Echo DNN

DL ImageGen Bench

概要设计说明

date	2018-12-17
version	1.0.0
author	CKH

Revision History

date	version	desc.	author
2018-12-17	1.0.0	created	CKH

Contents

Echo DNN

DL ImageGen Bench

概要设计说明

1. 工程配置

1.1 软件环境

1.2 第三方库

1.3 工程结构

1.3.1 整体工程结构

1.3.2 ./apputils目录结构

1.3.3 ./data目录结构

1.3.4 ./dataset目录结构

1.3.5 ./models目录结构

1.3.6 ./jupyter目录结构

1.3.7 ./options目录结构

1.3.8 ./distiller目录结构

2. 工程说明

2.1 DL ImageGen Bench简介

2.1 流程图

2.1.1 Train

2.1.2 Test

2.2 模块说明

2.2.1 Options Get

2.2.1.1 Training Phase Options

2.2.1.2 Testing Phase Options

2.2.2 Dataset Get

2.2.2.1 Training Phase Dataset

2.2.2.2 Testing Phase Dataset

2.2.3 Trainer

2.2.3.1 Build Model

2.2.3.2 Train Process

2.2.3.3 Model Save

2.2.3.4 Run

2.2.4 Tester

2.2.4.1 Build Model

2.2.4.2 Test Process

2.2.4.3 Run

2.3.5 Compression Scheduler

2.3.5.1 Distiller APIs

2.3.5.2 Compression Scheduler yaml
2.3.6 Tools Collections
2.3.6.1 Sensitivity Analysis
2.3.6.2 Sparsity Display
2.3.6.3 MACs Display
2.3.6.4 Draw Model
2.3.6.5 Transform Network to ONNX format

1. 工程配置

1.1 软件环境

- os: ubuntu 16.04 LTS
- cuda: 9.0.176
- cudnn: 7.0.5
- python: 3.6.5

1.2 第三方库

见工程目录./requirements.txt

Library Name	Version
torch	==0.4.0
numpy	==1.14.3
torchvision	==0.2.1
scipy	==1.1.0
gitpython	==2.1.11
torchnet	==0.0.4
tensorflow-gpu	==1.6.0
tensorboard	==1.6.0
tenosrboard-logger	==0.1.0
tensorboardX	==1.4
pydot	==1.2.4
tabulate	==0.8.2
pandas	==0.22.0
jupyter	==1.0.0
matplotlib	==2.2.2
qgrid	==1.0.2
graphviz	==0.8.2
ipywidgets	==7.1.2
bqplot	==0.10.5
pyyaml	==3.12
pytest	==3.5.1
h5py	==2.7.1
tabulate	==0.8.2

