu18.04

启动容器的命令如下，读者朋友也可以根据需要加上其他的参数。笔者已经克隆了PyTorch的源码，放在${HOME}/wo

```
docker run -it --rm -v ${HOME}/workspace/lab:/lab --gpus all nvidia/cuda:11.7.0-cudnn8-devel
```

另外，笔者编译PyTorch的时候，选择的是1.12.1的Tag，在编译的时候，要求cmake的版本高于3.13.0，而该容器自

从官网上下载cmake源代码，https://cmake.org/download/。解压后运行如下命令即可安装：

```
$ apt remove cmake
$ apt install libssl-dev
$ cd cmake-3.24.2
$ ./configure
$ make
$ make install
```

根据PyTorch README中的说明，需要在conda中安装多个依赖包：

```
$ conda install astunparse numpy ninja pyyaml setuptools cmake cffi typing_extensions future
$ conda install mkl mkl-include
```

编译步骤

```
$ git clone --recursive https://github.com/pytorch/pytorch
$ cd pytorch
# if you are updating an existing checkout
$ git submodule sync
$ git submodule update --init --recursive --jobs 0
$ git submodule update --init --recursive
```

启动容器，挂载PyTorch源码所在的目录，然后启动编译命令：

```
#    DEBUG      DEBUG=1 tools/setup_helpers/env.py           '-O0 -g'
#  toos/setup_helpers/cmake.py    make      MAX_JOBS                PC          CPU
python setup.py clean
MAX_JOBS=2 DEBUG=1 USE_GPU=1 python setup.py build 2>&1 | tee build_test.log
```

在编译启动后，会创建build目录，之后所有的编译工作都在这个目录下完成。

如果没有什么问题，编译的最后输出如下：

```

-- Build files have been written to: /lab/tmp/pytorch/build

[1191/6244] Generating src/x86_64-fma/softmax.py.o
[1208/6244] Building C object confu-deps/XNNPACK/CMakeFiles/all_microkernels.dir/src/f32-dwc
```

```
[1209/6244] Generating src/x86_64-fma/blas/sdotxf.py.o

......

[  0%] Linking C static library ../../../../lib/libclog.a
[  0%] Linking C static library ../../lib/libpthreadpool.a
[  1%] Linking CXX static library ../../../lib/libgtestd.a
[  2%] Linking C static library ../../../lib/libtensorpipe_uv.a
[  4%] Linking CXX static library ../../../lib/libprotobuf-lited.a
[  4%] Linking CXX static library ../../../lib/libbenchmark.a
[  4%] Linking CXX static library ../../../lib/libgloo.a
[  4%] Linking CXX static library ../../../lib/libasmjit.a
[  6%] Linking CXX static library ../../lib/libfmtd.a
[  7%] Linking CXX static library ../../../lib/libprotobufd.a
[  9%] Linking CXX shared library ../lib/libcaffe2_nvrtc.so
[  9%] Linking CXX shared library ../lib/libc10.so
[  9%] Linking C static library ../../lib/libfoxi_loader.a
[  9%] Linking C executable ../../bin/mkrename
[  9%] Linking C executable ../../bin/mkalias
[ 11%] Linking C executable ../../bin/mkdisp
[ 11%] Linking C shared library ../lib/libtorch_global_deps.so
[ 11%] Linking C executable ../../bin/mkrename_gnuabi
[ 11%] Linking C executable ../../bin/mkmasked_gnuabi
[ 11%] Linking C executable ../../bin/addSuffix
[ 13%] Linking C static library ../../lib/libcpuinfo.a
[ 15%] Linking C static library ../../lib/libcpuinfo_internals.a
[ 16%] Linking C static library ../../lib/libqnnpack.a
[ 16%] Linking C static library ../../lib/libnnpack_reference_layers.a
[ 18%] Linking CXX static library ../../lib/libpytorch_qnnpack.a
[ 23%] Linking CXX static library ../../../lib/libprotocd.a
[ 23%] Linking CXX static library ../../../lib/libbenchmark_main.a
[ 24%] Linking CXX static library ../../../lib/libgtest_maind.a
[ 24%] Linking CXX static library ../../../lib/libgmockd.a
[ 26%] Linking C static library ../../lib/libnnpack.a
[ 26%] Linking CXX static library ../../../../../../lib/libdnnl.a
[ 38%] Linking CXX static library ../../lib/libXNNPACK.a
[ 45%] Linking CXX static library ../../../lib/libtensorpipe.a
[ 50%] Linking CXX executable ../../bin/c10_intrusive_ptr_benchmark
[ 51%] Linking CXX shared library ../../lib/libc10_cuda.so
[ 54%] Linking CXX executable ../../../bin/protoc
[ 54%] Linking CXX static library ../../../lib/libkineto.a
[ 54%] Linking CXX static library ../../../../lib/libdnnl_graph.a
[ 54%] Linking CXX static library ../../../lib/libgmock_maind.a
[ 56%] Linking C static library ../../lib/libsleef.a
[ 57%] Linking CXX static library ../../../lib/libtensorpipe_cuda.a
[ 63%] Linking CXX static library ../../lib/libonnx_proto.a
[ 64%] Linking CXX static library ../lib/libcaffe2_protos.a
[ 70%] Linking CXX static library ../../lib/libonnx.a
[ 74%] Linking CXX static library ../../lib/libfbgemm.a
[ 74%] Linking CXX executable ../bin/vec_test_all_types_AVX2
[ 74%] Linking CXX executable ../bin/vec_test_all_types_DEFAULT
[ 88%] Linking CXX shared library ../lib/libtorch_cpu.so
Linking    libnccl.so.2.10.3                   > /lab/pytorch-build/pytorch/build/nccl/lib/l
[ 88%] Linking CXX static library ../../../lib/libgloo_cuda.a
[ 93%] Linking CXX shared library ../lib/libtorch_cuda.so
[ 93%] Linking CXX shared library ../lib/libtorch.so
[ 93%] Linking CXX shared library ../lib/libtorch_cuda_linalg.so
[ 93%] Linking CXX executable ../bin/example_allreduce
[ 93%] Linking CXX executable ../bin/basic
[ 93%] Linking CXX executable ../bin/atest
[ 94%] Linking CXX executable ../bin/test_parallel
[ 94%] Linking CXX executable ../bin/verify_api_visibility
[ 94%] Linking CXX executable ../bin/mobile_memory_cleanup
[ 94%] Linking CXX shared library ../lib/libbackend_with_compiler.so
[ 94%] Linking CXX executable ../bin/tutorial_tensorexpr
[ 94%] Linking CXX shared library ../../../../lib/libshm.so
[ 94%] Linking CXX executable ../bin/parallel_benchmark
[ 95%] Linking CXX executable ../../../../bin/torch_shm_manager
[ 98%] Linking CXX executable ../bin/nvfuser_bench
[100%] Linking CXX shared library ../../lib/libtorch_python.so
[100%] Linking CXX shared library ../../lib/libnnapi_backend.so
building 'torch._C' extension

building 'torch._C_flatbuffer' extension

building 'torch._dl' extension

-------------------------------------------------------------------------
|                                                                       |
|    It is no longer necessary to use the 'build' or 'rebuild' targets  |
|                                                                       |
|    To install:                                                        |
|      $ python setup.py install                                        |
|    To develop locally:                                                |
|      $ python setup.py develop                                        |
|    To force cmake to re-generate native build files (off by default): |
|      $ python setup.py develop --cmake                                |
|                                                                       |
-------------------------------------------------------------------------
```

PyTorch的setup.py

参考 https://blog.csdn.net/Sky_FULL1/article/details/125652654

PyTorch使用setuptools进行编译安装。

　　setuptools是常用的python库源码安装工具， 其最主要的函数是setup(…)，所有安装包需要的参数包括包名

下面我们看一下PyTorch的setup.py，为了节约篇幅，并且考虑到绝大多数同学会使用Linux环境进行编译，这里删掉

```
# Constant known variables used throughout this file
cwd = os.path.dirname(os.path.abspath(__file__))
lib_path = os.path.join(cwd, "torch", "lib")
third_party_path = os.path.join(cwd, "third_party")
caffe2_build_dir = os.path.join(cwd, "build")

def configure_extension_build():
    #YL
    cmake_cache_vars = defaultdict(lambda: False, cmake.get_cmake_cache_variables())

    #YL

    library_dirs.append(lib_path)
    main_compile_args = []
    main_libraries = ['torch_python']
    main_link_args = []
    main_sources = ["torch/csrc/stub.c"]

    if cmake_cache_vars['USE_CUDA']:
        library_dirs.append(
            os.path.dirname(cmake_cache_vars['CUDA_CUDA_LIB']))

    if build_type.is_debug():
        extra_compile_args += ['-O0', '-g']
        extra_link_args += ['-O0', '-g']


    ################################################################################
    # Declare extensions and package
    ################################################################################

    extensions = []
    packages = find_packages(exclude=('tools', 'tools.*'))
    C = Extension("torch._C",
                  libraries=main_libraries,
                  sources=main_sources,
                  language='c',
                  extra_compile_args=main_compile_args + extra_compile_args,
                  include_dirs=[],
                  library_dirs=library_dirs,
                  extra_link_args=extra_link_args + main_link_args + make_relative_rpath_arg
    C_flatbuffer = Extension("torch._C_flatbuffer",
                             libraries=main_libraries,
                             sources=["torch/csrc/stub_with_flatbuffer.c"],
                             language='c',
                             extra_compile_args=main_compile_args + extra_compile_args,
                             include_dirs=[],
                             library_dirs=library_dirs,
                             extra_link_args=extra_link_args + main_link_args + make_relativ
    extensions.append(C)
    extensions.append(C_flatbuffer)

    if not IS_WINDOWS:
        DL = Extension("torch._dl",
                       sources=["torch/csrc/dl.c"],
                       language='c')
        extensions.append(DL)

    # These extensions are built by cmake and copied manually in build_extensions()
    # inside the build_ext implementation
    if cmake_cache_vars['BUILD_CAFFE2']:
        extensions.append(
            Extension(
                name=str('caffe2.python.caffe2_pybind11_state'),
                sources=[]),
        )
        if cmake_cache_vars['USE_CUDA']:
            extensions.append(
                Extension(
                    name=str('caffe2.python.caffe2_pybind11_state_gpu'),
                    sources=[]),
            )
        if cmake_cache_vars['USE_ROCM']:
            extensions.append(
                Extension(
                    name=str('caffe2.python.caffe2_pybind11_state_hip'),
                    sources=[]),
            )

    cmdclass = {
        'bdist_wheel': wheel_concatenate,
        'build_ext': build_ext,
        'clean': clean,
        'install': install,
        'sdist': sdist,
    }

    entry_points = ...

    return extensions, cmdclass, packages, entry_points, extra_install_requires


if __name__ == '__main__':
    extensions, cmdclass, packages, entry_points, extra_install_requires = configure_extensi
    setup(
        ext_modules=extensions,
        cmdclass=cmdclass,
        packages=packages,
        entry_points=entry_points,
        install_requires=install_requires,
        package_data={
            #YL
        },
        #YL
    )
```

PyTorch使用的是自定义的编译方法，指定了wheel_concatenate和build_ext这两个函数，分别负责库文件和扩展模

在编译库文件时，setuptools默认会编译打包以下文件： - 由 py_modules 或 packages 指定的源文件 - 所有由 ext_modules 或 libraries 指定的 C 源码文件 - 由 scripts 指定的脚本文件 - 类似于 test/test*.py 的文件 - README.txt 或 README，setup.py，setup.cfg - 所有 package_data 或 data_files 指定的文件

从上面的代码中可以看到，最主要的两个Extension是torch._C

基于cmake的编译体系

参考https://blog.csdn.net/HaoBBNuanMM/article/details/115720457

在build_ext()函数中，调用了Caffe2的编译，并且是在pytorch目录下开始编译的。

首先，打开开关CMAKE_EXPORT_COMPILE_COMMANDS，这样可以将所有的编译命令输出到一个文件里，我们可以在编译

```
set(CMAKE_EXPORT_COMPILE_COMMANDS ON)
```

之后设置优先使用CMake中的pthread库，据说libstdc++封装pthread库后，如果以dlopen的方式使用会导致空指针
https://zhuanlan.zhihu.com/p/128519905

```
set(THREADS_PREFER_PTHREAD_FLAG ON)
```

之后是一些编译的配置，内容比较多，下面列出了一些主要的配置项。其中有很多配置项使用宏cmake_dependent_o
cmake cmake_dependent_option(USE_CUDNN "Use cuDNN" ON   "USE_CUDA"
OFF)c 代表当开启USE_CUDA的时候，也开启USE_CUDNN，否则关闭USE_CUDNN。

```
# ---[ Options.
# Note to developers: if you add an option below, make sure you also add it to
# cmake/Summary.cmake so that the summary prints out the option values.
include(CMakeDependentOption)
option(BUILD_BINARY "Build C++ binaries" OFF)
option(BUILD_PYTHON "Build Python binaries" ON)
option(BUILD_CAFFE2 "Master flag to build Caffe2" OFF)
cmake_dependent_option(
    BUILD_CAFFE2_OPS "Build Caffe2 operators" ON
    "BUILD_CAFFE2" OFF)
option(BUILD_SHARED_LIBS "Build libcaffe2.so" ON)
option(USE_CUDA "Use CUDA" ON)
cmake_dependent_option(
    USE_CUDNN "Use cuDNN" ON
    "USE_CUDA" OFF)
cmake_dependent_option(
    USE_NCCL "Use NCCL" ON
    "USE_CUDA OR USE_ROCM;UNIX;NOT APPLE" OFF)
option(USE_TENSORRT "Using Nvidia TensorRT library" OFF)

# Ensure that an MKLDNN build is the default for x86 CPUs
# but optional for AArch64 (dependent on -DUSE_MKLDNN).
cmake_dependent_option(
  USE_MKLDNN "Use MKLDNN. Only available on x86, x86_64, and AArch64." "${CPU_INTEL}"
  "CPU_INTEL OR CPU_AARCH64" OFF)

option(USE_DISTRIBUTED "Use distributed" ON)
cmake_dependent_option(
    USE_MPI "Use MPI for Caffe2. Only available if USE_DISTRIBUTED is on." ON
    "USE_DISTRIBUTED" OFF)
cmake_dependent_option(
    USE_GLOO "Use Gloo. Only available if USE_DISTRIBUTED is on." ON
    "USE_DISTRIBUTED" OFF)
```

PyTorch对ONNX的支持有两种方式，如果已有ONNX库，可以配置使用系统的自带的ONNX，否则重新编译生成。

```
if(NOT USE_SYSTEM_ONNX)
  set(ONNX_NAMESPACE "onnx_torch" CACHE STRING "A namespace for ONNX; needed to build with o
else()
  set(ONNX_NAMESPACE "onnx" CACHE STRING "A namespace for ONNX; needed to build with other f
endif()
```

接下来引用utils.cmake，这个文件里包括了很多工具函数，用于后边编译过程中的一些处理。

```
# ---[ Utils
include(cmake/public/utils.cmake)
```

之后是一些版本号的设置，不再赘述。

这里设置了cmake的modules查找路径，以及编译输出的路径

```
# ---[ CMake scripts + modules
list(APPEND CMAKE_MODULE_PATH ${PROJECT_SOURCE_DIR}/cmake/Modules)

# ---[ CMake build directories
set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/lib)
set(CMAKE_LIBRARY_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/lib)
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/bin)
```

在编译的过程中，产生了下面这些动态库：

```
[  2%] Linking C static library ../../../lib/libtensorpipe_uv.a
[  9%] Linking CXX shared library ../lib/libcaffe2_nvrtc.so
[  9%] Linking CXX shared library ../lib/libc10.so
[ 11%] Linking C shared library ../lib/libtorch_global_deps.so
[ 45%] Linking CXX static library ../../../lib/libtensorpipe.a
[ 51%] Linking CXX shared library ../../lib/libc10_cuda.so
[ 57%] Linking CXX static library ../../../lib/libtensorpipe_cuda.a
[ 88%] Linking CXX shared library ../lib/libtorch_cpu.so
Linking    libnccl.so.2.10.3                   > /lab/pytorch-build/pytorch/build/nccl/lib/
[ 93%] Linking CXX shared library ../lib/libtorch_cuda.so
[ 93%] Linking CXX shared library ../lib/libtorch.so
[ 93%] Linking CXX shared library ../lib/libc10d_cuda_test.so
[ 93%] Linking CXX shared library ../lib/libtorch_cuda_linalg.so
[ 93%] Linking CXX executable ../bin/NamedTensor_test
[ 94%] Linking CXX executable ../bin/scalar_tensor_test
[ 94%] Linking CXX executable ../bin/undefined_tensor_test
[ 94%] Linking CXX executable ../bin/lazy_tensor_test
[ 94%] Linking CXX executable ../bin/tensor_iterator_test
[ 94%] Linking CXX executable ../bin/cuda_packedtensoraccessor_test
[ 94%] Linking CXX shared library ../lib/libjitbackend_test.so
[ 94%] Linking CXX shared library ../lib/libtorchbind_test.so
[ 94%] Linking CXX shared library ../lib/libbackend_with_compiler.so
[ 94%] Linking CXX executable ../bin/tutorial_tensorexpr
[ 94%] Linking CXX shared library ../../../../lib/libshm.so
[ 98%] Linking CXX executable ../bin/test_tensorexpr
[100%] Linking CXX shared library ../../lib/libtorch_python.so
[100%] Linking CXX shared library ../../lib/libnnapi_backend.so
```

最后，在通过cmake将必要的库编译完成以后，再执行setup.py中的编译命令，生成PyTorch所依赖的扩展：

```
building 'torch._C' extension
creating build/temp.linux-x86_64-3.9
creating build/temp.linux-x86_64-3.9/torch
creating build/temp.linux-x86_64-3.9/torch/csrc
gcc -pthread -B /root/anaconda3/compiler_compat -Wno-unused-result -Wsign-compare -DNDEBUG -
gcc -pthread -B /root/anaconda3/compiler_compat -shared -Wl,-rpath,/root/anaconda3/lib -Wl,-
building 'torch._C_flatbuffer' extension
gcc -pthread -B /root/anaconda3/compiler_compat -Wno-unused-result -Wsign-compare -DNDEBUG -
gcc -pthread -B /root/anaconda3/compiler_compat -shared -Wl,-rpath,/root/anaconda3/lib -Wl,-
building 'torch._dl' extension
gcc -pthread -B /root/anaconda3/compiler_compat -Wno-unused-result -Wsign-compare -DNDEBUG -
gcc -pthread -B /root/anaconda3/compiler_compat -shared -Wl,-rpath,/root/anaconda3/lib -Wl,-
```

对比着，在安装pytorch后，我们可以看到torch目录下有如下的动态库：

```
./_dl.cpython-36m-x86_64-linux-gnu.so
./lib/libtorch_python.so
./lib/libcaffe2_observers.so
./lib/libcaffe2_nvrtc.so
./lib/libc10.so
./lib/libc10_cuda.so
./lib/libshm.so
./lib/libcaffe2_detectron_ops_gpu.so
./lib/libtorch.so
./lib/libcaffe2_module_test_dynamic.so
./_C.cpython-36m-x86_64-linux-gnu.so
```

Caffe2下有下列动态库： “ ‘Bash ./python/caffe2_pybind11_state.cpython-36m-x86_64-linux-gnu.so ./python/caffe2_pybind11_state_gpu.cpython-36m-x86_64-linux-gnu.so • • •

```
# ---[ Misc checks to cope with various compiler modes
include(cmake/MiscCheck.cmake)

# External projects
include(ExternalProject)

include(cmake/Dependencies.cmake)

# ---[ Allowlist file if allowlist is specified
include(cmake/Allowlist.cmake)


# Prefix path to Caffe2 headers.
# If a directory containing installed Caffe2 headers was inadvertently
# added to the list of include directories, prefixing
# PROJECT_SOURCE_DIR means this source tree always takes precedence.
include_directories(BEFORE ${PROJECT_SOURCE_DIR})
```

```
# Prefix path to generated Caffe2 headers.
# These need to take precedence over their empty counterparts located
# in PROJECT_SOURCE_DIR.
include_directories(BEFORE ${PROJECT_BINARY_DIR})

include_directories(BEFORE ${PROJECT_SOURCE_DIR}/aten/src/)
include_directories(BEFORE ${PROJECT_BINARY_DIR}/aten/src/)

# ---[ Main build
add_subdirectory(c10)
add_subdirectory(caffe2)


# ---[ Modules
# If master flag for buildling Caffe2 is disabled, we also disable the
# build for Caffe2 related operator modules.
if(BUILD_CAFFE2)
  add_subdirectory(modules)
endif()

# ---[ Binaries
# Binaries will be built after the Caffe2 main libraries and the modules
# are built. For the binaries, they will be linked to the Caffe2 main
# libraries, as well as all the modules that are built with Caffe2 (the ones
# built in the previous Modules section above).
if(BUILD_BINARY)
  add_subdirectory(binaries)
endif()

include(cmake/Summary.cmake)
caffe2_print_configuration_summary()

# ---[ Torch Deploy
if(USE_DEPLOY)
  add_subdirectory(torch/csrc/deploy)
endif()
```

PyTorch 动态代码生成

参考 https://zhuanlan.zhihu.com/p/59425970 参考 https://zhuanlan.zhihu.com/p/55966063

PyTorch代码主要包括三部分： - C10. C10是Caffe Tensor Library的缩写。PyTorch目前正在将代码从ATen/core目
- ATen，ATen是A TENsor library for C++11的缩写，是PyTorch的C++ tensor li-
brary。ATen部分有大量的代码是来声明和定义Tensor运算相关的逻辑的，除此之外，PyTorch还使用了aten/src/AT
- Torch，部分代码仍然在使用以前的快要进入历史博物馆的Torch开源项目，比如具有下面这些文件名格式的文件：

```
TH* = TorcH
THC* = TorcH Cuda
THCS* = TorcH Cuda Sparse (now defunct)
THCUNN* = TorcH CUda Neural Network (see cunn)
THD* = TorcH Distributed
THNN* = TorcH Neural Network
THS* = TorcH Sparse (now defunct)
THP* = TorcH Python
```

PyTorch会使用tools/setup_helpers/generate_code.py来动态生成Torch层面相关的一些代码，这部分动态生成的逻

C10目前最具代表性的一个class就是TensorImpl了，它实现了Tensor的最基础框架。继承者和使用者有：

编译第三方的库

```
#Facebook  cpuinfo  cpu
third_party/cpuinfo

#Facebook
# Pytorch caffe2 ncnn coreml
third_party/onnx

#FB (Facebook) + GEMM (General Matrix-Matrix Multiplication)
#Facebook          caffe2 x86    backend
third_party/fbgemm

#   benchmark
third_party/benchmark

#   protobuf
third_party/protobuf

#   UT
third_party/googletest

#Facebook
third_party/QNNPACK

#
third_party/gloo

#Intel   MKL-DNN
third_party/ideep
```

代码生成

ATen的native函数是PyTorch目前主推的operator机制，作为对比，老旧的TH/THC函数（使用cwrap定义）将逐渐被A
op需要修改这个yaml文件。

