Caffe2笔记

个人 API 接口

一个观察：在Caffe2里面，一个网络net其实是一系列operator按照添加顺序组成的集合（可用`core.Net._net.op`获取所有op）。在执行网络前（`workspace.RunNet(some_net)`），按照添加的顺序逐个完成操作。这是最简单的一种模式，可在`core.Net._net.type`中进行设置，默认为`simple`（猜测为DAG模式）。反向传播可通过`core.Net.AddGradientOperator`添加进op集合，附在前向op的后面。

To Do

- [ ] 整理API

Caffe2安装

参考官方

Caffe2基础模块

core

```
1. from caffe2.python import core
```

core模块处理caffe2的整个环境，Device，环境的初始化，网络的初始定义等。
常用的接口函数包括：

- `GlobalInit`：初始化Caffe2的环境。初始化分为三个步骤：
  - 使用`REGISTER_CAFFE2_EARLY_INIT_FUNCTION`注册op。因为尚未完全初始化，这一步中logging可能输出错误的内容
  - 解析Caffe的命令行输入参数，初始化logging
  - 使用`REGISTER_CAFFE2_INIT_FUNCTION`注册op
  - `core.GlobalInit(['caffe2', '--caffe2_log_level=0'])`这一行可能代表命令行`caffe2 --caffe2_log_level=0`，其中参数`caffe2_log_level`为日志的级别
- `DeviceOption`：设置模型所在设备参数。输入参数如下：

    - `device_type`：设备类型，Proto中有可选设备列表，包括`CPU` `CUDA` `MKLDNN` `OPENGL` `OPENCL` `IDEEP` `HIP`等；缺省为`CPU`
    - `cuda_gpu_id`：如果是CUDA（GPU），设定GPU编号；这个编号好像是与CUDA环境变量`CUDA_VISIBLE_DEVICES`设置无关的？**需验证**
    - `random_seed`：系统随机数种子，缺省为`None`
    - `node_name`：不明
    - `numa_node_id`：不明
    - `extra_info`：不明

- `ScopedName`：将当前作用域名前缀到数据名之前
- `CreateOperator`：用于将某operator加入到网络或模型中
- `Net`：Net类用于产生一个网络实例，初始化输入为name或者`proto.NetDef`。**网络类会有更具体的介绍**
- `BlobReference`：基础blob类，用于索引`workspace`的具体数据，可通过`str(blob)`直接转化为其名字
- `ExcutionStep`：不明
- `Plan`：不明

workspace

```
1. from caffe2.python import workspace
```

workspace有点像Matlab中的workspace，用于创建和存储数据blob，可在workspace中通过blob名字获取和修改对应的blob，所以workspace可看成一系列`{blob_name, blob_data}`组成的集合。当然，workspace本身还有其他功能。
常用函数接口：

- `FeedBlob(blob_name, data)`：给某blob输入数据
- `FetchBlob(blob_name)`：获得某个blob的数据，返回为`numpy.ndarray`
- `FetchBlobs(blob_names)`：获取某些blob数据
- `blobs`：当前`workspace`中所有的blob，`dict-like`
- `HasBlob(blob_name)`：判断当前`workspace`是否有某个blob
- `CreateNet(name)`：创建名为name的`Net`
- `RunNetOnce`：运行一次网络，运行完即销毁（个人理解是销毁`workspace`中的网络内的op的实例）；常用于初始化网络
- `RunNet`：运行网络；需要先`CreateNet`
- `RunOperatorOnce`：运行一次op
- `RunOperator`：运行op
- `CurrentWorkspace`：当前工作空间
- `ResetWorkspace`：重置工作空间
- `SwitchWorkspace`：切换工作空间

operator

`operator`代替了caffe中的layer，作为caffe2的基础单元。`operator`添加到网络时的方法基本统一为`net.SOME_OP([blobs_in], [blobs_out], **kwargs)`，blobs_in可以是`core.BlobReference`，或者它们的名字。

```
# X : a blob in the workspace, with name 'X', shape [batchsize, dim_in]
# w : a weight blob in the workspace, with name 'w', shape [dim_out, dim_in]
# b : a bias blob in the workspace, with name 'b', shape [dim_out]
# net : a core.Net instance network
# 我们添加一个FC算子，输出的blob名字为'y'
# 以下方法都是正确的
y1 = net.FC([X, w, b], 'y1')
y2 = net.FC(['X', w, b], 'y2')
y3 = net.FC(['X','w','b'], 'y3')

net.FC([X, w, b], 'y4')

workspace.CreateBlob('y5')
net.FC([X, w, b], 'y5')
# 或者
y5 = net.FC([X, w, b], 'y5')

# 这些方法都可以正确添加一个FC算子
```