1.3 工程结构

1.3.1 整体工程结构

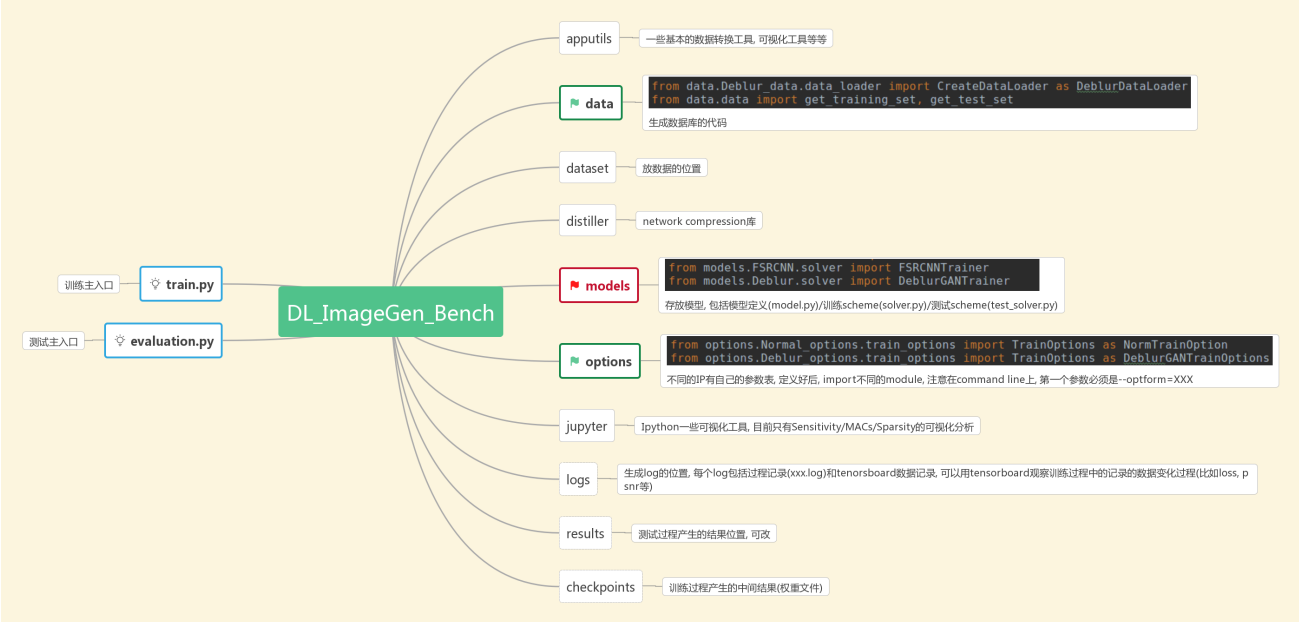

Figure 1. DL_ImageGen_Bench 代码结构

整个DL_ImageGen_Bench的代码主要包含以下几个部分

- **train.py** 训练的主程序, 在训练阶段由这个程序来管理参数列表获取, 数据导入以及模型的选择
- **test.py** 测试的主程序, 在测试阶段由这个程序来管理参数列表获取, 数据导入以及模型的选择
- **./apputils** 训练或测试时会用到的工具集, 比如数据转换或可视化等, Developer可以将自己定义的工具集放在这个文件夹下
- **./data** 生成数据库的代码, 生成数据时的归一化/缩放等操作的代码在这个文件夹下定义
- **./dataset** 存放原始数据的位置
- **./distiller** Network Compression所用到的Distiller库存放位置
- **./models** 存放模型的位置, 一般包括模型定义(model.py)/训练scheme(solver.py)测试scheme(test_solver.py)
- **./options** 定义参数列表的位置, 不同的应用有自己的参数表, 定义好后, import不同的module, 注意在command line上, 第一个参数必须是--optform=XXX
- **./jupyter** Ipython的一些可视化工具, 目前有Sensitivity/MACs/Sparsity的可视化分析
- **./logs** 生成log的位置, 每个log包括过程记录(xxx.log)和tenorsboard数据记录, 可以用tensorboard观察训练过程中的记录的数据变化过程(比如loss, psnr等)
- **./results** 测试过程产生的结果位置
- **./checkpoints** 训练过程中产生的权重文件的位置

1.3.2 ./apputils目录结构

```
apputils
├── __init__.py
├── execution_env.py            /* 通用工具, 打印log辅助工具 */
├── model_summaries.py          /* 通用工具, 模型评估工具*/
└── platform_summaries.py       /* 通用工具, 模型可视化及模型转换工具*/
├── Deblur_apputils             /* DeblurGAN 专有的工具集 */
|   ├── ...
```

1.3.3 ./data目录结构

```
data
├──__init__.py
├── data.py                     /* SR 数据库生成接口  */
├── dataset.py                  /* SR 从Folder原始数据生成数据库 */
├── Deblur_data                 /* DeblurGAN 专有数据生成代码 */
|   ├── ...
```