“ ‘

代码生成相关的工具在tools目录下：

```
autograd
    gen_annotated_fn_args.py
    gen_autograd_functions.py
    gen_autograd.py
    gen_inplace_or_view_type.py
    gen_python_functions.py
    gen_trace_type.py
    gen_variable_factories.py
    gen_variable_type.py
    templates
        ADInplaceOrViewType.cpp
        annotated_fn_args.py.in
        Functions.cpp
        Functions.h
        python_fft_functions.cpp
        python_functions.cpp
        python_functions.h
        python_linalg_functions.cpp
        python_nn_functions.cpp
        python_return_types.cpp
        python_sparse_functions.cpp
        python_special_functions.cpp
        python_torch_functions.cpp
        python_variable_methods.cpp
        TraceType.cpp
        variable_factories.h
        VariableType.cpp
        VariableType.h
code_analyzer
    gen_operators_yaml.py
    gen_oplist.py
    gen_op_registration_allowlist.py
generated_dirs.txt
jit
    gen_unboxing.py
    templates
        aten_schema_declarations.cpp
        external_functions_codegen_template.cpp
```

```
setup_helpers
    generate_code.py
    gen.py
    gen_unboxing.py
    gen_version_header.py
```

我们先看几个重要的文件：

- generated_dirs.txt： 这个文件里列举了编译过程中自动生成的代码所在的路径，当前版本中该文件的内容如

  ```
  torch/csrc/autograd/generated/      #
  torch/csrc/jit/generated/           # JIT
  build/aten/src/ATen                 # aten
  ```

- setup_helpers/generate_code.py: 这个文件中函数generate_code()是代码生成的入口。等下我们会沿着这个
-

代码生成的流程

代码生成沿着以下的流程进行：

调用tools/autograd/gen_autograd.py中的函数gen_autograd_python，这个函数输入参数
NATIVE_FUNCTIONS_PATH = “aten/src/ATen/native/native_functions.yaml”
TAGS_PATH = “aten/src/ATen/native/tags.yaml”

```
<li> native_functions_path: native functions
<li> derivatives.yaml:
<li> templates:
<li> deprecated.yaml:
</ol>
    gen_python_functions.gen()          ATen  Python   torch._C nn _fft _linalg _sparse _specia
<ol>
<li> native_functions.yaml tags.yaml    native
<li>
<li> deprecated.yaml
<li> FileManager.write_with_template()              python_variable_methods.cpp
   python_variable_methods
```

```
#define TORCH_ASSERT_ONLY_METHOD_OPERATORS
// ${generated_comment}

// ...
#include <stdexcept>

#ifndef AT_PER_OPERATOR_HEADERS
#include <ATen/Functions.h>
#else
$ops_headers
```

```
#endif

//...

// generated methods start here

${py_methods}

static PyObject * THPVariable_bool_scalar(PyObject* self, PyObject* args) {

//...

  {"tolist", THPVariable_tolist, METH_NOARGS, NULL},
  {"type", castPyCFunctionWithKeywords(THPVariable_type), METH_VARARGS | METH_KEYWORDS, NULL
  ${py_method_defs}
  {NULL}
};
```

其中的关键变量是py_methods，这个变量包含了很多函数的定义，其中每个函数是根据模板字符串生成的，如下是其

```
// tools/autograd/gen_python_functions.py

PY_VARIABLE_METHOD_VARARGS = CodeTemplate(
    r"""\
// ${name}
static PyObject * ${pycname}(PyObject* self_, PyObject* args, PyObject* kwargs)
{
  ${method_header}
  static PythonArgParser parser({
    ${signatures}
  }, /*traceable=*/${traceable});

  ParsedArgs<${max_args}> parsed_args;
  auto _r = parser.parse(${self_}, args, kwargs, parsed_args);
  ${check_has_torch_function}
  switch (_r.idx) {
    ${dispatch}
  }
  ${method_footer}
}

"""
)
```

根据这个模板生成的函数代码大概是下面这样：

```
//torch/csrc/autograd/generated/python_variable_methods.cpp
```

```
static PyObject * THPVariable_add(PyObject* self_, PyObject* args, PyObject* kwargs)
{
  HANDLE_TH_ERRORS
  const Tensor& self = THPVariable_Unpack(self_);
  static PythonArgParser parser({
    "add(Scalar alpha, Tensor other)|deprecated",
    "add(Tensor other, *, Scalar alpha=1)",
  }, /*traceable=*/true);

  ParsedArgs<2> parsed_args;
  auto _r = parser.parse(self_, args, kwargs, parsed_args);
  if(_r.has_torch_function()) {
    return handle_torch_function(_r, self_, args, kwargs, THPVariableClass, "torch.Tensor")
  }
  switch (_r.idx) {
    case 0: {
      // [deprecated] aten::add.Tensor(Tensor self, Tensor other, *, Scalar alpha=1) -> Tens

      auto dispatch_add = [](const at::Tensor & self, const at::Scalar & alpha, const at::Te
        pybind11::gil_scoped_release no_gil;
        return self.add(other, alpha);
      };
      return wrap(dispatch_add(self, _r.scalar(0), _r.tensor(1)));
    }
    case 1: {
      // aten::add.Tensor(Tensor self, Tensor other, *, Scalar alpha=1) -> Tensor

      auto dispatch_add = [](const at::Tensor & self, const at::Tensor & other, const at::Sc
        pybind11::gil_scoped_release no_gil;
        return self.add(other, alpha);
      };
      return wrap(dispatch_add(self, _r.tensor(0), _r.scalar(1)));
    }
  }
  Py_RETURN_NONE;
  END_HANDLE_TH_ERRORS
}
```

```
<li>
</ol>
```

生成的库

```
# /pytorch/build/lib.linux-x86_64-3.7/torch
./_C.cpython-37m-x86_64-linux-gnu.so
```

```
./lib/libtorch_python.so
./lib/libtorchbind_test.so
./lib/libtorch_cpu.so
./lib/libjitbackend_test.so
./lib/libc10.so
./lib/libshm.so
./lib/libtorch.so
./lib/libtorch_global_deps.so
./lib/libbackend_with_compiler.so
./_C_flatbuffer.cpython-37m-x86_64-linux-gnu.so
./_dl.cpython-37m-x86_64-linux-gnu.so
```

其中_C.cpython-37m-x86_64-linux-gnu.so是主要的入口点，后面的章节我们会从这个入口点分析PyTorch的初始化
found可忽略）。

```
# pytorch/build/lib.linux-x86_64-3.7/torch

$ ldd ./_C.cpython-37m-x86_64-linux-gnu.so
    linux-vdso.so.1 (0x00007fff18175000)
    libtorch_python.so => /home/harry/lab/tmp/pytorch/build/lib.linux-x86_64-3.7/torch/./lib
    libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007feff2b42000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007feff2751000)
    libshm.so => /home/harry/lab/tmp/pytorch/build/lib.linux-x86_64-3.7/torch/./lib/libshm.s
    libtorch.so => /home/harry/lab/tmp/pytorch/build/lib.linux-x86_64-3.7/torch/./lib/libtor
    libtorch_cpu.so => /home/harry/lab/tmp/pytorch/build/lib.linux-x86_64-3.7/torch/./lib/li
    libc10.so => /home/harry/lab/tmp/pytorch/build/lib.linux-x86_64-3.7/torch/./lib/libc10.s
    libstdc++.so.6 => /usr/lib/x86_64-linux-gnu/libstdc++.so.6 (0x00007fefddc7c000)
    libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007fefdd8de000)
    libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007fefdd6c6000)
    /lib64/ld-linux-x86-64.so.2 (0x00007feff4fcc000)
    librt.so.1 => /lib/x86_64-linux-gnu/librt.so.1 (0x00007fefdd4be000)
    libgomp.so.1 => /usr/lib/x86_64-linux-gnu/libgomp.so.1 (0x00007fefdd28f000)
    libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007fefdd08b000)
    libmkl_intel_lp64.so => not found
    libmkl_gnu_thread.so => not found
    libmkl_core.so => not found
```

常见问题

- submodule没有下载完整 一个简单的处理办法是删除third_party下的相关目录，然后手动git clone即可。相关的git url定义在.submodule以及.gi/config中
- 编译时出现RPATH相关的问题 处理办法是先运行clean命令，然后再编译

```
> python setup.py clean
> python setup.py build
```

- lib库找不到 错误详情：No rule to make target ’/usr/lib/x86_64-linux-gnu/libXXX.so “‘bash > find / -name “librt.so.*” > ln -s /lib/x86_64-linux-gnu/librt.so.1 /usr/lib/x86_64-linux-gnu/librt.so

- c++   bash > apt install g++ “‘ 注意，如果安装clang，也可以编译，但c++的版本如果比较低，比如6.0，就
命令编译开关没找到 的问题。

- 在PC上编译时Hang住

一般来说为了加快编译速度，编译大型项目时都会采用并行编译的方式，pytorch的编译也是，启动编译后，可以在

简单起见，在启动编译前，可以设置环境变量CMAKE_BUILD_PARALLEL_LEVEL来减少编译的并行度。

- 编译Debug版本时出现internal compiler error

如果只是在编译Debug版本时出现，可能是和优化编译选项有冲突，因为优化编译选项-
O1 -O2 -O3可能会重新排列代码导致代码对应出现问题，排查真正的问题非常困难，建议简单处理，对出现问题的编
g选项或者-O 选项。

PyTorch的编译由setup.py发起，但真正执行编译时，相关的命令写在build/build.ninja里，只要在这个文件里修改

参考

https://zhuanlan.zhihu.com/p/321449610

https://blog.51cto.com/SpaceVision/5072093

https://zhuanlan.zhihu.com/p/55204134

https://github.com/pytorch/pytorch#from-source

从零开始编译PyTorch软件包 https://zhuanlan.zhihu.com/p/347084475

Pytorch setup.py 详解 https://blog.csdn.net/Sky_FULL1/article/details/125652654

PyTorch 动态代码生成 https://zhuanlan.zhihu.com/p/55966063

PyTorch 动态代码生成 https://zhuanlan.zhihu.com/p/59425970

https://blog.csdn.net/HaoBBNuanMM/article/details/115720457

PyTorch引擎的主要模块及初始化

主要内容

本章对PyTorch的整体架构做了初步的分析，这部分也是理解PyTorch核心引擎工作机制的关键部分，在这里我们力图

PyTorch从上层到C++的底层包括哪些重要的模块

这些模块是如何初始化的

从设计上看，这些模块是如何配合的

PyTorch的核心模块

- PythonAPI
- C++部分Engine
- THP
- ATen
- JITwdq

```
src
!--- ATen       # Tensor   C++
|--- TH         # Tensor CPU
|--- THC        # Tensor CUDA
|--- THCUNN     #    CUDA
|--- THNN       #    CPU

torch
|--- csrc       # Torch C++
      |--- module.cpp       # Torch C++
```

PyTorch的C++扩展模块初始化

C++扩展模块_C可以说是PyTorch的核心，是PyTorch代码量最大最复杂的部分，下面我们来看看这个模块是如何加载

C++扩展模块的加载

在加载torch模块的时候，python会执行torch/init.py. 其中会加载_C模块，根据Python3的规范，如果某个模块是
.so，在linux环境下，对应的就是_C.cpython-37m-x86_64-linux-gnu.so。

加载这个动态库后，会调用其中的initModule()函数。 在这个函数中，进行了一系列的初始化工作

```
// torch/csrc/Module.cpp

PyObject* initModule() {

  // ...

  //   _C
  THPUtils_addPyMethodDefs(methods, TorchMethods);
  THPUtils_addPyMethodDefs(methods, DataLoaderMethods);
  THPUtils_addPyMethodDefs(methods, torch::autograd::python_functions());
  THPUtils_addPyMethodDefs(methods, torch::multiprocessing::python_functions());

  THPUtils_addPyMethodDefs(methods, THCPModule_methods());
  THPUtils_addPyMethodDefs(methods, torch::distributed::c10d::python_functions());

  THPUtils_addPyMethodDefs(methods, torch::distributed::rpc::python_functions());
  THPUtils_addPyMethodDefs(
      methods, torch::distributed::autograd::python_functions());
  THPUtils_addPyMethodDefs(methods, torch::distributed::rpc::testing::python_functions());

  //    _C
  static struct PyModuleDef torchmodule = {
     PyModuleDef_HEAD_INIT,
     "torch._C",
     nullptr,
     -1,
     methods.data()
  };
  ASSERT_TRUE(module = PyModule_Create(&torchmodule));
  ASSERT_TRUE(THPGenerator_init(module));
  ASSERT_TRUE(THPException_init(module));
  THPSize_init(module);
  THPDtype_init(module);
  THPDTypeInfo_init(module);
  THPLayout_init(module);
  THPMemoryFormat_init(module);
  THPQScheme_init(module);
  THPDevice_init(module);
  THPStream_init(module);

  //   Tensor
  ASSERT_TRUE(THPVariable_initModule(module));
  ASSERT_TRUE(THPFunction_initModule(module));
  ASSERT_TRUE(THPEngine_initModule(module));
  // NOTE: We need to be able to access OperatorExportTypes from ONNX for use in
  // the export side of JIT, so this ONNX init needs to appear before the JIT
  // init.
  torch::onnx::initONNXBindings(module);
  torch::jit::initJITBindings(module);
  torch::monitor::initMonitorBindings(module);
  torch::impl::dispatch::initDispatchBindings(module);
  torch::throughput_benchmark::initThroughputBenchmarkBindings(module);
  torch::autograd::initReturnTypes(module);
  torch::autograd::initNNFunctions(module);
  torch::autograd::initFFTFunctions(module);
  torch::autograd::initLinalgFunctions(module);
  torch::autograd::initSparseFunctions(module);
  torch::autograd::initSpecialFunctions(module);
  torch::autograd::init_legacy_variable(module);
```

```
  torch::python::init_bindings(module);
  torch::lazy::initLazyBindings(module);
#ifdef USE_CUDA
  torch::cuda::initModule(module);
#endif
  ASSERT_TRUE(THPStorage_init(module));

#ifdef USE_CUDA
  // This will only initialise base classes and attach them to library namespace
  // They won't be ready for real usage until importing cuda module, that will
  // complete the process (but it defines Python classes before calling back into
  // C, so these lines have to execute first)..
  THCPStream_init(module);
  THCPEvent_init(module);
  THCPGraph_init(module);
#endif

  auto set_module_attr = [&](const char* name, PyObject* v, bool incref = true) {
    // PyModule_AddObject steals reference
    if (incref) {
      Py_INCREF(v);
    }
    return PyModule_AddObject(module, name, v) == 0;
  };

  // ...

  ASSERT_TRUE(set_module_attr("has_openmp", at::hasOpenMP() ? Py_True : Py_False));
  ASSERT_TRUE(set_module_attr("has_mkl", at::hasMKL() ? Py_True : Py_False));
  ASSERT_TRUE(set_module_attr("has_lapack", at::hasLAPACK() ? Py_True : Py_False));

  // ...

  py::enum_<at::native::ConvBackend>(py_module, "_ConvBackend")
    .value("CudaDepthwise2d", at::native::ConvBackend::CudaDepthwise2d)
    .value("CudaDepthwise3d", at::native::ConvBackend::CudaDepthwise3d)
    .value("Cudnn", at::native::ConvBackend::Cudnn)
    .value("CudnnTranspose", at::native::ConvBackend::CudnnTranspose)
    .value("Empty", at::native::ConvBackend::Empty)
    .value("Miopen", at::native::ConvBackend::Miopen)
    .value("MiopenDepthwise", at::native::ConvBackend::MiopenDepthwise)
    .value("MiopenTranspose", at::native::ConvBackend::MiopenTranspose)
    .value("Mkldnn", at::native::ConvBackend::Mkldnn)
    .value("MkldnnEmpty", at::native::ConvBackend::MkldnnEmpty)
    .value("NnpackSpatial", at::native::ConvBackend::NnpackSpatial)
    .value("Overrideable", at::native::ConvBackend::Overrideable)
    .value("Slow2d", at::native::ConvBackend::Slow2d)
    .value("Slow3d", at::native::ConvBackend::Slow3d)
    .value("SlowDilated2d", at::native::ConvBackend::SlowDilated2d)
    .value("SlowDilated3d", at::native::ConvBackend::SlowDilated3d)
    .value("SlowTranspose2d", at::native::ConvBackend::SlowTranspose2d)
    .value("SlowTranspose3d", at::native::ConvBackend::SlowTranspose3d)
    .value("Winograd3x3Depthwise", at::native::ConvBackend::Winograd3x3Depthwise)
    .value("Xnnpack2d", at::native::ConvBackend::Xnnpack2d);

  py_module.def("_select_conv_backend", [](
        const at::Tensor& input, const at::Tensor& weight, const c10::optional<at::Tensor>&
        at::IntArrayRef stride_, at::IntArrayRef padding_, at::IntArrayRef dilation_,
        bool transposed_, at::IntArrayRef output_padding_, int64_t groups_) {
      return at::native::select_conv_backend(
          input, weight, bias_opt, stride_, padding_, dilation_, transposed_, output_padding
  });

  py::enum_<at::LinalgBackend>(py_module, "_LinalgBackend")
    .value("Default", at::LinalgBackend::Default)
    .value("Cusolver", at::LinalgBackend::Cusolver)
    .value("Magma", at::LinalgBackend::Magma);

  py_module.def("_set_linalg_preferred_backend", [](at::LinalgBackend b) {
    at::globalContext().setLinalgPreferredBackend(b);
  });
  py_module.def("_get_linalg_preferred_backend", []() {
    return at::globalContext().linalgPreferredBackend();
  });

  // ...

  return module;
  END_HANDLE_TH_ERRORS
}
```

Torch 函数库的初始化

在Python层面，模块torch提供了非常多的函数，比如torch.abs()，torch.randn()，torch.ones()等等，在初始化_C模块的时候，这些函数也被注册到_C模块中。

```
// torch/csrc/autograd/python_variable.cpp

bool THPVariable_initModule(PyObject *module)
{
  // ...
  PyModule_AddObject(module, "_TensorBase",   (PyObject *)&THPVariableType);
  torch::autograd::initTorchFunctions(module);
  // ...
  return true;
}
```

在下面的代码中，我们可以看到，相关的函数被收集到torch_functions中，同时这个函数列表也被注册到_C的_Var

```
// torch/csrc/autograd/python_torch_functions_manual.cpp

void initTorchFunctions(PyObject *module) {
  static std::vector<PyMethodDef> torch_functions;
  gatherTorchFunctions(torch_functions);
  THPVariableFunctions.tp_methods = torch_functions.data();

  //...
  if (PyModule_AddObject(module, "_VariableFunctionsClass",
                         reinterpret_cast<PyObject*>(&THPVariableFunctions)) < 0) {
    throw python_error();
  }
  // PyType_GenericNew returns a new reference
  THPVariableFunctionsModule = PyType_GenericNew(&THPVariableFunctions, Py_None, Py_None);
  // PyModule_AddObject steals a reference
  if (PyModule_AddObject(module, "_VariableFunctions", THPVariableFunctionsModule) < 0) {
    throw python_error();
  }
}
```

在torch模块的初始化过程中，_C模块的子模块_VariableFunctions中的所有属性都被注册到torch模块中，当然也包

```
# torch/__init__.py

for name in dir(_C._VariableFunctions):
    if name.startswith('__') or name in PRIVATE_OPS:
        continue
    obj = getattr(_C._VariableFunctions, name)
    obj.__module__ = 'torch'
    globals()[name] = obj
    if not name.startswith("_"):
        __all__.append(name)
```

下面我们看看具体有哪些函数被注册了。函数列表是通过gatherTorchFunctions()来收集的，这个函数又调用了gat
gatherTorchFunctions_1()，gatherTorchFunctions_2()这几个函数。

```
// torch/csrc/autograd/python_torch_functions_manual.cpp

void gatherTorchFunctions(std::vector<PyMethodDef> &torch_functions) {
  constexpr size_t num_functions = sizeof(torch_functions_manual) / sizeof(torch_functions_m
  torch_functions.assign(torch_functions_manual,
                              torch_functions_manual + num_functions);
  // NOTE: Must be synced with num_shards in tools/autograd/gen_python_functions.py
  gatherTorchFunctions_0(torch_functions);
  gatherTorchFunctions_1(torch_functions);
  gatherTorchFunctions_2(torch_functions);

  //...
```

为什么这样设计呢？大概有两个原因： - 函数的数量很多，而且在不断的增加，需要方便扩展
- 函数大多是算子，算子和平台相关，每个算子有多种实现，同样为了在不同平台迁移扩展，PyTorch设计了代码生

gatherTorchFunctions_N()这几个函数是通过模板生成的，完成编译后，可以在下面的文件中找到：

```
// torch/csrc/autograd/generated/python_torch_functions_0.cpp

static PyMethodDef torch_functions_shard[] = {
  {"_cast_Byte", castPyCFunctionWithKeywords(THPVariable__cast_Byte), METH_VARARGS | METH_K
  //...
  {"eye", castPyCFunctionWithKeywords(THPVariable_eye), METH_VARARGS | METH_KEYWORDS | METH_
  {"rand", castPyCFunctionWithKeywords(THPVariable_rand), METH_VARARGS | METH_KEYWORDS | ME
  //...
};

void gatherTorchFunctions_0(std::vector<PyMethodDef> &torch_functions) {
  constexpr size_t num_functions = sizeof(torch_functions_shard) / sizeof(torch_functions_s
  torch_functions.insert(
    torch_functions.end(),
    torch_functions_shard,
    torch_functions_shard + num_functions);
}
```

Tensor

在Pytorch的早期版本中，Tensor被定义在TH模块中的THTensor类中，后来TH模块被移除了，也就有了更直观的Tens

当前Tensor的定义在TensorBody.h中，

```
// torch/include/ATen/core/TensorBody.h

class TORCH_API Tensor: public TensorBase {
 public:
  Tensor(const Tensor &tensor) = default;
  Tensor(Tensor &&tensor) = default;

  using TensorBase::size;
  using TensorBase::stride;
  Tensor cpu() const {
    return to(options().device(DeviceType::CPU), /*non_blocking*/ false, /*copy*/ false);
  }

  // TODO: The Python version also accepts arguments
  Tensor cuda() const {
    return to(options().device(DeviceType::CUDA), /*non_blocking*/ false, /*copy*/ false);
  }

  void backward(const Tensor & gradient={}, ...) const {
    ...
  }
}
```

我们还可以看到，Tensor类本身的实现很少，大部分功能来自于其父类TensorBase。根据文档注释我们可以了解到，

```
// torch/include/ATen/core/TensorBase.h

class TORCH_API TensorBase {

  int64_t dim() const {
    return impl_->dim();
  }
  int64_t storage_offset() const {
    return impl_->storage_offset();
  }

  // ...

  bool requires_grad() const {
    return impl_->requires_grad();
  }
  bool is_leaf() const;
  TensorBase data() const;

  c10::intrusive_ptr<TensorImpl, UndefinedTensorImpl> impl_;
}
```

> https://blog.csdn.net/Chris_zhangrx/article/details/119086815
> c10::intrusive_ptr是PyTorch的内部智能指针实现，其工作方式如下：
> 首先完美转发所有的参数来构建 intrusive_ptr 用这些参数 new 一个新的 TTarget 类型的对象 用新的 TTarget 对象构造一个 intrusive_ptr 构造 intrusive_ptr 的同时对 refcount_ 和 weakcount_ 都加 1， 如果是默认构造，则两个引用计数都默认为 0，根据这个可以将通过 make_intrusive 构造的指针与堆栈上的会被自动析构的情况分开，用来确保内存是我们自己分配的。