`net.SOME_OP`返回的均为其输出blob对应的`BlobReference`

`operator`添加机制：这里提一下算子的添加机制，`net.SOME_OP`方法中，`net`是一个`core.Net`实例，`SOME_OP`其实不是`core.Net`的成员方法，只不过是`core.Net`类覆盖了`__getattr__`方法，将其修改为获取`SOME_OP`对应的已注册的算子，然后添加到网络的执行列表中。

caffe2_pb2

```
from caffe2.proto import caffe2_pb2
```

目前看到的使用是在`core.DeviceOption`中，用于设置GPU

brew

```
from caffe2.python import brew
```

`brew`模块包装了部分常见layer，如`fc`，`relu`，`conv`等，配合`model_helper.ModelHelper`模块快速搭建网络。如果使用非`brew`的operator添加一个全连接层，需要自己定义其参数`W`，`b`，`net.GaussianFill([], ['W'], shape=[dim_out, dim_in])`，

`net.GaussianFill([], ['b'], shape=[dim_out,])`，
然后
`net.FC([X, 'W', 'b'], 'y')`
而使用`brew.fc`接口，只需要决定输入输出维度，毋需自己定义`W`，`b`
`y = brew.fc(model, X, 'y', dim_in, dim_out)`
其中`model`为`model_helper.ModelHelper`类型

model_helper

```
1. from caffe2.python import model_helper
```

包装了`core.Net`，配合`brew`中的helper函数，可快速添加一些常见层。

optimizer

```
1. from caffe2.python import optimizer
```

predictor_exporter

```
1. import caffe2.python.predictor.predictor_exporter as pe
```

Net

```
1. some_net = core.Net(name='some_net')
```

Caffe2基本使用流

大部分代码来自官方教程

先介绍Caffe2的基本用法：

- **创造输入op和blob**：可使用`TensorProtoDBInput`等函数

```
1. # Add input to the net
2. def add_input(model, batch_size, db, db_type):
```

```
    ### load the data from db - Method 1 using brew
    #data_uint8, label = brew.db_input(
    #    model,
    #    blobs_out=["data_uint8", "label"],
    #    batch_size=batch_size,
    #    db=db,
    #    db_type=db_type,
    #)
    ### load the data from db - Method 2 using TensorProtosDB
    data_uint8, label = model.TensorProtosDBInput(
        [], ["data_uint8", "label"], batch_size=batch_size,
        db=db, db_type=db_type)
    # cast the data to float
    data = model.Cast(data_uint8, "data", to=core.DataType.FLOAT)
    # scale data from [0,255] down to [0,1]
    data = model.Scale(data, data, scale=float(1./256))
    # don't need the gradient for the backward pass
    data = model.StopGradient(data, data)
    return data, label
```

- **创造网络op和输出blob**：可使用`model.SOME_OP`的形式添加操作，或者使用`brew`接口定义网络各层。具体API待整理

```
# Define a mlp net
def mlp(model, data):
    dim_in = 28*28
    num_classes = 10
    fc1 = brew.fc(model, data, 'fc1', dim_in, 1024)
    fc1 = brew.relu(model, fc1, fc1)
    fc2 = brew.fc(model, fc1, 'fc2', 1024, 2048)
    fc2 = brew.relu(model, fc2, fc2)
    fc3 = brew.fc(model, fc2, 'fc3', 2048, num_classes)
    pred = brew.softmax(model, fc3, 'softmax')
    return pred
```