1.3.4 ./dataset目录结构

```
dataset
├── BSDS300                     /* SR数据 存放位置 */
|   └── images
|       ├── test
|       |   └── ...
|       └── train_aug
|           └── ...
└── Deblur                      /* DeblurGAN数据 存放位置 */
    └── train
        └── ...
```

1.3.5 ./models目录结构

```
models
├── __init__.py
├── test_solver.py                  /* 通用 test scheme定义, SR各种模型通用, 如果是其他IP需要
                                       参考这个文件自己定义 */
├── C2SRCNN                         /* SR 模型定义 */
│   ├── __init__.py
│   ├── solver.py                   /* SR train scheme定义 */
│   ├── model.py                    /* SR 模型结构定义*/
│   ├── agp_prune.yaml              /* SR 非结构化剪枝scheme定义 */
│   ├── filter_prune.yaml           /* SR 结构化剪枝scheme定义 */
│   ├── quantization_DoReFa.yaml    /* SR 量化scheme定义(DoReFa方法) */
│   └── quantization_PACT.yaml      /* SR 量化scheme定义(PACT方法) */
├── Deblur                         /* DeblurGAN 模型定义 */
│   ├── ...
├── cifar10                        /* CIFA10 数据库相关模型(用不到, 但distiller库有包含关系)
*/
│   ├── ...
├── imagenet                       /* IMAGENET 数据库相关模型(用不到, 但distiller库有包含关系)
*/
│   ├── ...
```

1.3.6 ./jupyter目录结构

```
jupyter
├── compression_insights.ipynb    /* MACs/Sparsity jupyter可视化工具 */
└── sensitivity_analysis.ipynb    /* Sensitivity jupyter可视化工具 */
```

1.3.7 ./options目录结构

```
options
├── __init__.py
└── Normal_options                /* SR 参数列表定义 */
    ├── __init__.py
    ├── test_options.py           /* SR 测试scheme 参数列表定义*/
    └── train_options.py          /* SR 训练scheme 参数列表定义*/
├── Deblur_options                /* DeblurGAN 参数列表定义 */
|   ├── ...
```

1.3.8 ./distiller目录结构

```
distiller
├── __init__.py
├── config.py                     /* Distiller算法库 参数列表定义 */
├── scheduler.py                  /* Distiller算法库 compression对象scheduler定义,
                                        compression过程的主入口 */
├── sensitivity.py                /* Distiller算法库 计算权重敏感度 */
├── thinning.py                   /* Distiller算法库 在filter prune后进行网络结构瘦身 */
├── thresholding.py               /* Distiller算法库 prune获取剪枝阈值 */
├── utils.py                      /* Distiller算法库 工具集合 */
├── directives.py                 /* Distiller算法库 yaml上的指令定义(指令对应的算法) */
├── knowledge_distillation.py     /* Distiller算法库 知识蒸馏方法定义 */
├── learning_rate.py              /* Distiller算法库 在训练过程中调整lr */
├── model_summaries.py            /* Distiller算法库 模型评估 */
├── policy.py                     /* Distiller算法库 network compression中对于各种操作的管理
                                     包括PruningPolicy, RegularizationPolicy, LRPolicy*/
├── pruning                       /* Distiller算法库 剪枝方法定义 */
|   ├── ...
├── quantization                  /* Distiller算法库 量化方法定义 */
|   ├── ...
├── regularization                /* Distiller算法库 正则化剪枝方法定义(结构性剪枝的一种) */
|   ├── ...
├── data_loggers                  /* Distiller算法库 compression scheme中的logger管理 */
|   ├── ...
```

2. 工程说明

2.1 DL ImageGen Bench简介

DL ImageGen Bench是一套基于pytorch的用于模型定义/训练/评估的代码框架, 它主要包括以下几个方面的功能:

- 模型训练(Train)
- 模型测试(Evaluation)
- 网络压缩(Network Compression)
- 模型性能评估, 包括参数敏感度分析(Sensitivity Analysis)、参数稀疏度分析(Sparsity Analysis)、计算量估计(MACs)、模型可视化(Visualization)等
- 模型转换(pytorch -> onnx)
- 多模型Benchmark, 统一模型训练/测试接口, 方便比较多个模型的表现