以后有机会我们再研究一下intrusive_ptr的实现，在此之前，我们主要关注impl_这个成员变量，也就是TensorImp

```
// c10/core/TensorImpl.h

struct C10_API TensorImpl : public c10::intrusive_ptr_target {

TensorImpl(
      Storage&& storage,
      DispatchKeySet,
      const caffe2::TypeMeta data_type);

 public:
  TensorImpl(const TensorImpl&) = delete;
  TensorImpl& operator=(const TensorImpl&) = delete;
  TensorImpl(TensorImpl&&) = delete;
  TensorImpl& operator=(TensorImpl&&) = delete;

  DispatchKeySet key_set() const {
    return key_set_;
  }

  int64_t dim() const {
    //...
  }
  bool is_contiguous(
    //...
  }

  Storage storage_;

private:
  std::unique_ptr<c10::AutogradMetaInterface> autograd_meta_ = nullptr;

 protected:
  std::unique_ptr<c10::NamedTensorMetaInterface> named_tensor_meta_ = nullptr;

  c10::VariableVersion version_counter_;

  std::atomic<impl::PyInterpreter*> pyobj_interpreter_;
  PyObject* pyobj_;

  c10::impl::SizesAndStrides sizes_and_strides_;

  int64_t storage_offset_ = 0;
  int64_t numel_ = 1;
  caffe2::TypeMeta data_type_;
  c10::optional<c10::Device> device_opt_;

  bool is_contiguous_ : 1;

  bool storage_access_should_throw_ : 1;

  bool is_channels_last_ : 1;
  bool is_channels_last_contiguous_ : 1;
  bool is_channels_last_3d_ : 1;

  bool is_channels_last_3d_contiguous_ : 1;

  bool is_non_overlapping_and_dense_ : 1;

  bool is_wrapped_number_ : 1;

  bool allow_tensor_metadata_change_ : 1;

  bool reserved_ : 1;
  uint8_t sizes_strides_policy_ : 2;

  DispatchKeySet key_set_;

}
```

对于TensorImpl类来说，比较重要的成员变量有以下几个： - storage_。这个变量存储了真正的张量数据 - autograd_meta_。存储反向传播所需要的元信息，如梯度计算函数和梯度等。 - pyobj_。Tensor所对应的Python Object - data_type_。Tensor内的数据类型。 - device_opt_。存放Tensor的设备。 -

下面我们看一下Tensor的存储，因为Tensor的存储方式和算子的计算息息相关，对性能的影响也非常的关键。

```
// c10/core/Storage.h

struct C10_API Storage {
  //...

 protected:
  c10::intrusive_ptr<StorageImpl> storage_impl_;
}
```

和Tensor的定义类似，Storage也是使用StorageImpl类来隐藏其复杂的实现。因此我们主要关注StorageImpl的实现

```
// c10/core/StorageImpl.h

struct C10_API StorageImpl : public c10::intrusive_ptr_target {
 public:
  struct use_byte_size_t {};

  StorageImpl(
      use_byte_size_t /*use_byte_size*/,
      size_t size_bytes,
      at::DataPtr data_ptr,
      at::Allocator* allocator,
      bool resizable)
      : data_ptr_(std::move(data_ptr)),
        size_bytes_(size_bytes),
        resizable_(resizable),
        received_cuda_(false),
        allocator_(allocator) {
    if (resizable) {
      TORCH_INTERNAL_ASSERT(
          allocator_, "For resizable storage, allocator must be provided");
    }
  }

  void* data() {
    return data_ptr_.get();
  }
  at::DeviceType device_type() const {
    return data_ptr_.device().type();
  }

private:
  DataPtr data_ptr_;
  size_t size_bytes_;
  bool resizable_;
  // Identifies that Storage was received from another process and doesn't have
  // local to process cuda memory allocation
  bool received_cuda_;
  Allocator* allocator_;
}
```

StorageImpl的关键成员是data_ptr_，其定义在这里：

```
// c10/core/Allocator.h

class C10_API DataPtr {
 private:
  c10::detail::UniqueVoidPtr ptr_;
  Device device_;


}
```

```cpp
// c10/util/UniqueVoidPtr.h

class UniqueVoidPtr {
 private:
  // Lifetime tied to ctx_
  void* data_;
  std::unique_ptr<void, DeleterFnPtr> ctx_;

  // ...
}
```

现在我们知道，在C++的层面，张量被Tensor类型所表示，但是我们平时是使用Python语言来训练推理模型的，使用
详细的过程我们留到后面的章节解释，不过机制并不复杂，PyTorch使用了THPVariable这个类型作为过渡，Python中
在前面初始化_C模块的时候，调用了THPVariable_initModule()这个函数，将Python中_TensorBase这个类型映射到

```cpp
// torch/csrc/autograd/python_variable.cpp

bool THPVariable_initModule(PyObject *module)
{
  // ...
  PyModule_AddObject(module, "_TensorBase",   (PyObject *)&THPVariableType);
  torch::autograd::initTorchFunctions(module);
  // ...
  return true;
}

PyTypeObject THPVariableType = {
    PyVarObject_HEAD_INIT(
        &THPVariableMetaType,
        0) "torch._C._TensorBase", /* tp_name */
    // ...
    THPVariable_pynew, /* tp_new */
};

PyObject *THPVariable_pynew(PyTypeObject *type, PyObject *args, PyObject *kwargs)
{
  HANDLE_TH_ERRORS
  TORCH_CHECK(type != &THPVariableType, "Cannot directly construct _TensorBase; subclass it
  jit::tracer::warn("torch.Tensor", jit::tracer::WARN_CONSTRUCTOR);
  auto tensor = torch::utils::base_tensor_ctor(args, kwargs);
  // WARNING: tensor is NOT guaranteed to be a fresh tensor; e.g., if it was
  // given a raw pointer that will refcount bump
  return THPVariable_NewWithVar(
      type,
      std::move(tensor),
      c10::impl::PyInterpreterStatus::MAYBE_UNINITIALIZED);
  END_HANDLE_TH_ERRORS
}

static PyObject* THPVariable_NewWithVar(
    PyTypeObject* type,
    Variable _var,
    c10::impl::PyInterpreterStatus status) {

  PyObject* obj = type->tp_alloc(type, 0);
  if (obj) {
    auto v = (THPVariable*) obj;
    // TODO: named constructor to avoid default initialization
    new (&v->cdata) MaybeOwned<Variable>();
    v->cdata = MaybeOwned<Variable>::owned(std::move(_var));
    const auto& var = THPVariable_Unpack(v);
    var.unsafeGetTensorImpl()->init_pyobj(self_interpreter.get(), obj, status);
    if (check_has_torch_dispatch(obj)) {
      var.unsafeGetTensorImpl()->set_python_dispatch(true);
    }
  }
  return obj;
}

// torch/csrc/autograd/python_variable.h
struct THPVariable {
  PyObject_HEAD;
  c10::MaybeOwned<at::Tensor> cdata;
  PyObject* backward_hooks = nullptr;
};
```

TensorOption

Note: 参考注释吧

TensorOption是设计用来构造Tensor的工具。

在C++中没有python中的keyword参数机制，比如这段代码：

```
torch.zeros(2, 3, dtype=torch.int32)
```

在keyword参数机制下，参数的顺序和定义的可能不一样。因此在C++中实现这些函数时，将TensorOptions作为最后

实际使用时，at::zeros()系列函数隐式的使用TensorOptions。 TensorOptions可以看作是一个字典。

Node

Node的定义在torch/csrc/autograd/function.h中。

从名称上不难看出，Node代表计算图中的节点。计算图除了节点之外，还会有边，也就是Edge.

Tensor中方法grad_fn()返回的就是一个Node

Edge

Node的定义在torch/csrc/autograd/edge.h中。

VariableHooks

获取Tensor的grad_fn()时，使用VariableHooks这个类来返回的，而且逻辑很复杂，还没看懂

https://blog.csdn.net/u012436149/article/details/69230136

这里要注意的是，hook 只能注册到 Module 上，即，仅仅是简单的 op 包装的 Module，而不是我们继承 Module时写的那个类，我们继承 Module写的类叫做 Container。 每次调用forward()计算输出的时候，这个hook就会被调用。它应该拥有以下签名：

可以看到，当我们执行model(x)的时候，底层干了以下几件事：

```
forward

  forward_hook    forward     hook    hook
```

register_backward_hook

在module上注册一个bachward hook。此方法目前只能用在Module上，不能用在Container上，当Module的forward函

每次计算module的inputs的梯度的时候，这个hook会被调用。hook应该拥有下面的signature。

hook(module, grad_input, grad_output) -> Tensor or None

如果module有多个输入输出的话，那么grad_input grad_output将会是个tuple。
hook不应该修改它的arguments，但是它可以选择性的返回关于输入的梯度，这个返回的梯度在后续的计算中会替代

这个函数返回一个 句柄(handle)。它有一个方法 handle.remove()，可以用这个方法将hook从module移除。

从上边描述来看，backward hook似乎可以帮助我们处理一下计算完的梯度。看下面nn.Module中register_backward_

Backward函数注册流程

```
initialize_autogenerated_functionsEverything();
   addClass<AddBackward0>(AddBackward0Class,"AddBackward0", AddBackward0_properties);
        _initFunctionPyTypeObject();
        registerCppFunction();
            cpp_function_types[idx] = type
```

参考

- https://blog.csdn.net/Xixo0628/article/details/112603174
- https://blog.csdn.net/Xixo0628/article/details/112603174
- https://pytorch.org/blog/a-tour-of-pytorch-internals-1/#the-thptensor-type
- PyTorch源码浅析(1)：THTensor https://blog.csdn.net/Xixo0628/article/details/112603174
- PyTorch源码浅析(1)：THTensor https://www.52coding.com.cn/2019/05/05/PyTorch1/

PyTorch的算子体系

主要内容

- PyTorch中算子的实现方式
- 源代码的组织
- 运行代码分析
- 自定义算子的实现
- torch模块中的函数
- Tensor算子
- torch.nn的算子
- 算子的注册过程
- 算子的调用过程

一个简单的例子

我们先从一个简单的例子出发，看看PyTorch中Python和C++是怎样一起工作的。

```
import torch

x = torch.ones(2, 2, requires_grad=True)
y = x + 2
```

在_C模块初始化的时候，THPVariable这个类型绑定了相应的方法，可以在执行加法操作的时候，调用的是THPVaria

```
PyMethodDef variable_methods[] = {
  // These magic methods are all implemented on python object to wrap NotImplementedError
  {"__add__", castPyCFunctionWithKeywords(TypeError_to_NotImplemented_<THPVariable_add>), ME
  {"__radd__", castPyCFunctionWithKeywords(TypeError_to_NotImplemented_<THPVariable_add>), M
  {"__iadd__", castPyCFunctionWithKeywords(TypeError_to_NotImplemented_<THPVariable_add_>),
  ...
}
```

THPVariable_add()方法的具体实现代码是生成的，因此我们在原始的模板文件中可以找到使用这个函数，真正的实

```
// torch/csrc/autograd/generated/python_variable_methods.cpp [generated file]

static PyObject * THPVariable_add(PyObject* self_, PyObject* args, PyObject* kwargs)
{
  HANDLE_TH_ERRORS
  const Tensor& self = THPVariable_Unpack(self_);
  static PythonArgParser parser({
    "add(Scalar alpha, Tensor other)|deprecated",
    "add(Tensor other, *, Scalar alpha=1)",
  }, /*traceable=*/true);

  ParsedArgs<2> parsed_args;
  auto _r = parser.parse(self_, args, kwargs, parsed_args);
  if(_r.has_torch_function()) {
    return handle_torch_function(_r, self_, args, kwargs, THPVariableClass, "torch.Tensor")
  }
  switch (_r.idx) {
    case 0: {
      // [deprecated] aten::add.Tensor(Tensor self, Tensor other, *, Scalar alpha=1) -> Tens

      auto dispatch_add = [](const at::Tensor & self, const at::Scalar & alpha, const at::Te
        pybind11::gil_scoped_release no_gil;
        return self.add(other, alpha);
      };
      return wrap(dispatch_add(self, _r.scalar(0), _r.tensor(1)));
    }
    case 1: {
      // aten::add.Tensor(Tensor self, Tensor other, *, Scalar alpha=1) -> Tensor

      auto dispatch_add = [](const at::Tensor & self, const at::Tensor & other, const at::Sc
        pybind11::gil_scoped_release no_gil;
        return self.add(other, alpha);
      };
      return wrap(dispatch_add(self, _r.tensor(0), _r.scalar(1)));
    }
  }
  Py_RETURN_NONE;
  END_HANDLE_TH_ERRORS
}
```

其中 PythonArgParser 定义了这个函数的几类参数，并将Python调用的参数转换成对应的C++类型，在这个例子里，

```
// aten/src/ATen/core/TensorBody.h
// aten::add.Tensor(Tensor self, Tensor other, *, Scalar alpha=1) -> Tensor
inline at::Tensor Tensor::add(const at::Tensor & other, const at::Scalar & alpha) const {
    return at::_ops::add_Tensor::call(const_cast<Tensor&>(*this), other, alpha);
}
```

```
// ./build/aten/src/ATen/Operators_2.cpp [generated file]

STATIC_CONST_STR_OUT_OF_LINE_FOR_WIN_CUDA(add_Tensor, name, "aten::add")
STATIC_CONST_STR_OUT_OF_LINE_FOR_WIN_CUDA(add_Tensor, overload_name, "Tensor")
STATIC_CONST_STR_OUT_OF_LINE_FOR_WIN_CUDA(add_Tensor, schema_str, "add.Tensor(Tensor self, T

// aten::add.Tensor(Tensor self, Tensor other, *, Scalar alpha=1) -> Tensor
static C10_NOINLINE c10::TypedOperatorHandle<add_Tensor::schema> create_add_Tensor_typed_han
  return c10::Dispatcher::singleton()
      .findSchemaOrThrow(add_Tensor::name, add_Tensor::overload_name)
      .typed<add_Tensor::schema>();
}

// aten::add.Tensor(Tensor self, Tensor other, *, Scalar alpha=1) -> Tensor
at::Tensor add_Tensor::call(const at::Tensor & self, const at::Tensor & other, const at::Sca

    static auto op = create_add_Tensor_typed_handle();
    return op.call(self, other, alpha);
}
```

这里创建的op的类型是c10::OperatorHandle

算子分发的基本概念

增加新的算子时，需要先使用TORCH_LIBRARY定义算子的schema，然后使用宏
TORCH_LIBRARY_IMPL来注册该算子在cpu、cuda、XLA等上的实现。注册的时候，需要指定namespace及该namespace

参考官方文档 https://pytorch.org/tutorials/advanced/dispatcher.html

在了解Dispatch的机制之前，我们先了解一下算子的类型。

BackendComponent

每一种“backend”可以看做是一种设备。

```
// c10/core/DispatchKey.h

enum class BackendComponent : uint8_t {
  InvalidBit = 0,
  CPUBit,
  CUDABit,
  HIPBit,
  XLABit,
  MPSBit,
  IPUBit,
  XPUBit,
  HPUBit,
  VEBit,
  LazyBit,
  PrivateUse1Bit,
  PrivateUse2Bit,
  PrivateUse3Bit,
  // Define an alias to represent end of backend dispatch keys.
  // If you add new backend keys after PrivateUse3, please also update it here.
  // (But you shouldn't: private use keys should have higher precedence than
  // all built-in keys)
  EndOfBackendKeys = PrivateUse3Bit,
};
```

DispatchKey

```
// c10/core/DispatchKey.h

enum class DispatchKey : uint16_t {

  Undefined = 0,

  CatchAll = Undefined,

  // ~~~~~~~~~~~~~~~~~~~~~~~~~~ Functionality Keys ~~~~~~~~~~~~~~~~~~~~~~ //
  Dense,

  // Below are non-extensible backends.
  // These are backends that currently don't have their own overrides for
  // Autograd/Sparse/Quantized kernels,
  // and we therefore don't waste space in the runtime operator table allocating
  // space for them.
  // If any of these backends ever need to customize, e.g., Autograd, then we'll
  // need to add a DispatchKey::*Bit for them.

  FPGA, // Xilinx support lives out of tree at
  // https://gitlab.com/pytorch-complex/vitis_kernels

  // ONNX Runtime, lives out of tree at https://github.com/pytorch/ort and
  // https://github.com/microsoft/onnxruntime, and is also used to test general
```

```
// backend/extension machinery in the core. cf:
// - test/cpp_extensions/ort_extension.cpp
// - test/test_torch.py
// - aten/src/ATen/test/extension_backend_test.cpp
ORT,

Vulkan,
Metal,

// A meta tensor is a tensor without any data associated with it.  (They
// have also colloquially been referred to as tensors on the "null" device).
// A meta tensor can be used to dry run operators without actually doing any
// computation, e.g., add on two meta tensors would give you another meta
// tensor with the output shape and dtype, but wouldn't actually add anything.
Meta,

// See [Note: Per-Backend Functionality Dispatch Keys]
Quantized,

// This backend is to support custom RNGs; it lets you go
// to a different kernel if you pass in a generator that is not a
// traditional CPUGeneratorImpl/CUDAGeneratorImpl.  To make use of this
// key:
//  1) set it as a second parameter of at::Generator constructor call in
//     the user-defined PRNG class.
//  2) use it as a dispatch key while registering custom kernels
//     (templatized kernels specialized for user-defined PRNG class)
// intended for out of tree use; tested by aten/src/ATen/test/rng_test.cpp
CustomRNGKeyId,

// Here are backends which specify more specialized operators
// based on the layout of the tensor.  Note that the sparse backends
// are one case where ordering matters: sparse multi-dispatches with
// the corresponding dense tensors, and must be handled before them.
MkldnnCPU, // registered at build/aten/src/ATen/RegisterMkldnnCPU.cpp
// NB: not to be confused with MKLDNN, which is Caffe2 only

// See [Note: Per-Backend Functionality Dispatch Keys]
Sparse,

SparseCsrCPU,
SparseCsrCUDA,

// Note [Non-Customizable Backend Keys]
// Every key above here is considered a "non-customizable backend".
// These are backends that will work correctly with autograd, but
// but currently don't require separate implementations
// for autograd sparse or quantized kernels.
// Any new backends that don't need to be customized should go above here.
// If an existing backend needs to e.g. override autograd, then we can
// consider promoting it into the "BackendComponent" enum
//
// For all intents and purposes from the perspective of DispatchKeySet,
// "non-customizable backend" keys are treated the same way
// as other functionality keys
EndOfNonCustomizableBackends = SparseCsrCUDA,

NestedTensor,

// In some situations, it is not immediately obvious what the correct
// backend for function is, because the function in question doesn't
// have any "tensor" arguments.  In this case, a BackendSelect function
// can be registered to implement the custom determination of the
// correct backend.
BackendSelect,

Python,

// Out-of-core key for Fake Tensor in torchdistx.
// See https://pytorch.org/torchdistx/latest/fake_tensor.html
Fake,

// The named dispatch key is set for any tensors with named dimensions.
// Although we have a dispatch key for named tensors, for historical reasons,
// this dispatch key doesn't do any of the substantive functionality for named
// tensor (though, hypothetically, it could!)  At the moment, it's just
// responsible for letting us give good error messages when operations
// don't support named tensors.
//
// NB: If you ever consider moving named tensor functionality into
// this dispatch key, note that it might be necessary add another dispatch
// key that triggers before composite operators, in case a composite operator
// has named dimension propagation that doesn't match that of its
// constituent parts.
Named,

// The Conjugate dispatch key is set for any tensors that need to perform
// conjugation
// This is implemented at a dispatch level right before any backends run
Conjugate,

// The Negative dispatch key is set for any tensors that need to perform
// negation
// This is implemented at a dispatch level right before any backends run
Negative,

ZeroTensor, // registered at build/aten/src/ATen/RegisterZeroTensor.cpp

// See Note [Out-of-tree vmap+grad prototype]. The purpose of this key
// is to insert code after the "autograd subsystem" runs, so this key should
// be directly after ADInplaceOrView and all of the autograd keys.
FuncTorchDynamicLayerBackMode,

// Note [ADInplaceOrView key]
// ADInplaceOrView key is used by inplace or view ops to register a kernel
// that does additional setup for future autograd computation.
//
// 1. For inplace ops this kernel does version bump
// 2. For view ops this kernel does `as_view` setup where we properly setup
//    DifferentiableViewMeta on the view tensors.
//
// For other ops it's fallthrough kernel since there's no extra
// work to do.
//
// Note [Dream: skip VariableType kernel when requires_grad=false]
//
// In an ideal world where we can skip VariableType kernel for inputs
// with requires_grad=false, instead of a fallthrough kernel, we'll
// register a kernel shown below to all functional ops as well:
// torch::Tensor my_functional_op(...) {
//   {
//     // Note for every op in VariableType, you need to go through
//     // `AutoDispatchBelowADInplaceOrView` guard exactly once to add the
//     // key to TLS excluded set. If you don't go through it at all,
//     // inplace/view ops called through `at::` inside your backend
//     // kernel will dispatch to ADInplaceOrView kernels and do a lot
//     // of extra work.
//     at::AutoDispatchBelowADInplaceOrView guard;
//     at::redispatch::my_functional_op(...);
//   }
// }
// But this work is currently blocked since it adds an extra dispatch
// for all ops and it's non-trivial overhead at model level(a few percents).
// Thus our current approach takes advantage of the fact every kernel go
// through VariableType kernel first and pulls the
// `at::AutoDispatchBelowADInplaceOrView` guard of functional ops
// up to the `VariableType` kernel. Thus we only add the extra dispatch
// to view/inplace ops to minimize its perf impact to real models.
```

```
ADInplaceOrView,
// Note [Alias Dispatch Key : Autograd]
// All backends are oblivious to autograd; autograd is handled as a
// layer which happens on top of all backends. It inspects the autograd
// metadata of all inputs, determines what autograd metadata should be
// constructed by the output, and otherwise defers to the backend to
// actually do the numeric computation.  Autograd contains
// the bulk of this logic.

// Autograd is now an alias dispatch key which by default maps to all
// backend-specific autograd keys.
// Backend-specific allow backends to override the default kernel registered
// to Autograd key as needed.
// For example, XLA wants to define autograd for einsum directly.
// Registering a custom autograd implementation at the XLA key won't work
// because we process Autograd before XLA.  This key has higher priority and
// gets processed first.  You generally should NOT redispatch after handling
// autograd here (since that would result in execution of the Autograd
// operator, which you're trying to skip).  In AutogradXLA implementations,
// you are responsible for handling autograd yourself, or deferring to other
// operators which support autograd.

// Currently we only have backend-specific autograd keys for CPU/CUDA/XLA and
// reserved user-defined backends. All other in-tree backends share the
// AutogradOther key. We can add specific autograd key for those backends
// upon request.
AutogradOther,

// See [Note: Per-Backend Functionality Dispatch Keys]
AutogradFunctionality,

// NestedTensor is an example of something that isn't a "real backend"
// (because it mostly consists of redispatching kernels)
// but it would like to override autograd functionality in C++.
// We can handle cases like this by adding an extra functionality key
// exclusively for handling autograd for NestedTensor.
// lives out of tree at
// https://github.com/pytorch/nestedtensor
AutogradNestedTensor,

Tracer,

// Autocasting precedes VariableTypeId, to ensure casts are autograd-exposed
// and inputs are saved for backward in the post-autocast type.
AutocastCPU,
AutocastXPU,
// Naughtily, AutocastCUDA is also being used for XLA.  In the terminal state,
// it probably should get its own Autocast key
AutocastCUDA,

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~ WRAPPERS ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ //
// There are a number of alternative modes which may want to handle before
// autograd; for example, error checking, tracing, profiling or vmap.  They
// go here.

FuncTorchBatched, // See Note [Out-of-tree vmap+grad prototype]
FuncTorchVmapMode, // See Note [Out-of-tree vmap+grad prototype]

// This is the dispatch key for BatchedTensorImpl, which is used to implement
// batching rules for vmap.
Batched,

// When we are inside a vmap, all tensors dispatch on this key.
// See Note: [DispatchKey::VmapMode usage] for more details.
VmapMode,

FuncTorchGradWrapper, // See Note [Out-of-tree vmap+grad prototype]

// Alias and mutation removal.
// If some backends want to opt into only alias removal or only mutation
// removal,
// we can consider adding separate keys dedicated to those individual passes.
// See Note [Functionalization Pass In Core] for details.
Functionalize,

// Out-of-core key for Deferred Module Initialization in torchdistx.
// See https://pytorch.org/torchdistx/latest/deferred_init.html
DeferredInit,

// Used by Python key logic to know the set of tls on entry to the dispatcher
// This kernel assumes it is the top-most non-functorch-related DispatchKey.
// If you add a key above, make sure to update the fallback implementation for
// this.
PythonTLSSnapshot,

// This key should be at the very top of the dispatcher
FuncTorchDynamicLayerFrontMode, // See Note [Out-of-tree vmap+grad prototype]

// TESTING: This is intended to be a generic testing tensor type id.
// Don't use it for anything real; its only acceptable use is within a single
// process test.  Use it by creating a TensorImpl with this DispatchKey, and
// then registering operators to operate on this type id.  See
```

```
// aten/src/ATen/core/dispatch/backend_fallback_test.cpp for a usage example.
TESTING_ONLY_GenericWrapper,

// TESTING: This is intended to be a generic testing tensor type id.
// Don't use it for anything real; its only acceptable use is within a ingle
// process test.  Use it by toggling the mode on and off via
// TESTING_ONLY_tls_generic_mode_set_enabled and then registering operators
// to operate on this type id.  See
// aten/src/ATen/core/dispatch/backend_fallback_test.cpp
// for a usage example
TESTING_ONLY_GenericMode,

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ FIN ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ //
EndOfFunctionalityKeys, // End of functionality keys.

// ~~~~~~~~~~~~~~ "Dense" Per-Backend Dispatch keys ~~~~~~~~~~~~~~~~~~~~ //
// Here are backends which you think of as traditionally specifying
// how to implement operations on some device.

// See Note [The Ordering of Per-Backend Dispatch Keys Matters!]
StartOfDenseBackends,
CPU, // registered at build/aten/src/ATen/RegisterCPU.cpp
CUDA, // registered at build/aten/src/ATen/RegisterCUDA.cpp
HIP, // NB: I think this is not actually used, due to Note [Masquerading as
// CUDA]
XLA, // lives out of tree at https://github.com/pytorch/xla
MPS, // registered at build/aten/src/ATen/RegisterMPS.cpp
IPU, // lives out of tree at https://github.com/graphcore/poptorch
XPU, // For out of tree Intel's heterogeneous computing plug-in
HPU, // For out of tree & closed source integration of HPU / Habana
VE, // For out of tree & closed source integration of SX-Aurora / NEC
Lazy, // For lazy tensor backends
// Here are reserved backends for user-defined backends, see Note [Private use
// DispatchKey]
// To see some example about how to use this, check out ORT
PrivateUse1,
PrivateUse2,
PrivateUse3,
EndOfDenseBackends = PrivateUse3,

// ~~~~~~~~~~~~~~ "Quantized" Per-Backend Dispatch keys ~~~~~~~~~~~~~~~~ //
// keys starting with an _ are not currently used,
// but are needed to ensure that every backend is indexed correctly.

// See Note [The Ordering of Per-Backend Dispatch Keys Matters!]
StartOfQuantizedBackends,
```

```
QuantizedCPU, // registered at build/aten/src/ATen/RegisterQuantizedCPU.cpp
QuantizedCUDA, // registered at build/aten/src/ATen/RegisterQuantizedCUDA.cpp
_QuantizedHIP,
_QuantizedXLA,
_QuantizedMPS,
_QuantizedIPU,
QuantizedXPU, // For out of tree Intel's heterogeneous computing plug-in
_QuantizedHPU,
_QuantizedVE,
_QuantizedLazy,
_QuantizedPrivateUse1,
_QuantizedPrivateUse2,
_QuantizedPrivateUse3,
EndOfQuantizedBackends = _QuantizedPrivateUse3,

// ~~~~~~~~~~~~~~ "Sparse" Per-Backend Dispatch keys ~~~~~~~~~~~~~~~~~~~ //
// keys starting with an _ are not currently used,
// but are needed to ensure that every backend is indexed correctly.

// See Note [The Ordering of Per-Backend Dispatch Keys Matters!]
StartOfSparseBackends,
SparseCPU, // registered at build/aten/src/ATen/RegisterSparseCPU.cpp
SparseCUDA, // registered at build/aten/src/ATen/RegisterSparseCUDA.cpp
SparseHIP, // TODO: I think this is not actually used, due to Note
// [Masquerading as CUDA]
_SparseXLA,
_SparseMPS,
_SparseIPU,
SparseXPU, // For out of tree Intel's heterogeneous computing plug-in
_SparseHPU,
SparseVE, // For out of tree & closed source integration of SX-Aurora / NEC
_SparseLazy,
_SparsePrivateUse1,
_SparsePrivateUse2,
_SparsePrivateUse3,
EndOfSparseBackends = _SparsePrivateUse3,

// ~~~~~~~~~~~~~~ "NestedTensor" Per-Backend Dispatch keys ~~~~~~~~~~~~~~~~~~
// //
// keys starting with an _ are not currently used,
// but are needed to ensure that every backend is indexed correctly.

// See Note [The Ordering of Per-Backend Dispatch Keys Matters!]
StartOfNestedTensorBackends,
// registered at build/aten/src/ATen/RegisterNestedTensorCPU.cpp
NestedTensorCPU,
```

```
// registered at build/aten/src/ATen/RegisterNestedTensorCUDA.cpp
NestedTensorCUDA,
_NestedTensorHIP,
_NestedTensorXLA,
_NestedTensorMPS,
_NestedTensorIPU,
_NestedTensorXPU,
_NestedTensorHPU,
_NestedTensorVE,
_NestedTensorLazy,
_NestedTensorPrivateUse1,
_NestedTensorPrivateUse2,
_NestedTensorPrivateUse3,
EndOfNestedTensorBackends = _NestedTensorPrivateUse3,

// ~~~~~~~~~~~~~~ "Autograd" Per-Backend Dispatch keys ~~~~~~~~~~~~~~~~~ //
// keys starting with an _ are not currently used,
// but are needed to ensure that every backend is indexed correctly.

// See Note [The Ordering of Per-Backend Dispatch Keys Matters!]
StartOfAutogradBackends,
AutogradCPU,
AutogradCUDA,
_AutogradHIP,
AutogradXLA,
AutogradMPS,
AutogradIPU,
AutogradXPU,
AutogradHPU,
_AutogradVE,
AutogradLazy,
// Here are some reserved pre-autograd keys for user-defined backends, see
// Note [Private use DispatchKey]
AutogradPrivateUse1,
AutogradPrivateUse2,
AutogradPrivateUse3,
EndOfAutogradBackends = AutogradPrivateUse3,
// If we add a new per-backend functionality key that has higher priority
// than Autograd, then this key should be updated.
EndOfRuntimeBackendKeys = EndOfAutogradBackends,

// ~~~~~~~~~~~~~~~~~~~~~~ Alias Dispatch Keys ~~~~~~~~~~~~~~~~~~~~~~~~~~ //
// Note [Alias Dispatch Keys]
// Alias dispatch keys are synthetic dispatch keys which map to multiple
// runtime dispatch keys. Alisa keys have precedence, but they are always
// lower precedence than runtime keys. You can register a kernel to an
```

```
  // alias key, the kernel might be populated to the mapped runtime keys
  // during dispatch table computation.
  // If a runtime dispatch key has multiple kernels from alias keys, which
  // kernel wins is done based on the precedence of alias keys (but runtime
  // keys always have precedence over alias keys).
  // Alias keys won't be directly called during runtime.

  // See Note [Alias Dispatch Key : Autograd]
  Autograd,
  CompositeImplicitAutograd, // registered at
  // build/aten/src/ATen/RegisterCompositeImplicitAutograd.cpp
  CompositeExplicitAutograd, // registered at
  // build/aten/src/ATen/RegisterCompositeExplicitAutograd.cpp

  // Define an alias key to represent end of alias dispatch keys.
  // If you add new alias keys after Autograd, please also update it here.
  StartOfAliasKeys = Autograd,
  EndOfAliasKeys = CompositeExplicitAutograd, //

  // ~~~~~~~~~~~~~~~~~~~~~~~~~ BC ALIASES ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ //
  // The aliases exist for backwards compatibility reasons, they shouldn't
  // be used
  CPUTensorId = CPU,
  CUDATensorId = CUDA,
  DefaultBackend = CompositeExplicitAutograd,
  PrivateUse1_PreAutograd = AutogradPrivateUse1,
  PrivateUse2_PreAutograd = AutogradPrivateUse2,
  PrivateUse3_PreAutograd = AutogradPrivateUse3,
  Autocast = AutocastCUDA,
};
```