- **设定loss，指标与优化器**
- **训练**
- **保存，部署**

```
# Toy example: MNIST
import numpy as np
import os
import shutil
import operator
import caffe2.python.predictor.predictor_exporter as pe
from caffe2.proto import caffe2_pb2
from caffe2.python import (
    brew,
    core,
```

```
    model_helper,
    optimizer,
    visualize,
    workspace,
)

# Initialize caffe2 core
core.GlobalInit(['caffe2', '--caffe2_log_level=0'])
print("Necessities imported!")

# Get the mnist lmdb
db_url = "http://download.caffe2.ai/databases/mnist-lmdb.zip"
data_dir = 'path/to/the/dataset/dir/mnist'
output_dir = 'path/to/the/output/dir'
def download_mnist():
    '''Downloads resources from s3 by url and unzips them to the provid
ed path'''
    import requests, zipfile, StringIO
    print("Downloading... {} to {}".format(db_url, data_dir))
    r = requests.get(db_url, stream=True)
    z = zipfile.ZipFile(StringIO.StringIO(r.content))
    z.extractall(data_dir)
    print("Completed download and extraction.")

# Get input
download_mnist()
arg_scope = {'order':'NCHW'}
gpu_id = 0
device_opt = caffe2_pb2.DeviceOption(caffe2_pb2.CUDA, gpu_id)
with core.DeviceScope(device_opt):
    train_model = model_helper.ModelHelper(name='train_model')
    train_data, train_label = add_input(train_model,
                        batch_size=64,
                        db=os.path.join(data_dir, 'mnist-train-nchw-lmd
b'),
                        db_type='lmdb')
    pred = mlp(train_model, train_data)
    xent = train_model.LabelCrossEntropy([pred, train_label], 'xent')
    loss = train_model.AveragedLoss(xent, 'loss')
    train_acc = brew.accuracy(train_model, [pred, train_label], 'train_
acc')
    train_model.AddGradientOperator([loss])

    # Define optimizer
    optimizer.build_sgd(train_model,
                        base_learning_rate=1e-2,
                        policy='step',
                        stepsize=1,
                        gamma=0.999)

    # Initialize
    workspace.RunNetOnce(train_model.param_init_net)
```

```
60.     workspace.CreateNet(train_model.net, overwrite=True)
61.
62.     # Train process
63.     max_epoch = 1000
64.     for epoch in range(max_epoch):
65.         workspace.RunNet(train_model)
66.         t_loss = workspace.blobs['loss']
67.         t_acc = workspace.blobs['train_acc']
68.         print 'Epoch {}, Loss {}, Acc {}'.format(epoch, t_loss, t_acc)
69.
70.     # For test
71.     test_model = model_helper.ModelHelper(name='test_model')
72.     test_data, test_label = add_input(test_model,
73.                         batch_size=64,
74.                         db=os.path.join(data_dir, 'mnist-test-nchw-lmd
    b'),
75.                         db_type='lmdb')
76.     pred = mlp(test_model, test_data)
77.     test_acc = brew.accuracy(test_model, [pred, test_label],
    'test_acc')
78.
79.     # Initialization
80.     workspace.RunNetOnce(test_model.param_init_net)
81.     workspace.CreateNet(test_model.net, overwrite=True)
82.     for epoch in range(100):
83.         workspace.RunNet(test_model)
84.         # te_acc = workspace.FetchBlob('test_acc')
85.         te_acc = workspace.blobs['test_acc']
86.         print 'Epoch {}, Acc {}'.format(epoch, te_acc)
87.
88.     # Define the deploy net
89.     deploy_model = model_helper.ModelHelper(name='deploy_model')
90.     pred = mlp(deploy_model, 'data')
91.
92.     # Save the model
93.     pe_meta = pe.PredictorExportMeta(
94.                 predict_net=deploy_model.net.Proto(),
95.                 parameters=[str(b) for b in deploy_model.params],
96.                 inputs=["data"],
97.                 outputs=["softmax"],
98.             )
99.
100.    # Save the model to a file. Use minidb as the file format
101.    pe.save_to_db("minidb", os.path.join(root_folder, "mnist_model.mini
    db"), pe_meta)
102.    t_data = workspace.blobs['data']
103.
104.    # Reset the workspace, and load
105.    workspace.ResetWorkspace()
106.    # Load the model
107.    predict_net = pe.prepare_prediction_net(os.path.join(output_dir, "m
    nist_model.minidb"), "minidb")
```

```
108.     workspace.FeedBlob('data', t_data)
109.     workspace.RunNetOnce(predict_net)
110.     t_pred = workspace.FetchBlob['softmax']
```

网络规模估计

一种直观的估计方法是在网络运行之后，获取workspace中所有Blob的数据值，然后获取其大小，统计总的网络规模。但是这种简单粗暴的方法存在一些问题

- 在网络已运行的情况下，再去统计网络规模可能意义不大。
- 很多OP的输入和输出的Blob命名完全相同，如果它们本身底层是在不同的存储空间中，这种方法就无法将被覆盖掉的Blob数据统计进来
- 无法直接估计FLOPs，仍需要根据网络拓扑来计算FLOPs
  实际所需的应该是先建立了网络拓扑图，然后根据拓扑图估计规模。这样仅需网络建立，甚至只需要网络的Proto文件，而不需要先运行起来，就可以完成网络规模的估计，提供参考。这一过程中，可以一并估计FLOPs，根据需要，也可以将被覆盖的Blob统计进来。总的来说，这一方法更为合理，只是在实施过程中，需要处理若干问题：
- 某些Blob无法直接估计，如`DequeBlob`队列算子，用于加载数据。这一Blob信息不会通过Proto文件体现出来；解决方法是，额外传入网络所需的外部Blob信息（大小）
- 某些Blob的输出大小是概率性的，如RPN网络中`GenerateProposals`，`SampleAs`等OP，其输出大小会根据实际数据发生改变，这也是Proto文件无法体现的；这一点无法完全解决，只能根据数据情况，自行设定一个大致可能的输出大小。
- 某些自定义OP无法直接处理，需要额外的定制化估算方法。最典型的就是Detectron模型中的`CollectDistribute`算子，将ROI分发到FPN各层。这种自定义算子需要自定义的估算方法。

参数量估计

由于能产生参数的OP有限，如`FC`，`Conv`，`ConvTranspose`等算子，只需要根据这些OP的接口，找出对应的参数Blob即可

数据量估计

网络的拓扑是位于Proto数据中，而且各个算子的顺序完全按照添加顺序而定；反向传播的算子位于前向传播算子的后面，而参数更新算子，如`MomentumSGDUpdate`算子，则位于反向传播算子后面。另一方面，每个OP都有固定顺序的输入，如`FC`算子的三个输入分别为数据`X`，权重`W`和偏置`b`（如果有的话）。因此，可按照OP的顺序，在已知输入图像大小的情况下，根据算子的处理方法，自行计算各算子的输出Blob大小，这样也就能够计算各算子的输出数据量，和算子所需的FLOPs。

FLOPs估计

与数据量估计一致，根据各OP的性质，手动编写各OP的FLOPs。