使用时, 分为两个阶段, 训练阶段(Training Phase)和测试阶段(Testing Phase), 分别对应着程序中的两个主入口`./train.py`和`./test.py`, 这两个主程序的具体使用方式在API Document (`./doc/html/index.html`)中有详细的介绍, 执行过程的work flow如下一节所示

2.1 流程图

2.1.1 Train

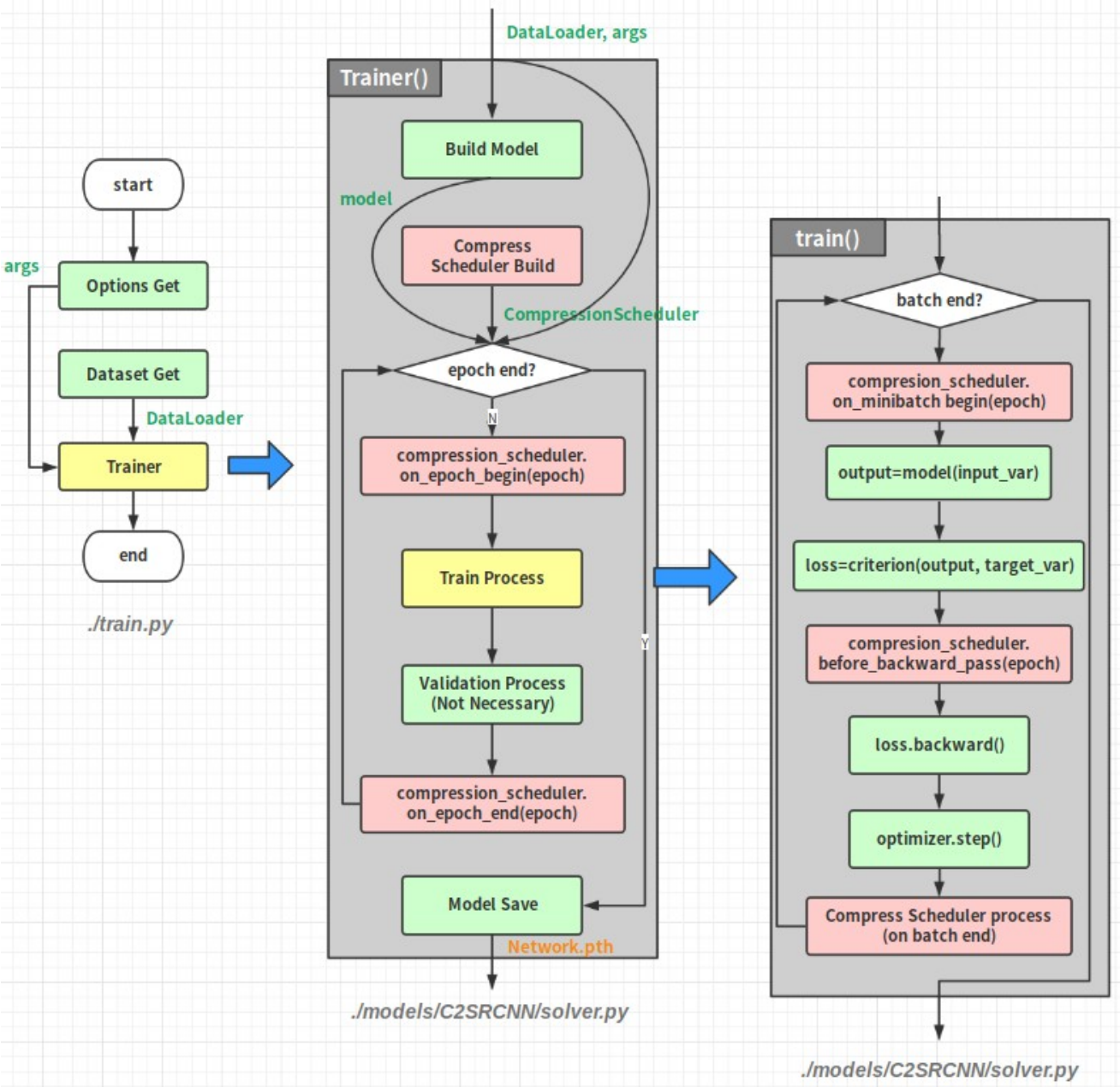

Figure 2. DL_ImageGen_Bench Train Flow Chart

在上图表示的Train Phase的Work Flow中, 不同的模块用不同的颜色表示

其中, 绿色的部分表示完全由Developer定义的部分, 也可以参照在**DL ImageGen Bench**中定义的*CXSRCNN for Image Super Resolution*或者是*DeblurGAN for Deblur*这两类模型的方法去定义, 因为每个模型的应用场景, 数据的处理方法还有需要的参数都不同, 所以把这些自由度留出来, 也方便从其他渠道得到的源码直接移植, 但是, 要保证这些模块提供给**DL ImageGen Bench**数据类型是一致的(具体需要什么接口数据类型在模块说明中会给出)(各模块的输出数据类型在下面模块说明中定义), 主要包括以下几个部分

- 模型定义(Build Model)
- 数据获取(Dataset Get)
- 参数列表(Options Get)
- 模型存储(Model Save))

红色的部分表示DL_ImageGen_Bench框架定义好的部分, 主要包括在Training的不同阶段插入的Compression Scheduler操作

黄色的部分表示由DL_ImageGen_Bench定义流程, 而其中的部分运算可以由Developer自己来定义, 比如在Train的过程中, 前向计算网络输出`output=model(input_var)`、反向计算梯度`loss=criterion(output, target_var)`以及优化器的具体定义`optizimer`这些都是由Developer来定义, 但是在Train的什么阶段插入什么Compression Scheduler操作必须按照Work Flow中规定的去执行