DispatchKeySet

所有的算子都是注册在Dispatcher里的，在调用的时候，根据函数名词和传递的参数类型，dispatcher会寻找相应的

下面内容来自PyTorch源码中对DispatchKeySet的注释（翻译不准确的请指正）： > DispatchKeySet就是一组DispatchKey，包括了“functionality”和“backend”两种比特位，每个tensor都有自己 > Dispatcher根据tensor的keyset或者多个tensor的keyset组合，实现了不同的dispatch，并分发到不同的实现（fu > 在内部实现上，Dispatch key 被打包成64位的DispatchKeySet对象。 > 总的key的数量是[backends] * [functionalities]，因此直接把每个key与每个bit关联是不太合适的，key太多了， > 两个枚举值（BackendComponent和DispatchKey）可以被分为5个类别： > (1) “Building block” keys > (a) backends: BackendComponent枚举，比如CPUBit, CUDABit > (b) functionalities (per-backend) 功能相关的dispatch key，比如 AutogradFunctionality, Sparse, Dense > (2) ” Runtime” keys > (a) “non-customizable backends”，比如FPGA > (b) “non-customizable function-

alites”，比如Functionalize > (c) “per-backend instances of customizable functionalities”， 比如CPU, SparseCPU, AutogradCPU > (3) “Alias” DispatchKeys > > (1) Building block的key可以组合成一个运行时使用的DispatchKeySet，例如：
> auto dense_cpu_ks = DispatchKeySet({DispatchKey::CPUBit, > DispatchKey::Dense}); > // The keyset has the runtime dense-cpu key. > dense_cpu_ks.has(DispatchKey::CPU); > // And it contains the building block keys too. > dense_cpu_ks.has(DispatchKey::CPUBit); > dense_cpu_ks.has(DispatchKey::Dense); > 但不是所有的backend或者functionality都可以作为building block，这样就允许了更灵活的设计 > ### Dispatcher

Dispatcher的作用是根据实际的上下文选择不同的operator实现，

```
class TORCH_API Dispatcher final {
private:

  struct OperatorDef final { ... };

public:
  static Dispatcher& realSingleton();

  C10_ALWAYS_INLINE static Dispatcher& singleton() { ...  }

  c10::optional<OperatorHandle> findSchema(const OperatorName& operator_name);

  OperatorHandle findSchemaOrThrow(const char* name, const char* overload_name);

  c10::optional<OperatorHandle> findOp(const OperatorName& operator_name);

  const std::vector<OperatorName> getAllOpNames();

  template<class Return, class... Args>
  Return call(const TypedOperatorHandle<Return (Args...)>& op, Args... args) const;

  template<class Return, class... Args>
  Return redispatch(const TypedOperatorHandle<Return (Args...)>& op, DispatchKeySet currentD

  // Invoke an operator via the boxed calling convention using an IValue stack
  void callBoxed(const OperatorHandle& op, Stack* stack) const;

  // TODO: This will only be useful if we write a backend fallback that plumbs dispatch keys
  // See Note [Plumbing Keys Through The Dispatcher]
  void redispatchBoxed(const OperatorHandle& op, DispatchKeySet dispatchKeySet, Stack* stack


  RegistrationHandleRAII registerDef(FunctionSchema schema, std::string debug);
  RegistrationHandleRAII registerImpl(OperatorName op_name, c10::optional<DispatchKey> dispa
  RegistrationHandleRAII registerName(OperatorName op_name);

  RegistrationHandleRAII registerFallback(DispatchKey dispatch_key, KernelFunction kernel, s

  RegistrationHandleRAII registerLibrary(std::string ns, std::string debug);

  std::vector<OperatorName> getRegistrationsForDispatchKey(c10::optional<DispatchKey> k) con

private:
  // ...

  std::list<OperatorDef> operators_;
  LeftRight<ska::flat_hash_map<OperatorName, OperatorHandle>> operatorLookupTable_;
  ska::flat_hash_map<std::string, std::string> libraries_;

  std::array<impl::AnnotatedKernel, num_runtime_entries> backendFallbackKernels_;

  // ...
};
```

算子注册过程

在PyTorch中，全局只有一个唯一的Dispatcher，所有的算子都注册到这个Dispatcher上，因为算子很多，为了方便

TORCH_LIBRARY及Schema说明

TORCH_LIBRARY可以用来注册Schema，在aten这个namespace下，就注册了超过2500个schema。

```
// build/aten/src/ATen/RegisterSchema.cpp

TORCH_LIBRARY(aten, m) {
  // ...
  m.def("cudnn_batch_norm(Tensor input, Tensor weight, Tensor? bias, Tensor? running_mean, T
  m.def("cudnn_batch_norm_backward(Tensor input, Tensor grad_output, Tensor weight, Tensor?
  m.def("cudnn_convolution(Tensor self, Tensor weight, int[] padding, int[] stride, int[] di
  m.def("cudnn_convolution_transpose(Tensor self, Tensor weight, int[] padding, int[] output
    // ...
}
```

我们看一下 TORCH_LIBRARY被定义在torch/library.h中，从这个文件的位置也可以看出其重要性。这个宏有两个参
Library.

```
// torch/library.h

class TorchLibraryInit final {
 private:
  using InitFn = void(Library&);
  Library lib_;

 public:
  TorchLibraryInit(
      Library::Kind kind,
      InitFn* fn,
      const char* ns,
      c10::optional<c10::DispatchKey> k,
      const char* file,
      uint32_t line)
      : lib_(kind, ns, k, file, line) {
    fn(lib_);
  }
};

#define TORCH_LIBRARY(ns, m)                                               \
  static void TORCH_LIBRARY_init_##ns(torch::Library&);                    \
  static const torch::detail::TorchLibraryInit TORCH_LIBRARY_static_init_##ns( \
      torch::Library::DEF,                                                 \
      &TORCH_LIBRARY_init_##ns,                                            \
      #ns,                                                                 \
      c10::nullopt,                                                        \
      __FILE__,                                                            \
      __LINE__);                                                           \
  void TORCH_LIBRARY_init_##ns(torch::Library& m)
```

在这个宏里，首先声明一个算子库的初始化函数，然后创建了一个TorchLibraryInit的实例，这个实例会初始化Lib
在Library的实例化过程中，该Library也会被注册到全局的Dispatcher里，如下面的实现所示，注册的时候以names

```
// aten/src/ATen/core/library.cpp

Library::Library(Kind kind, std::string ns, c10::optional<c10::DispatchKey> k, const char* f
  : kind_(kind)
  , ns_(ns == "_" ? c10::nullopt : c10::make_optional(std::move(ns)))
  , dispatch_key_((!k.has_value() || *k == c10::DispatchKey::CatchAll) ? c10::nullopt : k)
  , file_(file)
  , line_(line)
  {
    switch (kind_) {
      case DEF:
        registrars_.emplace_back(
          c10::Dispatcher::singleton().registerLibrary(
            *ns_, debugString(file_, line_)
          )
        );
      case FRAGMENT:
        //...
        break;
      case IMPL:
        // Nothing to do, everything is OK
        break;
    }
  }
```

TODO: add schema specification

TORCH_LIBRARY_IMPL

每个算子有唯一的schema，但是可能有很多的实现，在实际运行中，PyTorch会通过Dispatcher查找合适的实现并执

算子实现的注册方式是通过TORCH_LIBRARY_IMPL，例如，在下面的代码中，注册了多个Autograd算子和CUDA。

```
// torch/csrc/autograd/generated/VariableTypeEveryThing.cpp

TORCH_LIBRARY_IMPL(aten, Autograd, m) {
  // ...
  m.impl("add.Tensor",
         TORCH_FN(VariableType::add_Tensor)
  );
  m.impl("add.Scalar",
         TORCH_FN(VariableType::add_Scalar)
  );
  // ...
}

// build/aten/src/ATen/RegisterCPU.cpp
TORCH_LIBRARY_IMPL(aten, CPU, m) {
  // ...
    m.impl("add.Tensor", TORCH_FN(wrapper_add_Tensor));
    m.impl("add.out", TORCH_FN(wrapper_add_out_out));
  // ...
}

// build/aten/src/ATen/RegisterCUDA.cpp

TORCH_LIBRARY_IMPL(aten, CUDA, m) {
    //...
    m.impl("cudnn_batch_norm",
    TORCH_FN(wrapper__cudnn_batch_norm));

    m.impl("cudnn_batch_norm_backward",
    TORCH_FN(wrapper__cudnn_batch_norm_backward));
    m.impl("cudnn_convolution",
    TORCH_FN(wrapper__cudnn_convolution));

    m.impl("cudnn_convolution_transpose",
    TORCH_FN(wrapper__cudnn_convolution_transpose));
    //...
}
```

容易看出，TORCH_LIBRARY_IMPL定义了命名空间ns下，DispatchKeySet为CUDA的一组算子实现，开发者可以通过m.i

下面我们看一下这个宏的实现：

```
// torch/library.h

#define TORCH_LIBRARY_IMPL(ns, k, m) _TORCH_LIBRARY_IMPL(ns, k, m, C10_UID)

#define _TORCH_LIBRARY_IMPL(ns, k, m, uid)                              \
  static void C10_CONCATENATE(                                          \
      TORCH_LIBRARY_IMPL_init_##ns##_##k##_, uid)(torch::Library&);    \
  static const torch::detail::TorchLibraryInit C10_CONCATENATE(         \
      TORCH_LIBRARY_IMPL_static_init_##ns##_##k##_, uid)(             \
      torch::Library::IMPL,                                             \
      c10::guts::if_constexpr<c10::impl::dispatch_key_allowlist_check( \
          c10::DispatchKey::k)>(                                        \
          []() {                                                        \
            return &C10_CONCATENATE(                                    \
                TORCH_LIBRARY_IMPL_init_##ns##_##k##_, uid);            \
          },                                                            \
          []() { return [](torch::Library&) -> void {}; }),             \
      #ns,                                                              \
      c10::make_optional(c10::DispatchKey::k),                          \
      __FILE__,                                                         \
      __LINE__);                                                        \
  void C10_CONCATENATE(                                                 \
      TORCH_LIBRARY_IMPL_init_##ns##_##k##_, uid)(torch::Library & m)
```

和宏TORCH_LIBRARY类似，TORCH_LIBRARY_IMPL首先声明一个算子库的初始化函数，然后创建了一个TorchLibraryIn
在Library的实例化过程中，该Library也会被注册到全局的Dispatcher里，如下面的实现所示，注册的时候以names

接下来我们看一下注册方法实现的细节，因为算子对应的实现，也就是kernel
function，是通过m.impl()来注册的，我们看一下该方法的实现：

```
// aten/src/ATen/core/library.cpp

Library& Library::_impl(const char* name_str, CppFunction&& f) & {
  auto name = torch::jit::parseName(name_str);
  auto ns_opt = name.getNamespace();
  //...

  auto dispatch_key = f.dispatch_key_.has_value() ? f.dispatch_key_ : dispatch_key_;
  registrars_.emplace_back(
    c10::Dispatcher::singleton().registerImpl(
      std::move(name),
      dispatch_key,
      std::move(f.func_),
      // NOLINTNEXTLINE(performance-move-const-arg)
      std::move(f.cpp_signature_),
      std::move(f.schema_),
      debugString(std::move(f.debug_), file_, line_)
    )
  );
  return *this;
}

// aten/src/ATen/core/dispatch/Dispatcher.cpp
RegistrationHandleRAII Dispatcher::registerImpl(
  OperatorName op_name,
  c10::optional<DispatchKey> dispatch_key,
  KernelFunction kernel,
  c10::optional<impl::CppSignature> cpp_signature,
  std::unique_ptr<FunctionSchema> inferred_function_schema,
  std::string debug
) {
  std::lock_guard<std::mutex> lock(mutex_);

  auto op = findOrRegisterName_(op_name);

  auto handle = op.operatorDef_->op.registerKernel(
    *this,
    dispatch_key,
    std::move(kernel),
    // NOLINTNEXTLINE(performance-move-const-arg)
    std::move(cpp_signature),
    std::move(inferred_function_schema),
    std::move(debug)
  );

  ++op.operatorDef_->def_and_impl_count;

  return RegistrationHandleRAII([this, op, op_name, dispatch_key, handle] {
    deregisterImpl_(op, op_name, dispatch_key, handle);
  });
}
```

```
// aten/src/ATen/core/dispatch/OperatorEntry.cpp
OperatorEntry::AnnotatedKernelContainerIterator OperatorEntry::registerKernel(
  const c10::Dispatcher& dispatcher,
  c10::optional<DispatchKey> dispatch_key,
  KernelFunction kernel,
  c10::optional<CppSignature> cpp_signature,
  std::unique_ptr<FunctionSchema> inferred_function_schema,
  std::string debug
) {

  //   cpp_signature

  //  schema

  // Add the kernel to the kernels list,
  // possibly creating the list if this is the first kernel.
  // Redirect catchAll registrations to CompositeImplicitAutograd.
  auto& k = dispatch_key.has_value() ? kernels_[*dispatch_key] : kernels_[DispatchKey::Compo

  //  dispatch key,

  // kernel     OperatorEntry dispatch key
#ifdef C10_DISPATCHER_ONE_KERNEL_PER_DISPATCH_KEY
  k[0].kernel = std::move(kernel);
  k[0].inferred_function_schema = std::move(inferred_function_schema);
  k[0].debug = std::move(debug);
#else
  k.emplace_front(std::move(kernel), std::move(inferred_function_schema), std::move(debug))
#endif

  //  dispatch table
  AnnotatedKernelContainerIterator inserted = k.begin();
  // update the dispatch table, i.e. re-establish the invariant
  // that the dispatch table points to the newest kernel
  if (dispatch_key.has_value()) {
    updateDispatchTable_(dispatcher, *dispatch_key);
  } else {
    updateDispatchTableFull_(dispatcher);
  }
  return inserted;
}
```

算子封装

前面介绍到，注册算子的CPU实现的时候，注册的是函数wrapper_add_Tensor：

```
// build/aten/src/ATen/RegisterCPU.cpp

at::Tensor wrapper_add_Tensor(const at::Tensor & self, const at::Tensor & other, const at::S
  structured_ufunc_add_CPU_functional op;
  op.meta(self, other, alpha);
  op.impl(self, other, alpha, *op.outputs_[0]);
  return std::move(op.outputs_[0]).take();
}
```

其中meta函数会调用到命名空间meta下的函数，其中TORCH_META_FUNC2(add, Tensor)等同于“void structured_add_Tensor::meta”。

```
// aten/src/ATen/native/BinaryOps.cpp
namespace meta {

TORCH_META_FUNC2(add, Tensor) (
  const Tensor& self, const Tensor& other, const Scalar& alpha
) {
  build_borrowing_binary_op(maybe_get_output(), self, other);
  native::alpha_check(dtype(), alpha);
}
```

