第1章 PyTorch程序的基本结构

主要内容

- PyTorch介绍
-

  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，用

ATen

ATen，来自于 A TENsor library for C++11的缩写；PyTorch的C++ tensor library。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，这部分动态生成的这

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

初步介绍

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

以下的编译过程是基于

编译环境准备

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.

If you want to compile with CUDA support, install the following (note that
CUDA is not supported on macOS)

```
NVIDIA CUDA 10.2 or above
NVIDIA cuDNN v7 or above
Compiler compatible with CUDA
```

Note: You could refer to the cuDNN Support Matrix for cuDNN versions with
the various supported CUDA, CUDA driver and NVIDIA hardware

If you want to disable CUDA support, export the environment variable USE_CUDA=0. Other potentially useful environment variables may be found in setup.py.

If you are building for NVIDIA's Jetson platforms (Jetson Nano, TX1, TX2, AGX Xavier), Instructions to install PyTorch for Jetson Nano are available here

If you want to compile with ROCm support, install

```
AMD ROCm 4.0 and above installation
ROCm is currently supported only for Linux systems.
```

If you want to disable ROCm support, export the environment variable USE_ROCM=0. Other potentially useful environment variables may be found in setup.py.

PyTorch的setup.py

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

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：

编译步骤

如果没有什么问题，编译的最后输出如下：

```
......

building 'torch._C' extension
creating build/temp.linux-x86_64-3.7
creating build/temp.linux-x86_64-3.7/torch
creating build/temp.linux-x86_64-3.7/torch/csrc
gcc -pthread -B /root/anaconda3/compiler_compat -Wl,--sysroot=/ -Wsign-compare -DNDEBUG -g -
gcc -pthread -shared -B /root/anaconda3/compiler_compat -L/root/anaconda3/lib -Wl,-rpath=/ro
building 'torch._C_flatbuffer' extension
gcc -pthread -B /root/anaconda3/compiler_compat -Wl,--sysroot=/ -Wsign-compare -DNDEBUG -g -
gcc -pthread -shared -B /root/anaconda3/compiler_compat -L/root/anaconda3/lib -Wl,-rpath=/ro
building 'torch._dl' extension
gcc -pthread -B /root/anaconda3/compiler_compat -Wl,--sysroot=/ -Wsign-compare -DNDEBUG -g -
gcc -pthread -shared -B /root/anaconda3/compiler_compat -L/root/anaconda3/lib -Wl,-rpath=/ro
-------------------------------------------------------------------------
|                                                                       |
|    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                                |
|                                                                       |
-------------------------------------------------------------------------
```

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：

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-01 -02 -03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

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

```
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
}
```

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

第三章 PyTorch中重要的数据结构

Tensor

在C++中，Tensor的定义在

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。

```
// c10/core/TensorOptions.h
```

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
```

基于C++的算子实现

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

Dispatcher机制

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会寻找相应自

```
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_;

  // ...
};
```

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Dispather::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::(anonymous
    cb=..., strides=0x7fffffffd300) at ../aten/src/ATen/native/cpu/Loops.h:246
#3  at::native::AVX2::unroll_contiguous_scalar_checks<function_traits<at::native::(anonymous
    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::na
    params#0=params#0@entry=0x7fffffffd270, params#1=params#1@entry=0x7fffffffd300, params#2
    params#3=params#3@entry=1) at ../c10/util/FunctionRef.h:43
```

Dispatcher

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

算子的注册过程

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

下面我们看一下这两个宏的实现：

```
#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)
```

```
#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)
```

在VariableTypeEverything.cpp中，有这样一条语句：

```
TORCH_LIBRARY_IMPL(aten, Autograd, m) {
  ...
}
```

展开之后的形式如下：

```
static void TORCH_LIBRARY_IMPL_init_aten_Autograd_C10_UID(torch::Library&);
  static const torch::detail::TorchLibraryInit
      TORCH_LIBRARY_IMPL_static_init_aten_Autograd_C10_UID(
      torch::Library::IMPL,
      c10::guts::if_constexpr<c10::impl::dispatch_key_allowlist_check(
          c10::DispatchKey::k)>(
          []() {
            return & TORCH_LIBRARY_IMPL_init_aten_Autograd_C10_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)
```

对于每一个dispatch_key，宏TORCH_LIBRARY_IMPL定义了一个函数，允许用户在这个函数体内注册

例如，在下面的代码中，注册了包括add_Tensor在内的多个算子。

```
// 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)
  );
  // ...
}
```

THPVariable_add ->

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

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

```
# 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

```
# 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。

```
# 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:

```
# 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]) -> Tensor
        ...

    _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.forwa

        # 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

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：

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

计算图

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：

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   gr

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

PyTorch全局设计

GIL

GIL(Global Interpreter Lock)是一个全局的锁，深度学习框架属于计算密集型的系统，在进行计算时要尽可能的

参考

- 一文详解 Python GIL 设计 https://zhuanlan.zhihu.com/p/363040263
- 万字综述，核心开发者全面解读PyTorch内部机制 https://zhuanlan.zhihu.com/p/67834038
  # 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

参考

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

第11章 JIT

第3章 自动微分

Index

- 理论知识
- 梯度的保存
- 梯度的计算

- 反向传播

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。因此最合适的方式

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

```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,cppcoreguideline
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

```
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