2.1.2 Test

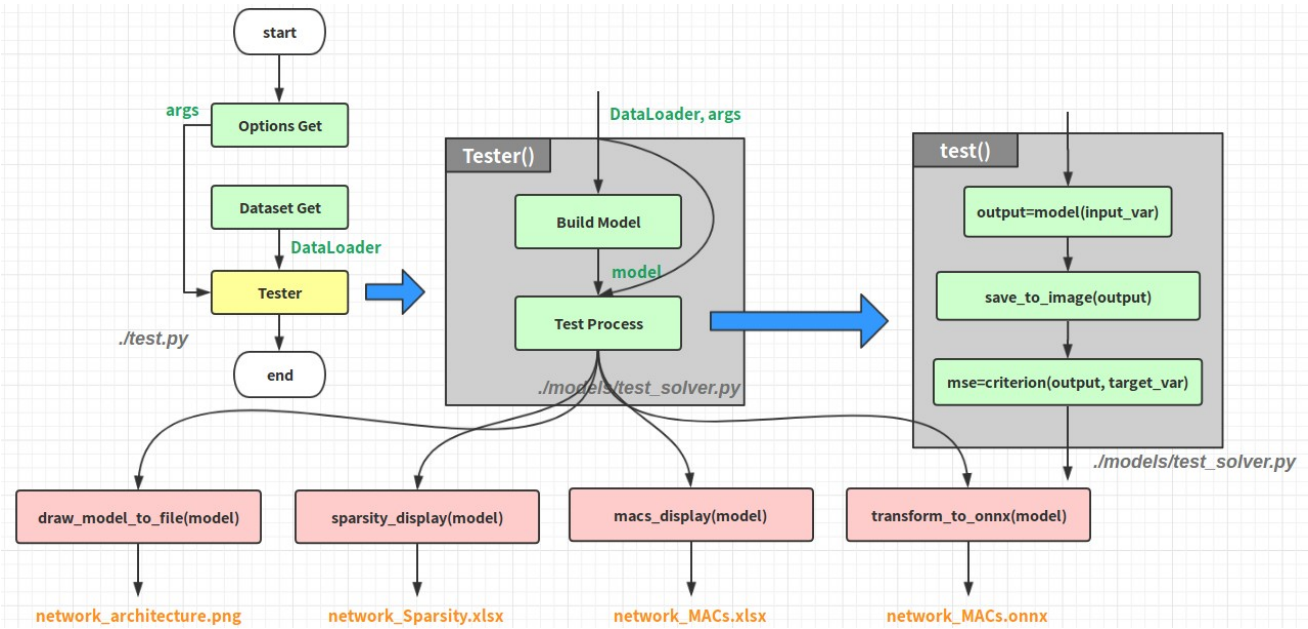

Figure 3. DL_ImageGen_Bench Evaluation Flow Chart

类似Train Phase的Work Flow, Evaluation Phase也包括了完全由Developer定义的部分(绿色)和DL_ImageGen_Bench框架定义好的部分(红色), 在Evaluation Phase中, 大部分的操作需要Developer去定义, 只是在测试结束`Test Process`后, 可以利用DL_ImageGen_Bench提供的工具(Tools Collections)来评估模型的性能以及转换模型, 这些工具包括:

- 权重敏感度分析(Sensitivity Analysis)
- 权重稀疏性分析(Sparsity Analysis)
- 模型可视化(Network Visualization)
- 统计计算量(MACs Estimator),
- 将模型由`pytorch`格式转换为通用框架格式`onnx`(Format Transform)

在下面的模块说明中, 我们会以SR的应用为例, 简单地介绍一下在使用过程中需要注意的模块的输出数据类型以及每个模块应该完成什么样的任务

2.2 模块说明

2.2.1 Options Get

2.2.1.1 Training Phase Options

CLASS TrainOptions()

定义位置

- `./options/Normal_options/train_options.py(4)`

简要描述

- 从命令行获取Training Phase的参数列表

调用例程

```
from options.Normal_options.train_options import TrainOptions as NormTrainOption
args = NormTrainOption().parse()
```

备注

- 在`./train.py`的执行过程中, 会根据第一个参数`--optform`的名字来调用不同的Options类, 新增的应用可以仿照SR和Deblur的写法来写自己的Options类

2.2.1.2 Testing Phase Options

CLASS TestOptions()

定义位置

- `./options/Normal_options/test_options.py(4)`

简要描述

- 从命令行获取Testing Phase的参数列表

调用例程

```
from options.Normal_options.test_options import TestOptions as NormTestOptions
args = NormTestOptions().parse()
```

备注

- 在`./test.py`的执行过程中, 同样会根据第一个参数`--optform`的名字来调用不同的Options类, 新增的应用可以仿照SR和Deblur的写法来写自己的Options类

2.2.2 Dataset Get

2.2.2.1 Training Phase Dataset

get_training_set*(upscale, train_dir)*

定义位置

- `./data/data.py(61)`

简要描述

- 获取Training dataset, 在pytorch中获取数据通常会先生成一个基类为`class torch.utils.data.dataset`的子类对象(比如SR应用中的`class DatasetFromFoler`), 这个对象并不直接保存样本数据, 而是定义一套从原始数据产生神经网络输入的方法(比如读取图像->随机Crop->将数据转为tensor), 然后用`class torch.utils.data.DataLoader`的对象来生成batch, 规定batch的大小, 来自哪个Dataset, 是否shuffle等, 只有在Train Process的内部迭代epoch时, 才会去真正地生成batch数据到cpu/gpu, 下面的`get_test_set()`和`get_eva_set()`都是类似的原理, 不再做冗述