在

```
// build/aten/src/ATen/UfuncCPUkernel_add.cpp

void add_kernel(TensorIteratorBase& iter, const at::Scalar & alpha) {
  at::ScalarType st = iter.common_dtype();
  RECORD_KERNEL_FUNCTION_DTYPE("add_stub", st);
  switch (st) {

AT_PRIVATE_CASE_TYPE("add_stub", at::ScalarType::Bool, bool,
  [&]() {

auto _s_alpha = alpha.to<scalar_t>();
cpu_kernel(iter,
  [=](scalar_t self, scalar_t other) { return ufunc::add(self, other, _s_alpha); }
);

  }
)
```

算子注册：

```
// build/aten/src/ATen/UfuncCPUkernel_add.cpp

using add_fn = void(*)(TensorIteratorBase&, const at::Scalar &);
```

```
DECLARE_DISPATCH(add_fn, add_stub);
REGISTER_DISPATCH(add_stub, &add_kernel);

// aten/src/ATen/native/DispatchStub.cpp

#define DECLARE_DISPATCH(fn, name)         \
  struct name : DispatchStub<fn, name> {   \
    name() = default;                      \
    name(const name&) = delete;            \
    name& operator=(const name&) = delete; \
  };                                       \
  extern TORCH_API struct name name

#define REGISTER_DISPATCH(name, fn) REGISTER_ARCH_DISPATCH(name, CPU_CAPABILITY, fn)
#define REGISTER_ARCH_DISPATCH(name, arch, fn) \
  template <> name::FnPtr TORCH_API DispatchStub<name::FnPtr, struct name>::arch = fn;
```

OperatorHandle

这里看到两种注册的类型，一种是OperatorHandler，注册到operatorLookupTable_中，可以根据OperatorName查询

比如对于例子中的 y = x + 2这条语句，dispatcher会查询到一个OperatorHandler
op， op.operatorDef_->op.name_就是OperatorName(“aten::add”， “Tensor”)，但是注册的kernelfunction很

```
// aten/src/ATen/core/dispatch/Dispatcher.h

class TORCH_API OperatorHandle {
public:
  OperatorHandle(OperatorHandle&&) noexcept = default;
  // ...

  // See [Note: Argument forwarding in the dispatcher] for why Args doesn't use &&
  C10_ALWAYS_INLINE Return call(Args... args) const {
    return c10::Dispatcher::singleton().call<Return, Args...>(*this, std::forward<Args>(args
  }

  // ...

private:
  // ...
  Dispatcher::OperatorDef* operatorDef_;
  std::list<Dispatcher::OperatorDef>::iterator operatorIterator_;
};
```

OperatorHandle的call()方法会调用Dispatcher::call()方法。

继续跟踪，会走到

```
at::native::AVX2::cpu_kernel_vec<> (grain_size=32768, vop=..., op=..., iter=...)
    at ../aten/src/ATen/native/cpu/Loops.h:349
```

```
#0  at::native::AVX2::cpu_kernel_vec<> (grain_size=32768, vop=..., op=..., iter=...)
    at ../aten/src/ATen/native/cpu/Loops.h:349
#1  at::native::(anonymous namespace)::<lambda()>::operator() (__closure=<optimized out>)
    at /lab/tmp/pytorch/build/aten/src/ATen/UfuncCPUKernel_add.cpp:61
#2  at::native::(anonymous namespace)::add_kernel (iter=..., alpha=...)
    at /lab/tmp/pytorch/build/aten/src/ATen/UfuncCPUKernel_add.cpp:61
#3  0x00007fffe717e7be in at::(anonymous namespace)::wrapper_add_Tensor (self=..., other=..
    at aten/src/ATen/RegisterCPU.cpp:1595
```

```
(gdb) bt
#0  at::native::AVX2::vectorized_loop<at::native::(anonymous namespace)::add_kernel(at::Tens
    at ../aten/src/ATen/native/cpu/Loops.h:212
#1  at::native::AVX2::VectorizedLoop2d<at::native::(anonymous namespace)::add_kernel(at::Te
    at ../aten/src/ATen/native/cpu/Loops.h:287
#2  at::native::AVX2::unroll_contiguous_scalar_checks<function_traits<at::native::(anonymou
    cb=..., strides=0x7fffffffd300) at ../aten/src/ATen/native/cpu/Loops.h:246
#3  at::native::AVX2::unroll_contiguous_scalar_checks<function_traits<at::native::(anonymou
    cb=..., strides=0x7fffffffd300) at ../aten/src/ATen/native/cpu/Loops.h:248
#4  at::native::AVX2::VectorizedLoop2d<at::native::(anonymous namespace)::add_kernel(at::Te
    at ../aten/src/ATen/native/cpu/Loops.h:283
#5  c10::function_ref<void(char**, long int const*, long int, long int)>::callback_fn<at::n
    params#0=params#0@entry=0x7fffffffd270, params#1=params#1@entry=0x7fffffffd300, params#
    params#3=params#3@entry=1) at ../c10/util/FunctionRef.h:43
```

算子调用的过程

我们再看一个简单的例子：

```
import torch

x = torch.randn(2,2, requires_grad=True)
y = x + 2
```

在调用上，依次进行如下的调用：

```
// torch/csrc/autograd/generated/python_variable_methods.cpp
//        self_ args    kwargs = 0x0
static PyObject * THPVariable_add(PyObject* self_, PyObject* args, PyObject* kwargs);

    // torch/csrc/utils/python_arg_parser.h
    inline PythonArgs PythonArgParser::parse(PyObject* self, PyObject* args, PyObject* kwarg

        // torch/csrc/utils/python_arg_parser.cpp
        PythonArgs PythonArgParser::raw_parse(PyObject* self, PyObject* args, PyObject* kwar
        bool FunctionSignature::parse(PyObject* self, PyObject* args, PyObject* kwargs, PyOb

    // torch/include/ATen/core/TensorBody.h   --- generated from aten/src/ATen/templates/Te
    inline at::Tensor & Tensor::add(const at::Tensor & other, const at::Scalar & alpha) cons

        // build/aten/src/ATen/Operators_2.cpp
        at::Tensor & add_Tensor::call(at::Tensor & self, const at::Tensor & other, const at:

            // aten/src/ATen/core/dispatch/Dispatcher.cpp
            OperatorHandle Dispatcher::findSchemaOrThrow(const char* name, const char* overl
                c10::optional<OperatorHandle> Dispatcher::findSchema(const OperatorName& ove
                c10::optional<OperatorHandle> Dispatcher::findOp(const OperatorName& overloa

            // aten/src/ATen/core/dispatch/Dispatcher.cpp
            Return TypedOperatorHandle::call(Args... args) const;

                // aten/src/ATen/core/dispatch/Dispatcher.cpp
                Return Dispatcher::call(const TypedOperatorHandle<Return(Args...)>& op, Args

                    // aten/src/ATen/core/dispatch/DispatchKeyExtractor.h
                    DispatchKeySet DispatchKeyExtractor::getDispatchKeySetUnboxed(const Args

                    // aten/src/ATen/core/boxing/KernelFunction.h
                    Return call(const OperatorHandle& opHandle, DispatchKeySet dispatchKeySe



    // torch/csrc/autograd/utils/wrap_outputs.h
    //         Python
    PyObject* wrap(PyTypeObject *type, std::tuple<Ts...> values);
```

在进入C++层面的第一步，是进行调用参数的解码。因为在Python层面和在C++层面类的体系是不一样的，Python语
PyTorch为此定义了PythonArgParser类，在函数被调用的入口处进行参数解析：

```
// torch/csrc/autograd/generated/python_variable_methods.cpp
static PyObject * THPVariable_add(PyObject* self_, PyObject* args, PyObject* kwargs)
{
  HANDLE_TH_ERRORS
  const Tensor& self = THPVariable_Unpack(self_);
  static PythonArgParser parser({
    "add(Scalar alpha, Tensor other)|deprecated",
    "add(Tensor other, *, Scalar alpha=1)",
  }, /*traceable=*/true);
  ParsedArgs<2> parsed_args;
  auto _r = parser.parse(self_, args, kwargs, parsed_args);
  if(_r.has_torch_function()) {
    return handle_torch_function(_r, self_, args, kwargs, THPVariableClass, "torch.Tensor")
  }
  switch (_r.idx) {
    case 0: {
      // [deprecated] aten::add.Tensor(Tensor self, Tensor other, *, Scalar alpha=1) -> Ten

      auto dispatch_add = [](const at::Tensor & self, const at::Scalar & alpha, const at::Te
        pybind11::gil_scoped_release no_gil;
        return self.add(other, alpha);
      };
      return wrap(dispatch_add(self, _r.scalar(0), _r.tensor(1)));
    }
    case 1: {
      // aten::add.Tensor(Tensor self, Tensor other, *, Scalar alpha=1) -> Tensor

      auto dispatch_add = [](const at::Tensor & self, const at::Tensor & other, const at::Sc
        pybind11::gil_scoped_release no_gil;
        return self.add(other, alpha);
      };
      return wrap(dispatch_add(self, _r.tensor(0), _r.scalar(1)));
    }
  }
  Py_RETURN_NONE;
  END_HANDLE_TH_ERRORS
}
```

如上面的代码，对于add方法，Pytorch支持两种不同的签名，但是前一种已经过时了，因此实际调用走的都是第二
C API: PyTuple_GET_ITEM()和PyDict_GetItem()，在调用Tensor::add()之前，PythonArgParser会通过其tensor()

在Tensor::add()的实现中，并不是真正的算子代码，因为刚才只完成了从Python到C++的调用转换，实际的算子实

```
// torch/include/ATen/core/TensorBody.h

// aten::add_.Tensor(Tensor(a!) self, Tensor other, *, Scalar alpha=1) -> Tensor(a!)
inline at::Tensor & Tensor::add_(const at::Tensor & other, const at::Scalar & alpha) const
    return at::_ops::add__Tensor::call(const_cast<Tensor&>(*this), other, alpha);
}


// build/aten/src/ATen/Operators_2.cpp

STATIC_CONST_STR_OUT_OF_LINE_FOR_WIN_CUDA(add__Tensor, name, "aten::add_")
STATIC_CONST_STR_OUT_OF_LINE_FOR_WIN_CUDA(add__Tensor, overload_name, "Tensor")
STATIC_CONST_STR_OUT_OF_LINE_FOR_WIN_CUDA(add__Tensor, schema_str, "add_.Tensor(Tensor(a!) s

// aten::add_.Tensor(Tensor(a!) self, Tensor other, *, Scalar alpha=1) -> Tensor(a!)
static C10_NOINLINE c10::TypedOperatorHandle<add__Tensor::schema> create_add__Tensor_typed_h
  return c10::Dispatcher::singleton()
      .findSchemaOrThrow(add__Tensor::name, add__Tensor::overload_name)
      .typed<add__Tensor::schema>();
}

// aten::add_.Tensor(Tensor(a!) self, Tensor other, *, Scalar alpha=1) -> Tensor(a!)
at::Tensor & add__Tensor::call(at::Tensor & self, const at::Tensor & other, const at::Scalar

    static auto op = create_add__Tensor_typed_handle();
    return op.call(self, other, alpha);
}
```

THPVariable_add ->

自定义算子的实现过程

原生算子的实现

所谓“原生”，指的就是内置在PyTorch中的算子，跟随PyTorch一起编译生成，可以同“torch.xxx”等方式使用的

由于原生算子的数量非常多，处于效率和可用性的考虑，在不同的平台上可能会有实现，另外算子要支持注册到tor

很多原生算子的模板定义在native_functions.yaml中，比如sigmoid函数：

```
# aten/src/ATen/native/native_functions.yaml

- func: sigmoid(Tensor self) -> Tensor
  device_check: NoCheck   # TensorIterator
  structured_delegate: sigmoid.out
  variants: function, method
  dispatch:
    QuantizedCPU: sigmoid_quantized_cpu
    MkldnnCPU: mkldnn_sigmoid


- func: sigmoid_backward(Tensor grad_output, Tensor output) -> Tensor
  python_module: nn
  structured_delegate: sigmoid_backward.grad_input
```

其中： - func字段定义了算子的名称和输入输出参数。 - device_check: 暂时还不清楚用途，在模板里都是NoCheck。 - structured_delegate: sigmoid.out - variants字段生命这个算子的类型和使用方式，function表明sigmoid这个算子可以通过函数torch.sigm

- dispatch字段定义了在不同的平台或者优化方式下该算子的变体。这里针对使用量化方式运行时，会调用相应的量
- python-module字段定义了该算法会被注册到的Python模块。

sigmoid函数是机器学习中最基本的函数之一，其公式如下：

$$Sigmoid(x) = \frac{1}{1 + e^{-x}}$$

我们在使用sigmoid函数时，调用的是torch.nn.Sigmoid函数，其背后则是调用了torch.sigmoid()函数，也就是上面

```
class Sigmoid(Module):
    r"""Applies the element-wise function:
    Examples::
        >>> m = nn.Sigmoid()
        >>> input = torch.randn(2)
        >>> output = m(input)
    """

    def forward(self, input: Tensor) -> Tensor:
        return torch.sigmoid(input)
```

在tools/autograd/derivatives.yaml中，定义了算子的前向计算输出反向计算梯度的对应关系，比如sigmoid算子的

```
- name: sigmoid(Tensor self) -> Tensor
  self: sigmoid_backward(grad, result)
  result: auto_element_wise
```

在native_functions.yaml中只是声明了sigmoid算子，具体的算子实现是和平台相关的，因此要到各个平台目录下去

```
// aten/src/ATen/native/cpu/UnaryOpsKernel.cpp

static void sigmoid_kernel(TensorIteratorBase& iter) {
  if (iter.common_dtype() == kBFloat16) {
    cpu_kernel_vec(
        iter,
        [=](BFloat16 a) -> BFloat16 {
          float a0 = static_cast<float>(a);
          return static_cast<float>(1) / (static_cast<float>(1) + std::exp((-a0)));
        },
        [=](Vectorized<BFloat16> a) {
          Vectorized<float> a0, a1;
          std::tie(a0, a1) = convert_bfloat16_float(a);
          a0 = (Vectorized<float>(static_cast<float>(1)) + a0.neg().exp()).reciprocal();
          a1 = (Vectorized<float>(static_cast<float>(1)) + a1.neg().exp()).reciprocal();
          return convert_float_bfloat16(a0, a1);
        });
  } else {
    AT_DISPATCH_FLOATING_AND_COMPLEX_TYPES(iter.common_dtype(), "sigmoid_cpu", [&]() {
      cpu_kernel_vec(
          iter,
          [=](scalar_t a) -> scalar_t {
            return (static_cast<scalar_t>(1) / (static_cast<scalar_t>(1) + std::exp((-a)));
          },
          [=](Vectorized<scalar_t> a) {
            a = Vectorized<scalar_t>(static_cast<scalar_t>(0)) - a;
            a = a.exp();
            a = Vectorized<scalar_t>(static_cast<scalar_t>(1)) + a;
            a = a.reciprocal();
            return a;
          });
    });
  }
}

REGISTER_DISPATCH(sigmoid_stub, &CPU_CAPABILITY::sigmoid_kernel);


// aten/src/ATen/native/cpu/BinaryOpsKernel.cpp

void sigmoid_backward_kernel(TensorIteratorBase& iter) {
  if (isComplexType(iter.dtype())) {
    // ......
  } else if (iter.dtype() == kBFloat16) {
    // ......
  } else {
    // ......
  }
}
// aten/src/ATen/native/cpu/UnaryOps.cpp

CREATE_UNARY_FLOAT_META_FUNC(sigmoid)

CREATE_UNARY_TORCH_IMPL_FUNC(sigmoid_out, sigmoid_stub)
DEFINE_DISPATCH(sigmoid_stub); // NOLINT(cppcoreguidelines-avoid-non-const-global-variables
```

在sigmoid_kernel()的实现里，根据传输Tensor类型的不同，构建了不同的匿名函数，然后调用cpu_kernel_vec()对

sigmoid_kernel是sigmoid算子在cpu下的实现，当然即使在CPU下，sigmoid函数也有多种形式，除了普通的浮点计算
AT_DISPATCH_FLOATING_AND_COMPLEX_TYPES宏有三个参数： - iter.common_dtype()，指明操作的Tensor属于哪种类
- “sigmoid_cpu”， 算子的名称 - 匿名函数，调用了cpu_kernel_vec

在aten/src/ATen/native/cpu/Loops.cpp中，有两个cpu_kernel相关的函数，由于cpu下的源文件在编译的时候会加
- cpu_kernel()：依赖于编译器自动实现计算的向量化 - cpu_kernel_vec()：使用x86
SIMD原语实现向量化。 一般来讲，使用cpu_kernel_vec()的时候，说明实现该算子的实现是经过精心优化的，效率

例如用这两个函数实现浮点数相乘的算子，可以这样实现：

```
cpu_kernel(iter, [](float a, float b) { return a * b; });

cpu_kernel_vec(iter,
     [](float a, float b) { return a * b; },
     [](Vectorized<float> a, Vectorized<float> b) { return a * b; });
```

下面我们看一下cpu_kernel_vec()函数的实现：

```
// aten/src/ATen/native/cpu/Loops.cpp

template <bool check_dynamic_cast=true, typename func_t, typename vec_func_t>
void cpu_kernel_vec(TensorIteratorBase& iter, func_t&& op, vec_func_t&& vop, int64_t grain_s
  using traits = function_traits<func_t>;
  // this could be extended to work with void return types
  TORCH_INTERNAL_ASSERT(iter.ninputs() == traits::arity);
  TORCH_INTERNAL_ASSERT(iter.noutputs() == 1);
  // dynamic casting not currently supported on CPU, but some kernels (like Fill)
  // explicitly dynamic_cast, so we give the opt-out of checking.
  c10::guts::if_constexpr<check_dynamic_cast>([&] {
    TORCH_INTERNAL_ASSERT(!needs_dynamic_casting<func_t>::check(iter));
  });

  iter.for_each(make_vectorized_loop2d(op, vop), grain_size);
  iter.cast_outputs();
}
```

可以看到，对每个Tensor，又调用了make_vectorized_loop2d()

```
// aten/src/ATen/native/cpu/Loops.cpp
template <typename op_t, typename vop_t>
VectorizedLoop2d<op_t, vop_t> make_vectorized_loop2d(
    const op_t &op, const vop_t &vop) {
  return VectorizedLoop2d<op_t, vop_t>(op, vop);
}

template <typename op_t, typename vop_t>
struct VectorizedLoop2d {
  op_t op;
  vop_t vop;

  using traits = function_traits<op_t>;
  static constexpr int ntensors = traits::arity + 1;
  using data_t = std::array<char*, ntensors>;

  VectorizedLoop2d(const op_t &op, const vop_t &vop):
    op(op), vop(vop) {}

  static void advance(data_t &data, const int64_t *outer_strides) {
    for (const auto arg : c10::irange(data.size())) {
      data[arg] += outer_strides[arg];
    }
  }

  void operator()(char** base, const int64_t *strides, int64_t size0, int64_t size1) {
    data_t data;
    std::copy_n(base, ntensors, data.data());
    const int64_t *outer_strides = &strides[ntensors];

    if (is_contiguous<traits>(strides)) {
      for (const auto i : c10::irange(size1)) {
        (void)i;
        vectorized_loop(data.data(), size0, 0, op, vop);
        advance(data, outer_strides);
      }
    } else {
      using Indices = std::make_index_sequence<traits::arity>;
      unroll_contiguous_scalar_checks<traits>(strides, Indices{}, [&](size_t idx) {
        if (idx) {
          for (const auto i : c10::irange(size1)) {
            (void)i;
            vectorized_loop(data.data(), size0, idx, op, vop);
            advance(data, outer_strides);
          }
        } else {
          for (const auto i : c10::irange(size1)) {
            (void)i;
            basic_loop(data.data(), strides, 0, size0, op);
            advance(data, outer_strides);
          }
        }
      });
    }
  }
};
```

很明显，VectorizedLoop2d的主要工作就是根据Tensor的stride的不同，选择不同的调用模式，但最终不管是调用v

现在我们回到当初sigmoid函数的实现部分，其中对每个Tensor的操作函数实现是这样的：

```
// aten/src/ATen/native/cpu/UnaryOpsKernel.cpp
      cpu_kernel_vec(
          iter,
          [=](scalar_t a) -> scalar_t {
            return (static_cast<scalar_t>(1) / (static_cast<scalar_t>(1) + std::exp((-a))))
          },
          [=](Vectorized<scalar_t> a) {
            a = Vectorized<scalar_t>(static_cast<scalar_t>(0)) - a;
            a = a.exp();
            a = Vectorized<scalar_t>(static_cast<scalar_t>(1)) + a;
            a = a.reciprocal();
            return a;
          });

// aten/src/ATen/native/cpu/vec/vec256/vec256_float.h
 Vectorized<float> exp() const {
    return Vectorized<float>(Sleef_expf8_u10(values));
  }
```

在代码中可以看出，对应cpu的实现有很多，实际运行时会根据不同的平台和数据类型调用相应的实现，以达到比较

> https://blog.csdn.net/yelede2009/article/details/120411361
> 有各种函数库以向量方式来计算数学函数，例如：对数、幂函数、三角函数等。这些函数库对向量化数学代码
> 有两种不同种类的向量数学库：长向量库和短向量库。来看看它们的不同。假设要计算1000个数字的某个函数
> 个库函数存储这1000个结果到另一个数组。使用长向量版库函数的缺点是，如果要做一系列计算，在下一次调
> 的向量库，可以把数据集拆分为子向量来适配向量寄存器。如果向量寄存器可以处理4个数字，那么需要调用2
> 被下一次计算利用，而不需要存储中间结果到RAM中。这可能更快。然而，短向量的库函数可能是不利的，如果
> 这是一些长向量函数库：
>
> Intel 向量数学库（VML, MKL）。工作在x86平台。这些库函数在非Intel的CPU上会低效，除非重写了Intel
> cpu分发器。 Intel的IPP。工作在x86平台。也适用于非Intel的CPU。包含很多统计、信号处理和图像处理函数
> Yeppp。开源库。支持x86和ARM平台，多种编程语言。参考Yeppp。
>
> 这是一些短向量库：
>
> Sleef库。支持多种平台。开源。参考www.sleef.org。 Intel短向量库（SVML）。Intel编译器提供，被自动向
> mveclibabi=svml使用这个库。如果用的是非Intel的CPU，也可以使用。
> AMD LIBM库。只支持64位Linux平台。没有FMA4指令集时，性能会降低。Gnu通过-
> mveclibabi=acml选项使用。 VCL库。个人开发。参考https://github.com/vectorclass。

Dispatch的过程似乎有些复杂，有很多宏处理，更是导致不容易看懂。 “ ‘C++ //
aten/src/ATen/Dispatch.h

define AT_PRIVATE_CASE_TYPE(NAME, enum_type, type, …)
AT_PRIVATE_CASE_TYPE_USING_HINT(NAME, enum_type, type, scalar_t, VA_ARGS)

define AT_DISPATCH_FLOATING_TYPES_AND_HALF(TYPE, NAME, …)
[&] {
const auto& the_type = TYPE;
/* don’ t use TYPE again in case it is an expensive or side-effect op */
at::ScalarType _st = ::detail::scalar_type(the_type);
RECORD_KERNEL_FUNCTION_DTYPE(NAME, _st);
switch (_st) {
AT_PRIVATE_CASE_TYPE(NAME, at::ScalarType::Double, double, VA_ARGS)
AT_PRIVATE_CASE_TYPE(NAME, at::ScalarType::Float, float, VA_ARGS)
AT_PRIVATE_CASE_TYPE(NAME, at::ScalarType::Half, at::Half, VA_ARGS)
default:
AT_ERROR(#NAME, ” not implemented for ’ “, toString(_st), “” ’);
}
}() “ ‘