参数列表

参数名	必选	类型	说明	in/out
upscale_factor	Yes	int	SR图像的scale倍率	in
train_dir	Yes	str	训练库的位置	in

返回参数说明

参数名	类型	说明
dataset	class DatasetFromFolder (./data/dataset.DatasetFromFolder)	训练数据库

调用例程

```
from torch.utils.data import DataLoader
from data.data import get_training_set
train_set = get_training_set(args.upscale_factor, args.traindata)
training_data_loader = DataLoader(dataset=train_set, batch_size=args.batchSize,
                                  shuffle=True)
```

get_validation_set*(upscale, train_dir)*

定义位置

- `./data/data.py(84)`

简要描述

- 获取Validation dataset, 在train的过程中用于supervise模型performance的数据库

参数列表

参数名	必选	类型	说明	in/out
upscale_factor	Yes	int	SR图像的scale倍率	in
train_dir	Yes	str	验证库的位置	in

返回参数说明

参数名	类型	说明
dataset	class DatasetFromFolder (./data/dataset.DatasetFromFolder)	验证数据库

调用例程

```
from torch.utils.data import DataLoader
from data.data import get_validation_set
validate_set = get_validation_set(args.upscale_factor, args.traindata)
validation_data_loader = DataLoader(dataset=validate_set, batch_size=args.batchSize,
                                    shuffle=True)
```

2.2.2.2 Testing Phase Dataset

get_testing_set*(opt)*

定义位置

- `./data/data.py(105)`

简要描述

- 获取Testing dataset

参数列表

参数名	必选	类型	说明	in/out
opt	Yes	dict	参数列表	in

返回参数说明

参数名	类型	说明
dataset	class DatasetFromFolder (./data/dataset.DatasetFromFolder)	测试数据库

调用例程

```
from torch.utils.data import DataLoader
from data.data import get_testing_set
test_set = get_testing_set(args.upscale_factor, args.traindata)
test_data_loader = DataLoader(dataset=test_set, batch_size=1, shuffle=True)
```

2.2.3 Trainer

CLASS C2SRCNNTrainer*(config, training_loader, validation_loader)*

基类

- `object`

定义位置

- `./models/C2SRCNN/solver.py(42)`

简要描述

- SR 神经网络训练过程(C2SRCNN, SRCNN的变体)

参数列表

参数名	必选	类型	说明	in/out
config	Yes	dict	参数列表	in
training_loader	Yes	class torch.utils.data.DataLoader()	训练集DataLoader	in
validation_loader	Yes	class torch.utils.data.DataLoader()	验证集DataLoader	in

调用例程

```
from models.C2SRCNN.solver import C2SRCNNTrainer
model = C2SRCNNTrainer(args, training_data_loader, validation_loader)
model.run()
```

成员函数说明

2.2.3.1 Build Model

> build_model*()*

简要描述

- 构建模型, 不只是构建模型结构, 还包括定义loss计算的方式, 优化方式(optimizeer), 还有learning rate策略

2.2.3.2 Train Process

> train*(self, epoch, compression_scheduler)*

简要描述

- 训练过程, 包括正向inference、反向backpropogation以及compression scheduler在训练过程需要加入的必要操作, 具体的过程可以看下面的编程示例

参数列表

参数名	必选	类型	说明	in/out
epoch	Yes	int	epoch id, 当前是training过程的第几个epoch	in
compression_scheduler	Yes	class CompressionScheduler (./distiller/scheduler .CompressionScheduler)	由compression schedule定义文件***.yaml构建的CompressionScheduler对象	in

编程示例

```
compression_scheduler.on_minibatch_begin(epoch)
output = self.model(data)                    # 模型inferecne
loss = self.criterion(output, target)    # 计算loss
compression_scheduler.before_backward_pass(epoch)
loss.backward()                              # 反向计算梯度
self.optimizer.step()                        # 根据梯度优化权重
compression_scheduler.on_minibatch_end(epoch)
```