宏AT_DISPATCH_FLOATING_AND_COMPLEX_TYPES

参考

- https://pytorch.org/tutorials/advanced/dispatcher.html
- http://blog.ezyang.com/2020/09/lets-talk-about-the-pytorch-dispatcher/

- https://blog.csdn.net/Chris_zhangrx/article/details/119512418
- https://zhuanlan.zhihu.com/p/67834038
- https://blog.csdn.net/xixiaoyaoww/article/details/112211025
- pytorch中的dispatcher https://zhuanlan.zhihu.com/p/390049109
- [Pytorch 源码阅读] —— 谈谈 dispatcher（二）https://blog.csdn.net/Chris_zhangrx/article/details/
- [Pytorch 源码阅读] —— 谈谈 dispatcher（一） https://blog.csdn.net/Chris_zhangrx/article/details/
- https://zhuanlan.zhihu.com/p/349560723
- https://zhuanlan.zhihu.com/p/499979372
- 这可能是关于Pytorch底层算子扩展最详细的总结了 https://wenku.baidu.com/view/1415b43ac181e53a58021

计算图

基本内容

本章内容主要回答以下几个问题：

神经网络的基本结构

深度学习框架时如何执行计算图的

计算图执行过程中的基本数据结构

PyTorch中的具体实现

神经网络的基本结构

深度学习解决的是深度神经网络的优化问题，虽然深度神经网络的模型种类繁多，从最简单的MLP模型到近年流行的

```
import torch
from torch import nn

class DemoNet(nn.Module):
    def __init__(self):
        super(DemoNet, self).__init__()
        self.w = torch.rand(2,2)
    def forward(self, x):
        y = self.w * x
        return y * y

input = torch.rand(2, 2)
model = DemoNet()
```

使用TensorBoard查看该网络的可视化，如下图：

其中y处是一个算子”Operation: aten::mul“

虽然上面只是最简单的一个例子，但也包括了神经网络作为有向无环图的基本结构：
- 顶点：代表一个输入数据、算子、或者输出数据 - 边：代表数据和算子、算子和算子之间的输入输出关系。

深度神经网络包括结果的前向计算过程和梯度的反向传播过程，显而易见的是，深度学习框架需要事先构造计算图，
- 根据代码逻辑，构造好一个计算图，之后这个计算图可以反复执行 - 每次在执行时，都重新构造好计算图

PyTorch选择的是第二种方式，也就是动态图的方式。动态图的好处是可以在代码逻辑中使用各种条件判断。

PyTorch中计算图的实现

虽然不是所有的计算图都通过上面的例子中的nn.Module来实现，但nn.Module确实是PyTorch中神经网络的基础结构

```python
# torch/nn/modules/module.py

class Module:
    r"""Base class for all neural network modules.
    ...
    """

     training: bool
    _is_full_backward_hook: Optional[bool]

    def __init__(self) -> None:
        """
        Initializes internal Module state, shared by both nn.Module and ScriptModule.
        """
        torch._C._log_api_usage_once("python.nn_module")

        self.training = True
        self._parameters: Dict[str, Optional[Parameter]] = OrderedDict()
        self._buffers: Dict[str, Optional[Tensor]] = OrderedDict()
        self._non_persistent_buffers_set: Set[str] = set()
        self._backward_hooks: Dict[int, Callable] = OrderedDict()
        self._is_full_backward_hook = None
        self._forward_hooks: Dict[int, Callable] = OrderedDict()
        self._forward_pre_hooks: Dict[int, Callable] = OrderedDict()
        self._state_dict_hooks: Dict[int, Callable] = OrderedDict()
        self._load_state_dict_pre_hooks: Dict[int, Callable] = OrderedDict()
        self._load_state_dict_post_hooks: Dict[int, Callable] = OrderedDict()
        self._modules: Dict[str, Optional['Module']] = OrderedDict()

    forward: Callable[..., Any] = _forward_unimplemented
```

Module类的主要属性及方法如下：

一个神经网络，最重要的是其内部的参数，在Module中有两个属性和参数相关：_parameters和_buffers，它们的类

从定义上看，_buffers中存放的是Tensor类型的数据，而_parameters中存放的是Parameter类型的数据，在构造时参

```python
# torch/nn/parameter.py

class Parameter(torch.Tensor, metaclass=_ParameterMeta):
    def __new__(cls, data=None, requires_grad=True):
        # ......
```

当构造好Parameter并且赋值给nn.Module时，会自动调用nn.Module的register_parameter()方法进行注册。

```python
# torch/nn/modules/module.py

class Module:

    def __setattr__(self, name: str, value: Union[Tensor, 'Module']) -> None:

        params = self.__dict__.get('_parameters')
        if isinstance(value, Parameter):
            self.register_parameter(name, value)
        # handle value with other types
```

为了看的更清楚一些，我们看一下PyTorch中内置的网络组件，例如：

```python
# torch/nn/modules/conv.py

class _ConvNd(Module):

    __constants__ = ['stride', 'padding', 'dilation', 'groups',
                     'padding_mode', 'output_padding', 'in_channels',
                     'out_channels', 'kernel_size']
    __annotations__ = {'bias': Optional[torch.Tensor]}

    def _conv_forward(self, input: Tensor, weight: Tensor, bias: Optional[Tensor]) -> Tenso
        ...

    _in_channels: int
    _reversed_padding_repeated_twice: List[int]
    out_channels: int
    kernel_size: Tuple[int, ...]
    stride: Tuple[int, ...]
    padding: Union[str, Tuple[int, ...]]
    dilation: Tuple[int, ...]
    transposed: bool
    output_padding: Tuple[int, ...]
    groups: int
    padding_mode: str
    weight: Tensor
    bias: Optional[Tensor]

    def __init__(self,
                 in_channels: int,
                 out_channels: int,
                 kernel_size: Tuple[int, ...],
                 stride: Tuple[int, ...],
                 padding: Tuple[int, ...],
                 dilation: Tuple[int, ...],
                 transposed: bool,
                 output_padding: Tuple[int, ...],
                 groups: int,
                 bias: bool,
                 padding_mode: str,
                 device=None,
                 dtype=None) -> None:
        super(_ConvNd, self).__init__()

        # check and handle padding and other parameter...

        if transposed:
            self.weight = Parameter(torch.empty(
                (in_channels, out_channels // groups, *kernel_size), **factory_kwargs))
        else:
            self.weight = Parameter(torch.empty(
                (out_channels, in_channels // groups, *kernel_size), **factory_kwargs))
        if bias:
            self.bias = Parameter(torch.empty(out_channels, **factory_kwargs))
        else:
            self.register_parameter('bias', None)

        self.reset_parameters()
```

计算图的执行过程

在深度学习中，我们的神经网络一般是基于nn.Module实现的，典型的调用方式是：

```
    y = DemoNet(x)
    loss = compute_loss(y, label)
```

可见计算图的执行其实就是nn.Module的调用过程，从下面的实现中可以看出，主要的工作就是调用forward()方法

```
# torch/nn/modules/module.py

class Module:
    def _call_impl(self, *input, **kwargs):
        forward_call = (self._slow_forward if torch._C._get_tracing_state() else self.forwar

        # YL: handle pre-forward hooks, you can change input here
        # ...

        result = forward_call(*input, **kwargs)
        # YL: handle forward hooks
        # ...

        # Handle the non-full backward hooks
        # ...

        return result

    __call__ : Callable[..., Any] = _call_impl
```

相应的，我们可以看一下卷积操作的实现：

```
# torch/nn/modules/conv.py

from .. import functional as F

class Conv2d(_ConvNd):

    ## YL  __init__() implemetation here


    def _conv_forward(self, input: Tensor, weight: Tensor, bias: Optional[Tensor]):
        if self.padding_mode != 'zeros':
            return F.conv2d(F.pad(input, self._reversed_padding_repeated_twice, mode=self.pa
                            weight, bias, self.stride,
                            _pair(0), self.dilation, self.groups)
        return F.conv2d(input, weight, bias, self.stride,
                        self.padding, self.dilation, self.groups)

    def forward(self, input: Tensor) -> Tensor:
        return self._conv_forward(input, self.weight, self.bias)
```

由此可见，卷积算子的实现调用了functional模块中的卷积函数。这也说明，在PyTorch中，神经网络的定义和算子

Dispatch

参考

- https://zhuanlan.zhihu.com/p/89442276

自动微分

自动微分一直被视为深度学习框架的核心能力，在训练深度学习神经网络的时候，网络的参数需要根据输出端的梯度

自动微分的理论基础

在了解自动微分之前，我们先从优化的角度看一下参数和梯度的关系，这也是深度学习的目标。

考虑下面这个公式，这是典型的线性回归的公式，我们需要根据输出与实际值的差异调整系数w及截距b：

$$y = w * x + b$$

根据微分原理我们知道：

$$\frac{\partial y}{\partial w} = x$$

$$\frac{\partial y}{\partial b} = 1$$

根据上面的式子，在微小的取值范围内，为了调整w，可以这样计算：

$$\mathrm{d}w = x * \mathrm{d}y$$

其中$\mathrm{d}y$ 就是输出与实际值的差异。在实际计算中，由于$\mathrm{d}y$的值不会很小，我们会加一个比较小的系数α来缓慢调整

$$\mathrm{d}w = \alpha * x * \mathrm{d}y$$

同理，对于另一个算子：

$$y = w * x^2$$

我们可以计算得到：

$$\mathrm{d}w = \alpha * x^2 * \mathrm{d}y$$

下面我们看看自动微分是怎样在PyTorch中实现的，在探究之前，我们先关注几个问题：
- PyTorch中的计算图是怎样构建的？ - 反向传播的流程是什么样的？

计算图及反向传播

在计算图中，autograd会记录所有的操作，并生成一个DAG（有向无环图），其中输出的tensor是根节点，输入的te

在前向阶段，autograd同时做两件事： - 根据算子计算结果Tensor - 维护算子的梯度函数

在反向阶段，当.backward()被调用时，autograd: - 对于节点的每一个梯度函数，计算相应节点的梯度
- 在节点上对梯度进行累加，并保存到节点的.grad属性上 - 根据链式法则，按照同样的方式计算，一直到叶子节点

对于一个简单的例子：

```
import torch

a = torch.tensor([2., 3.], requires_grad=True)
b = torch.tensor([6., 4.], requires_grad=True)

Q = 3*a**3 - b**2
```

下图是对应的计算图，其中的函数代表梯度计算函数：

自动微分相关的核心数据结构

TensorImpl是Tensor的实现

at::Tensor：shared ptr 指向 TensorImpl

TensorImpl：对 at::Tensor 的实现

    [AutogradMetaInterface](c10::AutogradMetaInterface) autograd_meta_ tensor   variable

Variable: 就是Tensor，为了向前兼容保留的

using Variable = at::Tensor;

  , Variable   gradient , Tensor   gradient

Variable AutogradMeta [AutogradMetaInterface](c10::AutogradMetaInterface)   Variable

 version view

   AutogradMeta , autograd

```
// c10/core/TensorImpl.h

struct C10_API TensorImpl : public c10::intrusive_ptr_target {
  // ...
public:
  Storage storage_;

private:
  std::unique_ptr<c10::AutogradMetaInterface> autograd_meta_ = nullptr;

 protected:
  std::unique_ptr<c10::NamedTensorMetaInterface> named_tensor_meta_ = nullptr;

  c10::VariableVersion version_counter_;

  PyObject* pyobj_;
  c10::impl::SizesAndStrides sizes_and_strides_;

  int64_t storage_offset_ = 0;

  int64_t numel_ = 1;

  caffe2::TypeMeta data_type_;

  c10::optional<c10::Device> device_opt_;

  bool is_contiguous_ : 1;

  bool storage_access_should_throw_ : 1;

  bool is_channels_last_ : 1;

  bool is_channels_last_contiguous_ : 1;

  bool is_channels_last_3d_ : 1;

  bool is_channels_last_3d_contiguous_ : 1;

  bool is_non_overlapping_and_dense_ : 1;

  bool is_wrapped_number_ : 1;

  bool allow_tensor_metadata_change_ : 1;

  bool reserved_ : 1;

  uint8_t sizes_strides_policy_ : 2;

  DispatchKeySet key_set_;
}
```

autograd_meta_表示 Variable 中关于计算梯度的元数据信息，AutogradMetaInterface 是一个接口，有不同的子类，这里的 Variable 对象的梯度计算的元数据类型为 AutogradMeta，其部分成员为

```
// torch/csrc/autograd/variable.h

struct TORCH_API AutogradMeta : public c10::AutogradMetaInterface {
  std::string name_;

  Variable grad_;
  std::shared_ptr<Node> grad_fn_;
  std::weak_ptr<Node> grad_accumulator_;
  std::shared_ptr<ForwardGrad> fw_grad_;

  std::vector<std::shared_ptr<FunctionPreHook>> hooks_;
  std::shared_ptr<hooks_list> cpp_hooks_list_;

  bool requires_grad_;
  bool retains_grad_;
  bool is_view_;
  uint32_t output_nr_;

  // ...
}
```

grad_ 表示反向传播时，关于当前 Variable 的梯度值。grad_fn_ 是用于计算非叶子-Variable的梯度的函数，比如 AddBackward0对象用于计算result这个Variable 的梯度。对于叶子Variable，此字段为 None。grad_accumulator_ 用于累加叶子 Variable 的梯度累加器，比如 AccumulateGrad 对象用于累加 self的梯度。对于非叶 Variable，此字段为 None。output_nr_ 表示当前 Variable 是 计算操作的第一个输出，此值从 0 开始。

可以看到，grad_fn_和grad_accumulator_都是Node的指针，这是因为在计算图中，算子的C++类型是Node，不同的

Node是由上一级的Node创建的

```
// torch/include/torch/csrc/autograd/function.h

struct TORCH_API Node : std::enable_shared_from_this<Node> {
 public:
  /// Construct a new `Node` with the given `next_edges`
  // NOLINTNEXTLINE(cppcoreguidelines-pro-type-member-init)
  explicit Node(
      uint64_t sequence_nr,
      edge_list&& next_edges = edge_list())
      : sequence_nr_(sequence_nr),
      next_edges_(std::move(next_edges)) {

    for (const Edge& edge: next_edges_) {
      update_topological_nr(edge);
    }

    if (AnomalyMode::is_enabled()) {
      metadata()->store_stack();

      assign_parent();
    }

    // Store the thread_id of the forward operator.
    // See NOTE [ Sequence Numbers ]
    thread_id_ = at::RecordFunction::currentThreadId();
  }



  /// Evaluates the function on the given inputs and returns the result of the
  /// function call.
  variable_list operator()(variable_list&& inputs) {
    // ...
    return apply(std::move(inputs));
  }

  uint32_t add_input_metadata(const at::Tensor& t) noexcept {
    // ...
  }

  void add_next_edge(Edge edge) {
    update_topological_nr(edge);
    next_edges_.push_back(std::move(edge));
  }

 protected:
  /// Performs the `Node`'s actual operation.
  virtual variable_list apply(variable_list&& inputs) = 0;

  variable_list traced_apply(variable_list inputs);


  const uint64_t sequence_nr_;

  uint64_t topological_nr_ = 0;


  mutable bool has_parent_ = false;


  uint64_t thread_id_ = 0;


  std::mutex mutex_;

  edge_list next_edges_;

  PyObject* pyobj_ = nullptr;
  std::unique_ptr<AnomalyMetadata> anomaly_metadata_ = nullptr;

  std::vector<std::unique_ptr<FunctionPreHook>> pre_hooks_;

  std::vector<std::unique_ptr<FunctionPostHook>> post_hooks_;

  at::SmallVector<InputMetadata, 2> input_metadata_;
};
```

AutoGradMeta

AutoGradMeta :   Variable autograd

   grad_ Variable   AutoGradMeta   var tensor

    Node   grad_fn var graph   grad_accumulator var   ,   grad_

  output_nr   var grad_fn

       Edge gradient_edge, gradient_edge.function   grad_fn,   gradient_edge.input_nr   gra

Edge

autograd::Edge: 指向autograd::Node的一个输入

    Node   edge Node

  input_nr   edge Node

Node

autograd::Node: 对应AutoGrad Graph中的Op

  autograd op   apply

    next_edges_

    input_metadata_   tensor metadata

           op

Node in AutoGrad Graph

    Variable Edge Node

Edge Var

call operator

next_edge

Node

Node next_edge(index)/next_edges()

add_next_edge()

前向计算

PyTorch通过tracing只生成了后向AutoGrad Graph.

代码是生成的，需要编译才能看到对应的生成结果

gen_variable_type.py op

pytorch/torch/csrc/autograd/generated/

tracing

relu pytorch/torch/csrc/autograd/generated/VariableType_0.cpp

grad_fn trace op .

后向计算

autograd::backward():计算output var的梯度值，调用的 run_backward()

autograd::grad() ：计算有output var和到特定input的梯度值，调用的 run_backward()

autograd::run_backward()• g’f

output var grad_fn roots

input var grad_fn output_edges,

autograd::Engine::get_default_engine().execute(...)

autograd::Engine::execute(…)

GraphTask

GraphRoot Node roots Node apply() roots grad

compute_dependencies(...)

GraphRoot grad_fn grad_fn GraphTask

GraphTask input var

GraphTask

CPU or GPU

CPU autograd::Engine::thread_main(...)

autograd::Engine::thread_main(…)

evaluate_function(...)

call_function(...) , Node

grad Tensor grad tensor grad_fn grad_fn backward backward

Topic .

参考

- https://blog.csdn.net/zandaoguang/article/details/115713552
- https://zhuanlan.zhihu.com/p/111239415
- https://zhuanlan.zhihu.com/p/138203371

数据加载

主要内容

数据的加载主要包括以下几个方面： - 数据集的格式转换，需要支持各种类型各种格式的数据，如图片、语音、文
- 数据的采样和shuffle，可能面临分布式的挑战。 - 数据增强，会产生额外的数据
- 数据预处理，如图片事先进行黑白二值化等 - 数据分batch - 数据加载到内存，并且进入锁页内存
- 数据加载到GPU - 数据分发给不同的计算单元，并且不会重复，且支持分布式训练

数据加载的设计

下面我们先看一个利用CIFAR10数据集进行模型训练的例子：

```
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])

trainset = torchvision.datasets.CIFAR10(
    root='./data', train=True, download=True, transform=transform_train)
trainloader = torch.utils.data.DataLoader(
    trainset, batch_size=128, shuffle=True, num_workers=2)

# Model
print('==> Building model..')
net = SENet18()
net = net.to(device)
if device == 'cuda':
    net = torch.nn.DataParallel(net)
    cudnn.benchmark = True

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=args.lr,
                      momentum=0.9, weight_decay=5e-4)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=200)

# Training
def train(epoch):
    print('\nEpoch: %d' % epoch)
    net.train()
    train_loss = 0
    correct = 0
    total = 0
    for batch_idx, (inputs, targets) in enumerate(trainloader):
        inputs, targets = inputs.to(device), targets.to(device)
        optimizer.zero_grad()
        outputs = net(inputs)
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()


for epoch in range(start_epoch, start_epoch+200):
    train(epoch)
    scheduler.step()
```

在这个例子中，训练使用的是torch.utils.data.DataLoader，我们先从DataLoader入手，看看PyTorch是如何管理数

并行数据读取

当前业界普遍使用GPU进行模型训练，GPU的吞吐率很高，很容易导致数据的加载成为瓶颈。因此PyTorch的DataLoad

支持锁页内存

出于安全性的考虑，现代操作系统为每个进程提供了独立的虚拟地址空间，虚拟地址空间和物理内存的地址是通过内

在关键的应用场景，或者高性能计算的应用中，为了避免内存数据被交换到磁盘上，可以使用操作系统提供的能力，
memory）。

在深度学习模型训练过程中，因为数据集所占的内存比较多，又需要被频繁访问，因此一个比较好的加速方法就是将

使用锁页内存的另一个好处是主机内存和GPU内存之间的数据传输，基于锁页内存传输数据可以避免一次临时的数据

数据加载的整体设计

相比算子实现来讲，数据加载可以算作是非常简单直接的实现了。如下是单进程下数据加载的运行时，_SingleProc

在多进程的情况下，最耗费时间的Fetcher部分和pin_memory()部分改成了多进程，如下图：

数据读取

在torch模块中，DataSet是所有数据集的基类，其中关键的方法是__getitem__（），因为关联的DataLoader依靠这

但是Pytorch将__getitem__()的实现下放到了其他模块中，原因在于不同类型的数据集差异很大，比如图像类数据集

```
# torch/utils/data/dataset.py

class Dataset(Generic[T_co]):
    def __getitem__(self, index) -> T_co:
        raise NotImplementedError

    def __add__(self, other: 'Dataset[T_co]') -> 'ConcatDataset[T_co]':
        return ConcatDataset([self, other])
```

在torchvision中，可以比较清楚的看到，CIFAR10的数据集继承了VisionDataset（VisionDataset继承了torch.uti

```python
# torchvision/datasets/cifar.py

class CIFAR10(VisionDataset):

     def __init__(
            self,
            root: str,
            train: bool = True,
            transform: Optional[Callable] = None,
            target_transform: Optional[Callable] = None,
            download: bool = False,
    ) -> None:

        super(CIFAR10, self).__init__(root, transform=transform,
                                      target_transform=target_transform)
        #...
        self.data: Any = []
        self.targets = []

        # now load the picked numpy arrays
        for file_name, checksum in downloaded_list:
            file_path = os.path.join(self.root, self.base_folder, file_name)
            with open(file_path, 'rb') as f:
                entry = pickle.load(f, encoding='latin1')
                self.data.append(entry['data'])
                if 'labels' in entry:
                    self.targets.extend(entry['labels'])
                else:
                    self.targets.extend(entry['fine_labels'])

        self.data = np.vstack(self.data).reshape(-1, 3, 32, 32)
        self.data = self.data.transpose((0, 2, 3, 1))  # convert to HWC

        self._load_meta()

    def _load_meta(self) -> None:
        path = os.path.join(self.root, self.base_folder, self.meta['filename'])
        if not check_integrity(path, self.meta['md5']):
            raise RuntimeError('Dataset metadata file not found or corrupted.' +
                               ' You can use download=True to download it')
        with open(path, 'rb') as infile:
            data = pickle.load(infile, encoding='latin1')
            self.classes = data[self.meta['key']]
        self.class_to_idx = {_class: i for i, _class in enumerate(self.classes)}
    def __getitem__(self, index: int) -> Tuple[Any, Any]:
        img, target = self.data[index], self.targets[index]

        img = Image.fromarray(img)

        if self.transform is not None:
            img = self.transform(img)

        if self.target_transform is not None:
            target = self.target_transform(target)

        return img, target
```

从上面的实现中可以看到，CIFAR10的数据集在初始化的时候就把所有的图片都读取并做了初始的处理，考虑到某些

我们会有个疑问，所有的worker使用同一个Dataset吗？getitem()会成为瓶颈么？

数据采样

我们在训练模型的时候，一般是把DataLoader当作迭代器来使用，缺省情况下DataLoader只使用一个进程来读取数
称为_SingleProcessDataLoaderIter，但是当计算速度比较快，比如使用GPU或者多卡进行训练时，为了加快数据加
以设置DataLoader使用多进程进行读取，此时DataLoader返回的迭代器称为_MultiProcessingDataLoaderIter。

```
#Harry  torch/utils/data/dataloader.py

class DataLoader(Generic[T_co]):
    dataset: Dataset[T_co]
    batch_size: Optional[int]
    num_workers: int
    pin_memory: bool
    drop_last: bool
    timeout: float
    sampler: Union[Sampler, Iterable]
    pin_memory_device: str
    prefetch_factor: int
    _iterator : Optional['_BaseDataLoaderIter']
    __initialized = False

    def _get_iterator(self) -> '_BaseDataLoaderIter':
        if self.num_workers == 0:
            return _SingleProcessDataLoaderIter(self)
        else:
            self.check_worker_number_rationality()
            return _MultiProcessingDataLoaderIter(self)
```

由于要协调进程间数据的读取，_MultiProcessingDataLoaderIter的实现略微复杂一些。首先，在初始化的时候，就
multiprocessing库创建多个子进程，每个子进程都在执行_worker_loop()函数。

```python
# torch/utils/data/dataloader.py

class _MultiProcessingDataLoaderIter(_BaseDataLoaderIter):
    def __init__(self, loader):
        #...
        for i in range(self._num_workers):
            index_queue = multiprocessing_context.Queue()
            index_queue.cancel_join_thread()
            w = multiprocessing_context.Process(
                target=_utils.worker._worker_loop,
                args=(self._dataset_kind, self._dataset, index_queue,
                      self._worker_result_queue, self._workers_done_event,
                      self._auto_collation, self._collate_fn, self._drop_last,
                      self._base_seed, self._worker_init_fn, i, self._num_workers,
                      self._persistent_workers, self._shared_seed))
            w.daemon = True
            w.start()
            self._index_queues.append(index_queue)
            self._workers.append(w)
        #...
```

在多进程中环境中，不能使用Python标准库中的Queue。需要使用进程安全的multiprocessing.Queue，和其他语言的

进程安全的Queue是_MultiProcessDataLoaderIter中主进程及各个worker子进程之间传递消息的通道，包括以下几种
- index_queue。存放数据为(send_idx, index)，由main_thread生产，worker_1~n_process消费。其中send_idx是
- worker_result_queue。存放数据为(send_idx, pageble tensor)，由worker_1~n_process产生，pin_memory_threa
- data_queue。存放数据为(send_idx, pinned tensor)，由- pin_memory_thread产生，main_thread消费。

对应于

```python
# torch/utils/data/_utils/worker.py

def _worker_loop(dataset_kind, dataset, index_queue, data_queue, done_event,
                 auto_collation, collate_fn, drop_last, base_seed, init_fn, worker_id,
                 num_workers, persistent_workers, shared_seed):

        #...
        global _worker_info
        _worker_info = WorkerInfo(id=worker_id, num_workers=num_workers,
                                  seed=seed, dataset=dataset)

        #...
        fetcher = _DatasetKind.create_fetcher(dataset_kind, dataset, auto_collation, collate
        #...
        while watchdog.is_alive():
            r = index_queue.get(timeout=MP_STATUS_CHECK_INTERVAL)

            if isinstance(r, _ResumeIteration):
                #...
                fetcher = _DatasetKind.create_fetcher(
                    dataset_kind, dataset, auto_collation, collate_fn, drop_last)
                continue
            #...
            idx, index = r
            data: Union[_IterableDatasetStopIteration, ExceptionWrapper]

            #...
            data = fetcher.fetch(index)
            data_queue.put((idx, data))
            del data, idx, index, r  # save memory
            #...
```

这里简单介绍一下fetcher，fetcher的工作就是从Dataset中读取数据，根据Dataset的类型（Map类型或者Iterable

从上面代码可以看出worker的工作流程也比较简单，先根据Dataset类型创建相应的fetcher，然后启动循环，从ind

值得注意的是，当读到末尾的时候，worker会根据drop_last参数决定是否要丢弃最后这一部分数据，同时如果设置

数据预处理及数据增强

在Dataset的定义中，本身是没有transform参数的，但是我们平时在使用具体的Dataset时，一般都有transform这个

```
# torchvision/datasets/folder.py

class DatasetFolder(VisionDataset):
    def __getitem__(self, index: int) -> Tuple[Any, Any]:
        path, target = self.samples[index]
        sample = self.loader(path)
        if self.transform is not None:
            sample = self.transform(sample)
        if self.target_transform is not None:
            target = self.target_transform(target)

        return sample, target
```

锁页内存

到了这里，原始的文件中的数据已经被读取，经过变换后，放到了DataLoader的data_queue里，

```
# torch/utils/data/dataloader.py

class _MultiProcessingDataLoaderIter(_BaseDataLoaderIter):
    def __init__(self, loader):
        #...

       if self._pin_memory:
            self._pin_memory_thread_done_event = threading.Event()

            # Queue is not type-annotated
            self._data_queue = queue.Queue()  # type: ignore[var-annotated]
            pin_memory_thread = threading.Thread(
                target=_utils.pin_memory._pin_memory_loop,
                args=(self._worker_result_queue, self._data_queue,
                      torch.cuda.current_device(),
                      self._pin_memory_thread_done_event, self._pin_memory_device))
            pin_memory_thread.daemon = True
            pin_memory_thread.start()
            # Similar to workers (see comment above), we only register
            # pin_memory_thread once it is started.
            self._pin_memory_thread = pin_memory_thread
        else:
            self._data_queue = self._worker_result_queue
```

可以看到，DataLoader只启动了一个pin_memory的线程，这个线程的工作相当简单，就是将_data_queue中的样本数

TODO: Tensor的pin_memory方法以后有机会可以再看一下。

数据加载到GPU

数据分发

DistributedSampler

torch/utils/dataset.py

模型训练中的数据集

设计原则1. DataLoader -> Dataset

参考

- 万字综述，核心开发者全面解读PyTorch内部机制 https://zhuanlan.zhihu.com/p/67834038
- https://blog.csdn.net/u013608424/article/details/123782284 # 第9章
  优化器

分布式

本章主要内容

- 为什么需要分布式
- 分布式的难点在哪里？
- PyTorch中的相关模块
  - THD
  - C10D
  - torch.multiprocessing
  - torch.distributedDataParallel（DP）
  - DistributedDataParallel（DDP）
  - torch.distributed.rpc

什么是分布式训练

分布式计算

由于单个节点的计算能力有限，对于计算密集型的任务，只在单个节点上运行，可能会花费非常多的时间，此时充分
将任务从单节点转化为分布式任务，需要考虑不同节点间的通信，包括输入数据的拆分，临时数据的分发与归并，计

为了简化算法开发的复杂度，将分布式计算中的数据分发和网络通信与具体的算法应用分开，先驱们开发了不同的分

在深度学习领域，模型的效果主要来自于两个方面：海量的数据和精心设计的复杂网络结构，这两点使得深度学习模

来源：Compute Trends Across Three Eras of Machine Learning

深度学习模型分布式训练的进展

PyTorch中的分布式训练

参考

- https://zhuanlan.zhihu.com/p/136372142

第11章 JIT

TorchScript

为什么需要JIT

- 性能

实现JIT的挑战

- 动态图中的条件逻辑

一个简单的例子

为了说明JIT是如何工作的，我们看一个简单的例子：

```python
@torch.jit.script
def foo(len):
    # type: (int) -> torch.Tensor
    rv = torch.zeros(3, 4)
    for i in range(len):
        if i < 10:
            rv = rv - 1.0
        else:
            rv = rv + 1.0
    return rv

print(foo.code)
```

加上修饰器后，上面的函数foo的类型变成了， 并且其代码被重新编译成了下面的形式：

```
def foo(len: int) -> Tensor:
  rv = torch.zeros([3, 4], dtype=None, layout=None, device=None, pin_memory=None)
  rv0 = rv
  for i in range(len):
    if torch.lt(i, 10):
      rv1 = torch.sub(rv0, 1., 1)
    else:
      rv1 = torch.add(rv0, 1., 1)
    rv0 = rv1
  return rv0
```

可见其中基本的条件语句被转换成了torch的函数，但这仍然是Python代码层面，在执行层，TorchScript使用的是
single assignment (SSA) intermediate representation (IR)），其中的指令包括
ATen (the C++ backend of PyTorch) 算子及其他一些原语，比如条件控制和循环控制的原语。

如果我们打印print(foo.graph)，可以看到如下的输出，其中“:5:4” 这样的注释代表中间代码所对应的Python源
Notebook，读者朋友可以忽略文件名，只关注代码位置即可。

```
graph(%len.1 : int):
  %20 : int = prim::Constant[value=1]()
  %13 : bool = prim::Constant[value=1]() # <ipython-input-4-01a58e79a588>:5:4
  %5 : None = prim::Constant()
  %1 : int = prim::Constant[value=3]() # <ipython-input-4-01a58e79a588>:4:21
  %2 : int = prim::Constant[value=4]() # <ipython-input-4-01a58e79a588>:4:24
  %16 : int = prim::Constant[value=10]() # <ipython-input-4-01a58e79a588>:6:15
  %19 : float = prim::Constant[value=1]() # <ipython-input-4-01a58e79a588>:7:22
  %4 : int[] = prim::ListConstruct(%1, %2)
  %rv.1 : Tensor = aten::zeros(%4, %5, %5, %5, %5) # <ipython-input-4-01a58e79a588>:4:9
  %rv : Tensor = prim::Loop(%len.1, %13, %rv.1) # <ipython-input-4-01a58e79a588>:5:4
    block0(%i.1 : int, %rv.14 : Tensor):
      %17 : bool = aten::lt(%i.1, %16) # <ipython-input-4-01a58e79a588>:6:11
      %rv.13 : Tensor = prim::If(%17) # <ipython-input-4-01a58e79a588>:6:8
        block0():
          %rv.3 : Tensor = aten::sub(%rv.14, %19, %20) # <ipython-input-4-01a58e79a588>:7:1
          -> (%rv.3)
        block1():
          %rv.6 : Tensor = aten::add(%rv.14, %19, %20) # <ipython-input-4-01a58e79a588>:9:1
          -> (%rv.6)
      -> (%13, %rv.13)
  return (%rv)
```

JIT trace的实现

```
def fill_row_zero(x):
    x[0] = torch.rand(*x.shape[1:2])
    return x

traced = torch.jit.trace(fill_row_zero, (torch.rand(3, 4),))
print(traced.graph)
```

Trace的实现在这里（不同版本的实现位置可能不一样）：

```
# torch/jit/_trace.py

def trace(
    func,
    example_inputs,
    optimize=None,
    check_trace=True,
    check_inputs=None,
    check_tolerance=1e-5,
    strict=True,
    _force_outplace=False,
    _module_class=None,
    _compilation_unit=_python_cu,
):

    #YL      Module   trace_module

    var_lookup_fn = _create_interpreter_name_lookup_fn(0)

    name = _qualified_name(func)
    traced = torch._C._create_function_from_trace(
        name,
        func,
        example_inputs,
        var_lookup_fn,
        strict,
        _force_outplace,
        get_callable_argument_names(func)
    )

    # Check the trace against new traces created from user-specified inputs

    return traced
```

_C是torch的C++模块，因此该调用转到了C++部分，在初始化的时候，_create_function_from_trace被注册到了tor

```
//YL torch/csrc/jit/python/script_init.cpp

  m.def(
      "_create_function_from_trace",
      [](const std::string& qualname,
         const py::function& func,
         const py::tuple& input_tuple,
         const py::function& var_name_lookup_fn,
         bool strict,
         bool force_outplace,
         const std::vector<std::string>& argument_names) {
        auto typed_inputs = toTraceableStack(input_tuple);
        std::shared_ptr<Graph> graph = std::get<0>(tracer::createGraphByTracing(
            func,
            typed_inputs,
            var_name_lookup_fn,
            strict,
            force_outplace,
            /*self=*/nullptr,
            argument_names));
        auto cu = get_python_cu();
        auto name = c10::QualifiedName(qualname);
        auto result = cu->create_function(
            std::move(name), std::move(graph), /*shouldMangle=*/true);
        StrongFunctionPtr ret(std::move(cu), result);
        didFinishEmitFunction(ret);
        return ret;
      },
      py::arg("name"),
      py::arg("func"),
      py::arg("input_tuple"),
      py::arg("var_name_lookup_fn"),
      py::arg("strict"),
      py::arg("force_outplace"),
      py::arg("argument_names") = std::vector<std::string>());
```

可以看到，主要的工作是构造一个Graph，并且是由tracer::createGraphByTracing()完成的。

参考

- https://pytorch.org/docs/stable/jit.html
- https://zhuanlan.zhihu.com/p/410507557

第3章 自动微分

Index

梯度的初步认识

我们知道，深度神经网络的训练时依赖于梯度的反向传播的，因此在深度学习框架的设计上就涉及到几个问题：
- 梯度保存在哪里？ - 梯度是怎样计算的？ - 神经网络的参数是如何更新的？ -
如何实现反向传播？

神经网络的核心数据结构是Tensor，对于需要优化的Tensor，每次更新，都会有一个对应的梯度。因此最合适的方

在初始化Tensor的时候，可以指定一个参数requires_grad，代表这个Tensor是否需要计算梯度。

在涉及复杂的神经网络之前，我们先看一个非常简单的计算，这个例子来自于pytorch官方文档。

```
import torch

x = torch.ones(2, 2, requires_grad=True)
print(x)
```

输出结果为：

```
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
```

如果对这个Tensor做一些操作：

```
y = x + 2
print(y)
```

输出为：

```
tensor([[3., 3.],
        [3., 3.]], grad_fn=<AddBackward0>)
```

可以看到基于加法操作的Tensor y，被附加了一个grad_fn的函数。因为x是需要梯度的，而y是基于x的加法操作得到

同理做更多的操作：

```
z = y * y * 3
out = z.mean()

print(z, out)
```

输出如下，可见计算梯度的函数不是固定的，不同的操作对应不同的梯度计算函数。

```
tensor([[27., 27.],
        [27., 27.]], grad_fn=<MulBackward0>)
tensor(27., grad_fn=<MeanBackward0>)
```

现在我们再看一下梯度的计算和反向传播过程，刚才提到梯度是保存在Tensor里的，在pytorch中，可以通过Tensor

```
out.backward()

print(x.grad)
```

输出：

```
tensor([[4.5000, 4.5000],
        [4.5000, 4.5000]])
```

关于梯度的基本理论

雅克比矩阵

一元Tensor的梯度计算，不需要雅克比矩阵

待补充

PyTorch中梯度的计算过程

从刚才的例子可以看到，梯度可以通过Tensor.backward()函数计算得到。那么这个函数都做了什么呢？

```python
class Tensor(torch._C._TensorBase):
    def backward(self, gradient=None, retain_graph=None, create_graph=False, inputs=None):

        if has_torch_function_unary(self):
            return handle_torch_function(
                Tensor.backward,
                (self,),
                self,
                gradient=gradient,
                retain_graph=retain_graph,
                create_graph=create_graph,
                inputs=inputs)
        torch.autograd.backward(self, gradient, retain_graph, create_graph, inputs=inputs)
```

我们先忽略对一元情况的处理，一般来说，最终会调用autograd.backward()函数进行梯度的计算，这个函数定义在

这个函数在计算梯度并且反向传播的时候，会把梯度保存在计算图的叶子节点中。需要注意的是，在调用backward前

```python
def backward(
    tensors: _TensorOrTensors,
    grad_tensors: Optional[_TensorOrTensors] = None,
    retain_graph: Optional[bool] = None,
    create_graph: bool = False,
    grad_variables: Optional[_TensorOrTensors] = None,
    inputs: Optional[_TensorOrTensors] = None,
) -> None:
    if grad_variables is not None:
        warnings.warn("'grad_variables' is deprecated. Use 'grad_tensors' instead.")
        if grad_tensors is None:
            grad_tensors = grad_variables
        else:
            raise RuntimeError("'grad_tensors' and 'grad_variables' (deprecated) "
                               "arguments both passed to backward(). Please only "
                               "use 'grad_tensors'.")
    if inputs is not None and len(inputs) == 0:
        raise RuntimeError("'inputs' argument to backward() cannot be empty.")

    tensors = (tensors,) if isinstance(tensors, torch.Tensor) else tuple(tensors)
    inputs = (inputs,) if isinstance(inputs, torch.Tensor) else \
        tuple(inputs) if inputs is not None else tuple()

    grad_tensors_ = _tensor_or_tensors_to_tuple(grad_tensors, len(tensors))
    grad_tensors_ = _make_grads(tensors, grad_tensors_, is_grads_batched=False)
    if retain_graph is None:
        retain_graph = create_graph

    # The reason we repeat same the comment below is that
    # some Python versions print out the first line of a multi-line function
    # calls in the traceback and some print out the last line
    Variable._execution_engine.run_backward(  # Calls into the C++ engine to run the backwa
        tensors, grad_tensors_, retain_graph, create_graph, inputs,
        allow_unreachable=True, accumulate_grad=True)  # Calls into the C++ engine to run th
```

在经过一些处理之后，最后调用的是Variable._execution_engine.run_backwar()函数，但事实上，Variable._exe

```
import torch
from torch._six import with_metaclass


class VariableMeta(type):
    def __instancecheck__(cls, other):
        return isinstance(other, torch.Tensor)

# mypy doesn't understand torch._six.with_metaclass
class Variable(with_metaclass(VariableMeta, torch._C._LegacyVariableBase)):  # type: ignore
    pass

from torch._C import _ImperativeEngine as ImperativeEngine
Variable._execution_engine = ImperativeEngine()
```

在对应的C++代码中，使用PyModule_AddObject注册了_ImperativeEngine这个类对象。
torch/csrc/autograd/python_engine.cpp

```
PyTypeObject THPEngineType = {
    PyVarObject_HEAD_INIT(nullptr, 0) "torch._C._EngineBase", /* tp_name */
    sizeof(THPEngine), /* tp_basicsize */
    0, /* tp_itemsize */
    nullptr, /* tp_dealloc */
    0, /* tp_vectorcall_offset */
    nullptr, /* tp_getattr */
    nullptr, /* tp_setattr */
    nullptr, /* tp_reserved */
    nullptr, /* tp_repr */
    nullptr, /* tp_as_number */
    nullptr, /* tp_as_sequence */
    nullptr, /* tp_as_mapping */
    nullptr, /* tp_hash  */
    nullptr, /* tp_call */
    nullptr, /* tp_str */
    nullptr, /* tp_getattro */
    nullptr, /* tp_setattro */
    nullptr, /* tp_as_buffer */
    Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE, /* tp_flags */
    nullptr, /* tp_doc */
    nullptr, /* tp_traverse */
    nullptr, /* tp_clear */
    nullptr, /* tp_richcompare */
    0, /* tp_weaklistoffset */
    nullptr, /* tp_iter */
    nullptr, /* tp_iternext */
    THPEngine_methods, /* tp_methods */
    nullptr, /* tp_members */
    nullptr, /* tp_getset */
    nullptr, /* tp_base */
    nullptr, /* tp_dict */
    nullptr, /* tp_descr_get */
    nullptr, /* tp_descr_set */
    0, /* tp_dictoffset */
    nullptr, /* tp_init */
    nullptr, /* tp_alloc */
    THPEngine_new /* tp_new */
};

bool THPEngine_initModule(PyObject* module) {
#ifndef _WIN32
  if (pthread_atfork(nullptr, nullptr, child_atfork) != 0) {
    throw std::runtime_error("unable to set pthread_atfork handler");
  }
#endif
  if (PyType_Ready(&THPEngineType) < 0)
    return false;
  Py_INCREF(&THPEngineType);
  PyModule_AddObject(module, "_ImperativeEngine", (PyObject*)&THPEngineType);
  set_default_engine_stub(python::PythonEngine::get_python_engine);
  return true;
}
```

希望了解PyModule_AddObject细节的同学可以学习一下Cython。在这里我们只需要知道这个函数可以将C++的类型注

可以看到，实际注册的对象是一个PyTypeObject。PyTypeObject是Python中非常重要的一种类型，PyTypeObject就是

参考 https://blog.csdn.net/zhangyifei216/article/details/50581787

对象中每个字段的含义可以从注释中看出来，不过基本可以忽略，大部分都是空，最后一个字段是THPEngine_new，

有一点待确认，就是PyTypeObject各个字段的定义，在不同Python版本中估计是不一样的，如何保证兼容呢？至少参

对于_ImperativeEngine这个类，在C++中注册了以下几个函数，其中就包括run_backward函数，对应的C++实现是TH

```
// NOLINTNEXTLINE(cppcoreguidelines-avoid-c-arrays,modernize-avoid-c-arrays,cppcoreguidelin
static struct PyMethodDef THPEngine_methods[] = {
    {(char*)"run_backward",
     castPyCFunctionWithKeywords(THPEngine_run_backward),
     METH_VARARGS | METH_KEYWORDS,
     nullptr},
    {(char*)"queue_callback", THPEngine_queue_callback, METH_O, nullptr},
    {(char*)"is_checkpoint_valid",
     THPEngine_is_checkpoint_valid,
     METH_NOARGS,
     nullptr},
    {nullptr}};
```

THPEngine_run_backward函数的实现相对比较复杂，但是其中开始部分是对输入参数进行解析，在结束部分是对Ten

```
// Implementation of torch._C._EngineBase.run_backward
PyObject* THPEngine_run_backward(
    PyObject* self,
    PyObject* args,
    PyObject* kwargs) {

  HANDLE_TH_ERRORS
  PyObject* tensors = nullptr;
  PyObject* grad_tensors = nullptr;
  unsigned char keep_graph = 0;
  unsigned char create_graph = 0;
  PyObject* inputs = nullptr;
  unsigned char allow_unreachable = 0;
  unsigned char accumulate_grad =
      0; // Indicate whether to accumulate grad into leaf Tensors or capture
  const char* accepted_kwargs[] = {// NOLINT
                                   "tensors",
                                   "grad_tensors",
                                   "keep_graph",
                                   "create_graph",
                                   "inputs",
                                   "allow_unreachable",
                                   "accumulate_grad",
                                   nullptr};

    if (!PyArg_ParseTupleAndKeywords(
          args,
          kwargs,
          "OObb|Obb",
          (char**)accepted_kwargs,
          &tensors,
          &grad_tensors,
          &keep_graph,
          &create_graph,
          &inputs,
          &allow_unreachable,
          &accumulate_grad))
    return nullptr;

  // ... check arguments

  // ... init edges

  variable_list outputs;
  {
    pybind11::gil_scoped_release no_gil;
    auto& engine = python::PythonEngine::get_python_engine();
    outputs = engine.execute(
        roots, grads, keep_graph, create_graph, accumulate_grad, output_edges);
  }

  // ... assign gradients to Tensor

}
```

在执行run_backward()函数时，首先通过PyArg_ParseTupleAndKeywords()函数对入参进行格式解析，将Python的对

可以看到，计算梯度的核心函数是engine.execute()，PythonEngine继承自Engine，实现execute()的时候也是简单

下面的代码来自于torch/csrc/autograd/python_engine.h 和torch/csrc/autograd/python_engine.cpp。

```
struct PythonEngine : public Engine {
  static Engine& get_python_engine();
  ~PythonEngine() override;
  void thread_init(
      int device,
      const std::shared_ptr<ReadyQueue>& ready_queue,
      bool should_increment) override;
  void thread_on_exception(
      std::shared_ptr<GraphTask> graph_task,
      const std::shared_ptr<Node>& fn,
      std::exception& e) override;
  variable_list execute(
      const edge_list& roots,
      const variable_list& inputs,
      bool keep_graph,
      bool create_graph,
      bool accumulate_grad,
      const edge_list& outputs = {}) override;

  c10::intrusive_ptr<at::ivalue::Future> execute_with_graph_task(
      const std::shared_ptr<GraphTask>& graph_task,
      std::shared_ptr<Node> graph_root,
      InputBuffer&& input_buffer) override;

  std::unique_ptr<AnomalyMetadata> make_anomaly_metadata() override;
  std::unique_ptr<SavedVariableHooks> get_default_saved_variable_hooks()
      override;

 private:
  PythonEngine();
};

Engine& PythonEngine::get_python_engine() {
  static PythonEngine engine;
  // This is "probably" thread-safe because the flag is set in a fork handler
  // before any threads are created, and this function is only called with the
  // GIL held. However, using fork + threads is playing with fire so this is
  // more of a "best effort" thing. For example, if the fork occurs while the
  // backwards threads hold a lock, we'll probably deadlock in the engine
  // destructor.
  if (_reinitialize_engine) {
    engine.release_workers();
    engine.~PythonEngine();
    new (&engine) torch::autograd::python::PythonEngine();
    _reinitialize_engine = false;
  }
  return engine;
}

variable_list PythonEngine::execute(
    const edge_list& roots,
    const variable_list& inputs,
    bool keep_graph,
    bool create_graph,
    bool accumulate_grad,
    const edge_list& outputs) {
  TORCH_CHECK(
      !PyGILState_Check(),
      "The autograd engine was called while holding the GIL. If you are using the C++ "
      "API, the autograd engine is an expensive operation that does not require the "
      "GIL to be held so you should release it with 'pybind11::gil_scoped_release no_gil;'"
      ". If you are not using the C++ API, please report a bug to the pytorch team.")
  try {
    return Engine::execute(
        roots, inputs, keep_graph, create_graph, accumulate_grad, outputs);
  } catch (python_error& e) {
    e.restore();
    throw;
  }
}
```

Engine的定义和实现分别在torch/csrc/autograd/engine.h和torch/csrc/autograd/engine.cpp中。

在一个平台级的系统里，能够被命名为Engine的类型，一定是整个系统的核心，而
Engine.execute()函数的实现肯定是这个核心对象的主要执行逻辑，在深度学习框架中，这个最主要的执行逻辑就是

```
auto Engine::execute(
    const edge_list& roots,
    const variable_list& inputs,
    bool keep_graph,
    bool create_graph,
    bool accumulate_grad,
    const edge_list& outputs) -> variable_list {
  // NOLINTNEXTLINE(cppcoreguidelines-pro-type-const-cast)
  validate_outputs(
      roots, const_cast<variable_list&>(inputs), [](const std::string& msg) {
        return msg;
      });
  if (accumulate_grad && create_graph) {
    TORCH_WARN_ONCE(
        "Using backward() with create_graph=True will create a reference cycle "
        "between the parameter and its gradient which can cause a memory leak. "
        "We recommend using autograd.grad when creating the graph to avoid this. "
        "If you have to use this function, make sure to reset the .grad fields of "
        "your parameters to None after use to break the cycle and avoid the leak.");
  }

  // accumulate_grad is true if and only if the frontend call was to
  // grad(), not backward(). grad() returns the sum of the gradients
  // w.r.t. the inputs and thus needs the inputs to be present.
  TORCH_CHECK_VALUE(
      accumulate_grad || !outputs.empty(), "grad requires non-empty inputs.");

  // A fresh first time Engine::execute call should start on the CPU device,
  // initialize a new thread local ready queue on CPU or reuse the existing one
  // (if there is one allocated already, i.e. consecutive backward calls,
  // re-entrant backward calls), then memoize the local_ready_queue in GraphTask
  init_local_ready_queue();
  bool not_reentrant_backward_call = worker_device == NO_DEVICE;

  auto graph_task = std::make_shared<GraphTask>(
      /* keep_graph */ keep_graph,
      /* create_graph */ create_graph,
      /* depth */ not_reentrant_backward_call ? 0 : total_depth + 1,
      /* cpu_ready_queue */ local_ready_queue);

  // If we receive a single root, skip creating extra root node
  bool skip_dummy_node = roots.size() == 1;
  auto graph_root = skip_dummy_node
      ? roots.at(0).function
      : std::make_shared<GraphRoot>(roots, inputs);

  auto min_topo_nr = compute_min_topological_nr(outputs);
  // Now compute the dependencies for all executable functions
  compute_dependencies(graph_root.get(), *graph_task, min_topo_nr);

  if (!outputs.empty()) {
    graph_task->init_to_execute(
        *graph_root, outputs, accumulate_grad, min_topo_nr);
  }

  // Queue the root
  if (skip_dummy_node) {
    InputBuffer input_buffer(roots.at(0).function->num_inputs());
    auto input = inputs.at(0);

    const auto input_stream = InputMetadata(input).stream();
    const auto opt_next_stream =
        roots.at(0).function->stream(c10::DeviceType::CUDA);
    input_buffer.add(
        roots.at(0).input_nr, std::move(input), input_stream, opt_next_stream);

    execute_with_graph_task(graph_task, graph_root, std::move(input_buffer));
  } else {
    execute_with_graph_task(
        graph_task, graph_root, InputBuffer(variable_list()));
  }
  // Avoid a refcount bump for the Future, since we check for refcount in
  // DistEngine (see TORCH_INTERNAL_ASSERT(futureGrads.use_count() == 1)
  // in dist_engine.cpp).
  auto& fut = graph_task->future_result_;
  fut->wait();
```

```
  graph_task->warning_handler_.replay_warnings();
  return fut->value().toTensorVector();
}
```

GraphTask在执行的过程中创建出来的。

明显能够看出，execute()方法中的重要步骤是execute_with_graph_task()函数。

执行的时候就是对graph_task进行BFS遍历，从root开始调用各Node的operator()重载函数。

```
c10::intrusive_ptr<at::ivalue::Future> Engine::execute_with_graph_task(
    const std::shared_ptr<GraphTask>& graph_task,
    std::shared_ptr<Node> graph_root,
    InputBuffer&& input_buffer) {
  initialize_device_threads_pool();
  // Lock mutex for GraphTask.
  std::unique_lock<std::mutex> lock(graph_task->mutex_);

  auto queue = ready_queue(graph_task->cpu_ready_queue_, input_buffer.device());

  // worker_device == NO_DEVICE it's a CPU thread and it's trying to drive the
  // autograd engine with corresponding GraphTask, and its NOT a re-entrant call
  if (worker_device == NO_DEVICE) {
    // We set the worker_device to CPU_DEVICE only if worker_device was
    // previously NO_DEVICE. Setting it to CPU afterwards allow us to detect
    // whether this is a re-entrant call or not.
    set_device(CPU_DEVICE);

    // set the graph_task owner to the current device
    graph_task->owner_ = worker_device;

    // Now that all the non-thread safe fields of the graph_task have been
    // populated, we can enqueue it.
    queue->push(
        NodeTask(graph_task, std::move(graph_root), std::move(input_buffer)));

    // The owning thread start to drive the engine execution for any CPU task
    // that was just pushed or will be added later from other worker threads
    lock.unlock();
    thread_main(graph_task);
    TORCH_INTERNAL_ASSERT(graph_task->future_result_->completed());
    // reset the worker_device after the completion of the graph_task, this is
    // so that the initial state of the engine remains the same across every
    // backward() or grad() call, we don't need to reset local_ready_queue as we
    // could possibly reuse it for new backward calls.
    worker_device = NO_DEVICE;
  } else {
    // If worker_device is any devices (i.e. CPU, CUDA): this is a re-entrant
  //    backward call from that device.
  graph_task->owner_ = worker_device;

  // Now that all the non-thread safe fields of the graph_task have been
  // populated, we can enqueue it.
  queue->push(
    NodeTask(graph_task, std::move(graph_root), std::move(input_buffer)));

  if (current_depth >= max_recursion_depth_) {
   // See Note [Reentrant backwards]
   // If reached the max depth, switch to a different thread
   add_thread_pool_task(graph_task);
  } else {
   // Total depth needs to be updated only in this codepath, since it is
   // not used in the block above (when we call add_thread_pool_task).
   // In the codepath above, GraphTask.reentrant_depth_ is used to
   // bootstrap total_depth in the other thread.
   ++total_depth;

   // Get back to work while we wait for our new graph_task to
   // complete!
   ++current_depth;
   lock.unlock();
   thread_main(graph_task);
   --current_depth;
   --total_depth;

   // The graph task should have completed and the associated future should
   // be marked completed as well since 'thread_main' above is a call
   // blocking an autograd engine thread.
   TORCH_INTERNAL_ASSERT(graph_task->future_result_->completed());
  }
 }
 // graph_task_exec_post_processing is done when the Future is marked as
 // completed in mark_as_completed_and_run_post_processing.
 return graph_task->future_result_;
}
```

这里涉及到几个逻辑： - 梯度的计算一般也是矩阵计算，对算力要求比较高，在有GPU的情况下可以使用GPU计算，
- 由于计算图是一个有向无环图，计算的时候有很多可以并行的节点，因此在设计上可以将任务推到队列中进行并行

从上面的代码可以看到，计算的核心是thread_main(graph_task)

```
auto Engine::thread_main(const std::shared_ptr<GraphTask>& graph_task) -> void {
 // When graph_task is nullptr, this is a long running thread that processes
 // tasks (ex: device threads). When graph_task is non-null (ex: reentrant
 // backwards, user thread), this function is expected to exit once that
```

```
  // graph_task complete.

#ifdef USE_ROCM
  // Keep track of backward pass for rocblas.
  at::ROCmBackwardPassGuard in_backward;
#endif

  // local_ready_queue should already been initialized when we get into
  // thread_main
  TORCH_INTERNAL_ASSERT(local_ready_queue != nullptr);
  while (graph_task == nullptr || !graph_task->future_result_->completed()) {
    // local_graph_task represents the graph_task we retrieve from the queue.
    // The outer graph_task represents the overall graph_task we need to execute
    // for reentrant execution.
    std::shared_ptr<GraphTask> local_graph_task;
    {
      // Scope this block of execution since NodeTask is not needed after this
      // block and can be deallocated (release any references to grad tensors
      // as part of inputs_).
      NodeTask task = local_ready_queue->pop();
      // This will only work if the worker is running a non backward task
      // TODO Needs to be fixed this to work in all cases
      if (task.isShutdownTask_) {
        C10_LOG_API_USAGE_ONCE("torch.autograd.thread_shutdown");
        break;
      }

      if (!(local_graph_task = task.base_.lock())) {
        // GraphTask for function is no longer valid, skipping further
        // execution.
        continue;
      }

      if (task.fn_ && !local_graph_task->has_error_.load()) {
        // Set the ThreadLocalState before calling the function.
        // NB: The ThreadLocalStateGuard doesn't set the grad_mode because
        // GraphTask always saves ThreadLocalState without grad_mode.
        at::ThreadLocalStateGuard tls_guard(local_graph_task->thread_locals_);
        c10::Warning::WarningHandlerGuard warnings_guard(
            &local_graph_task->warning_handler_);

        try {
          // The guard sets the thread_local current_graph_task on construction
          // and restores it on exit. The current_graph_task variable helps
          // queue_callback() to find the target GraphTask to append final
          // callbacks.
          GraphTaskGuard guard(local_graph_task);
          NodeGuard ndguard(task.fn_);
          {
            RECORD_FUNCTION(
                c10::str(
                    "autograd::engine::evaluate_function: ",
                    task.fn_.get()->name()),
                c10::ArrayRef<const c10::IValue>());
            evaluate_function(
                local_graph_task,
                task.fn_.get(),
                task.inputs_,
                local_graph_task->cpu_ready_queue_);
          }
        } catch (std::exception& e) {
          thread_on_exception(local_graph_task, task.fn_, e);
        }
      }
    }

    // Decrement the outstanding tasks.
    --local_graph_task->outstanding_tasks_;

    // Check if we've completed execution.
    if (local_graph_task->completed()) {
      local_graph_task->mark_as_completed_and_run_post_processing();

      auto base_owner = local_graph_task->owner_;
      // The current worker thread finish the graph_task, but the owning thread
      // of the graph_task might be sleeping on pop() if it does not have work.
      // So we need to send a dummy function task to the owning thread just to
      // ensure that it's not sleeping, so that we can exit the thread_main.
      // If it has work, it might see that graph_task->outstanding_tasks_ == 0
      // before it gets to the task, but it's a no-op anyway.
      //
      // NB: This is not necessary if the current thread is the owning thread.
      if (worker_device != base_owner) {
        // Synchronize outstanding_tasks_ with queue mutex
        std::atomic_thread_fence(std::memory_order_release);
        ready_queue_by_index(local_graph_task->cpu_ready_queue_, base_owner)
            ->push(NodeTask(local_graph_task, nullptr, InputBuffer(0)));
      }
    }
  }
}
```

thread_main()方法的最重要的步骤是调用evaluate_function().

```
void Engine::evaluate_function(
    std::shared_ptr<GraphTask>& graph_task,
    Node* func,
    InputBuffer& inputs,
    const std::shared_ptr<ReadyQueue>& cpu_ready_queue) {
  // The InputBuffer::adds that supplied incoming grads took pains to
  // ensure they're safe to consume in the context of the present
  // func's stream (if applicable). So we guard onto that stream
  // before working with the grads in any capacity.
  const auto opt_parent_stream = (*func).stream(c10::DeviceType::CUDA);
  c10::OptionalStreamGuard parent_stream_guard{opt_parent_stream};

  // If exec_info_ is not empty, we have to instrument the execution
  auto& exec_info_ = graph_task->exec_info_;
  if (!exec_info_.empty()) {
    auto& fn_info = exec_info_.at(func);
    if (auto* capture_vec = fn_info.captures_.get()) {
      // Lock mutex for writing to graph_task->captured_vars_.
      std::lock_guard<std::mutex> lock(graph_task->mutex_);
      for (const auto& capture : *capture_vec) {
        auto& captured_grad = graph_task->captured_vars_[capture.output_idx_];
        captured_grad = inputs[capture.input_idx_];
        for (auto& hook : capture.hooks_) {
          captured_grad = (*hook)(captured_grad);
        }
        if (opt_parent_stream) {
          // No need to take graph_task->mutex_ here, we already hold it
          graph_task->leaf_streams.emplace(*opt_parent_stream);
        }
      }
    }
    if (!fn_info.needed_) {
      // Skip execution if we don't need to execute the function.
      return;
    }
  }

  auto outputs = call_function(graph_task, func, inputs);

  auto& fn = *func;
  if (!graph_task->keep_graph_) {
    fn.release_variables();
  }
```

```
int num_outputs = outputs.size();
if (num_outputs == 0) { // Note: doesn't acquire the mutex
  // Records leaf stream (if applicable)
  // See Note [Streaming backwards]
  if (opt_parent_stream) {
    std::lock_guard<std::mutex> lock(graph_task->mutex_);
    graph_task->leaf_streams.emplace(*opt_parent_stream);
  }
  return;
}

if (AnomalyMode::is_enabled()) {
  AutoGradMode grad_mode(false);
  for (const auto i : c10::irange(num_outputs)) {
    auto& output = outputs[i];
    at::OptionalDeviceGuard guard(device_of(output));
    if (output.defined() && isnan(output).any().item<uint8_t>()) {
      std::stringstream ss;
      ss << "Function '" << fn.name() << "' returned nan values in its " << i
         << "th output.";
      throw std::runtime_error(ss.str());
    }
  }
}

// Lock mutex for the accesses to GraphTask dependencies_, not_ready_ and
// cpu_ready_queue_ below
std::lock_guard<std::mutex> lock(graph_task->mutex_);
for (const auto i : c10::irange(num_outputs)) {
  auto& output = outputs[i];
  const auto& next = fn.next_edge(i);

  if (!next.is_valid())
    continue;

  // Check if the next function is ready to be computed
  bool is_ready = false;
  auto& dependencies = graph_task->dependencies_;
  auto it = dependencies.find(next.function.get());

  if (it == dependencies.end()) {
    auto name = next.function->name();
    throw std::runtime_error(std::string("dependency not found for ") + name);
  } else if (--it->second == 0) {
    dependencies.erase(it);
    is_ready = true;
    }

    auto& not_ready = graph_task->not_ready_;
    auto not_ready_it = not_ready.find(next.function.get());
    if (not_ready_it == not_ready.end()) {
      // Skip functions that aren't supposed to be executed
      if (!exec_info_.empty()) {
        auto it = exec_info_.find(next.function.get());
        if (it == exec_info_.end() || !it->second.should_execute()) {
          continue;
        }
      }
      // No buffers have been allocated for the function
      InputBuffer input_buffer(next.function->num_inputs());

      // Accumulates into buffer
      const auto opt_next_stream = next.function->stream(c10::DeviceType::CUDA);
      input_buffer.add(
          next.input_nr, std::move(output), opt_parent_stream, opt_next_stream);

      if (is_ready) {
        auto queue = ready_queue(cpu_ready_queue, input_buffer.device());
        queue->push(
            NodeTask(graph_task, next.function, std::move(input_buffer)));
      } else {
        not_ready.emplace(next.function.get(), std::move(input_buffer));
      }
    } else {
      // The function already has a buffer
      auto& input_buffer = not_ready_it->second;

      // Accumulates into buffer
      const auto opt_next_stream = next.function->stream(c10::DeviceType::CUDA);
      input_buffer.add(
          next.input_nr, std::move(output), opt_parent_stream, opt_next_stream);
      if (is_ready) {
        auto queue = ready_queue(cpu_ready_queue, input_buffer.device());
        queue->push(
            NodeTask(graph_task, next.function, std::move(input_buffer)));
        not_ready.erase(not_ready_it);
      }
    }
  }
}
```

其核心操作是这一个调用：

```
auto outputs = call_function(graph_task, func, inputs);
```

call_function的实现也在engine.cpp中。

```
static variable_list call_function(
    std::shared_ptr<GraphTask>& graph_task,
    Node* func,
    InputBuffer& inputBuffer) {
  CheckpointValidGuard cpvguard(graph_task);
  auto& fn = *func;
  auto inputs =
      call_pre_hooks(fn, InputBuffer::variables(std::move(inputBuffer)));

  if (!graph_task->keep_graph_) {
    fn.will_release_variables();
  }

  const auto has_post_hooks = !fn.post_hooks().empty();
  variable_list outputs;

  if (has_post_hooks) {
    // In functions/accumulate_grad.cpp, there is some logic to check the
    // conditions under which the incoming gradient can be stolen directly
    // (which elides a deep copy) instead of cloned. One of these conditions
    // is that the incoming gradient's refcount must be 1 (nothing else is
    // referencing the same data).  Stashing inputs_copy here bumps the
    // refcount, so if post hooks are employed, it's actually still ok for
    // accumulate_grad.cpp to steal the gradient if the refcount is 2.
    //
    // "new_grad.use_count() <= 1 + !post_hooks().empty()" in
    // accumulate_grad.cpp accounts for this, but also creates a silent
    // dependency between engine.cpp (ie, this particular engine
    // implementation) and accumulate_grad.cpp.
    //
    // If you change the logic here, make sure it's compatible with
    // accumulate_grad.cpp.
    auto inputs_copy = inputs;
    outputs = fn(std::move(inputs_copy));
  } else {
    outputs = fn(std::move(inputs));
  }

  validate_outputs(fn.next_edges(), outputs, [&](const std::string& msg) {
    std::ostringstream ss;
    ss << "Function " << fn.name() << " returned an " << msg;
    return ss.str();
  });
  if (has_post_hooks) {
    // NOLINTNEXTLINE(bugprone-use-after-move)
    return call_post_hooks(fn, std::move(outputs), inputs);
  }
  return outputs;
}
```

可以看到，call_function()的核心逻辑就是执行fn()函数，这个fn函数指针是NodeTask的成员。而这个NodeTask是

```
    queue->push(
        NodeTask(graph_task, std::move(graph_root), std::move(input_buffer)));
```

```
struct NodeTask {
  std::weak_ptr<GraphTask> base_;
  std::shared_ptr<Node> fn_;
  // This buffer serves as an implicit "addition" node for all of the
  // gradients flowing here.  Once all the dependencies are finished, we
  // use the contents of this buffer to run the function.
  InputBuffer inputs_;
  // When worker receives a task with isShutdownTask = true, it will immediately
  // exit. The engine sends a shutdown task to every queue upon its destruction.
  bool isShutdownTask_;

  int getReentrantDepth() const;

  NodeTask(
      // NOLINTNEXTLINE(modernize-pass-by-value)
      std::weak_ptr<GraphTask> base,
      std::shared_ptr<Node> fn,
      InputBuffer inputs,
      bool isShutdownTask = false)
      : base_(base),
        fn_(std::move(fn)),
        inputs_(std::move(inputs)),
        isShutdownTask_(isShutdownTask) {}
};
```

这样就知道所谓的NodeTask的成员fn_其实就是graph_root，而graph_root又是edge_list的第一项

```
  auto graph_root = skip_dummy_node
      ? roots.at(0).function
      : std::make_shared<GraphRoot>(roots, inputs);
```

roots是一开始从Python调用C++函数的时候生成的，也就是在函数THPEngine_run_backward的实现里，相关的代码如

```
PyObject* THPEngine_run_backward(
    PyObject* self,
    PyObject* args,
    PyObject* kwargs) {
//...

  edge_list roots;
  roots.reserve(num_tensors);
  variable_list grads;
  grads.reserve(num_tensors);
  for (const auto i : c10::irange(num_tensors)) {
    PyObject* _tensor = PyTuple_GET_ITEM(tensors, i);
    THPUtils_assert(
        THPVariable_Check(_tensor),
        "element %d of tensors "
        "tuple is not a Tensor",
        i);
    const auto& variable = THPVariable_Unpack(_tensor);
    TORCH_CHECK(
        !isBatchedTensor(variable),
        "torch.autograd.grad(outputs, inputs, grad_outputs) called inside ",
        "torch.vmap. We do not support the case where any outputs are ",
        "vmapped tensors (output ",
        i,
        " is being vmapped over). Please "
        "call autograd.grad() outside torch.vmap or file a bug report "
        "with your use case.")
    auto gradient_edge = torch::autograd::impl::gradient_edge(variable);
    THPUtils_assert(
        gradient_edge.function,
        "element %d of tensors does not require grad and does not have a grad_fn",
        i);
    roots.push_back(std::move(gradient_edge));

    //...
  }

//...
    }
```

gradient_edge的定义在torch/csrc/autograd/variable.cpp中:

```
Edge gradient_edge(const Variable& self) {
  // If grad_fn is null (as is the case for a leaf node), we instead
  // interpret the gradient function to be a gradient accumulator, which will
  // accumulate its inputs into the grad property of the variable. These
  // nodes get suppressed in some situations, see "suppress gradient
  // accumulation" below. Note that only variables which have `requires_grad =
  // True` can have gradient accumulators.
  if (const auto& gradient = self.grad_fn()) {
    return Edge(gradient, self.output_nr());
  } else {
    return Edge(grad_accumulator(self), 0);
  }
}
```

Edge的定义在torch/csrc/autograd/edge.h中，可以看出，Edge中的函数其实就是Variable中的grad_fn，而Variab

```
/// Represents a particular input of a function.
struct Edge {
  Edge() noexcept : function(nullptr), input_nr(0) {}

  Edge(std::shared_ptr<Node> function_, uint32_t input_nr_) noexcept
      : function(std::move(function_)), input_nr(input_nr_) {}

  /// Convenience method to test if an edge is valid.
  bool is_valid() const noexcept {
    return function != nullptr;
  }

  // Required for use in associative containers.
  bool operator==(const Edge& other) const noexcept {
    return this->function == other.function && this->input_nr == other.input_nr;
  }

  bool operator!=(const Edge& other) const noexcept {
    return !(*this == other);
  }

  /// The function this `Edge` points to.
  std::shared_ptr<Node> function;

  /// The identifier of a particular input to the function.
  uint32_t input_nr;
};
```

参考

- PYTORCH 自动微分（二）https://zhuanlan.zhihu.com/p/111874952
- https://zhuanlan.zhihu.com/p/69294347
- https://pytorch.org/blog/how-computational-graphs-are-executed-in-pytorch/
- https://www.cnblogs.com/rossiXYZ/p/15481235.html