2.2.3.3 Model Save

> save()

简要描述

- 保存训练模型

备注

- 用的是torch.save()方式保存模型, inference时只用torch.load()就可以还原模型, 不用重新构建模型结构
- 要注意, 在inference时, 定义模型结构的model.py文件要保留在trainging phase时相同的位置

2.2.3.4 Run

> run()

简要描述

- Trainer的主函数, 控制training的流程, 包括上面提到的Build Model、Train Process、Model Save等, 具体的过程可以看下面的编程示例

编程示例

```
self.build_model()  # 构建模型
compression_scheduler = distiller.config.file_config(self.model, self.optimizer,
                                      self.compress)   # 构建compression scheduler
compression_scheduler.on_epoch_begin(epoch)
self.train(epoch, compression_scheduler)   # Train Process
self.validate(self.model, epoch)           # Validation Process, not necessary
compression_scheduler.on_epoch_end(epoch)
self.save()                       # 保存模型
```

2.2.4 Tester

CLASS XSRCNNTester*(config, testing_loader)*

基类

- `object`

定义位置

- `./source/test_solver.py(34)`

简要描述

- SR 神经网络测试过程(C2SRCNN, SRCNN的变体)

参数列表

参数名	必选	类型	说明	in/out
config	Yes	dict	参数列表	in
testing_loader	Yes	class torch.utils.data.DataLoader()	测试集DataLoader	in

调用例程

```
from models.test_solver import XSRCNNTester
model = XSRCNNTester(args, test_data_loader)
model.run(args)
```

成员函数说明

2.2.4.1 Build Model

> build_model*()*

简要描述

- 构建模型, 从已经存储的权重文件.pth还原模型, 并定义评估metric(比如l2 loss)
- SR 应用目前不支持load_dict()方式还原模型, 如有需要, 可以参考 `models.Deblur.test_solver.build_model`

2.2.4.2 Test Process

> test*(model)*

简要描述

- Test Process, 测试过程, 主要包括正向Inferece、计算测试集loss以及将输出保存为图像文件, 具体的过程可以看下面的编程示例

参数列表

参数名	必选	类型	说明	in/out
model	Yes	class models.C2SRCNN.model.Net	已训练好的模型	in

编程示例

```
output = self.model(data)                          # 模型inferecne
mse = self.criterion(prediction, target)           # 测试集计算模型loss
self.save_tensor2img(output, output_save_path)     # 将模型输出存储为图像文件
```

2.2.4.3 Run

> run*(opt)*

简要描述

- Tester的主函数, 控制testing的流程, 包括上面提到的Build Model、Test Process等, 以及利用工具(参见Tools Collection)去可视化网络、统计计算量等, 都是在这里进行, 具体的过程可以看下面的编程示例

参数列表

参数名	必选	类型	说明	in/out
opt	Yes	dict	参数列表	in

编程示例

```
from apputils.platform_summaries import *
self.build_model()          # 构建模型
self.test(self.model)       # 测试模型
draw_model_to_file(model, 'arch.png', torch.FloatTensor(1, 3, 128, 128))  # 画模型结构
sensitivity_analysis(model, 'sense_file.xlsx', test_func, 'element')      # 权重敏感度分析
sparsity_display(model, 'spars_file.xlsx')                                # 权重稀疏性分析
macs_display(model, 'macs_file.xlsx', torch.FloatTensor(1, 3, 128, 128)) # 模型计算量估计
transform_to_onnx(model, 'model.onnx', torch.FloatTensor(1, 3, 128, 128), False) # 将模型和权重存为onnx格式文件
```

2.3.5 Compression Scheduler

Compression Scheduler指的是完成Network Compression流程管理, 包括AGP Pruning, Filter Pruning, Knowledge Distiller, Network Quantization等, Network Compression流程记录在***.yaml文件中, 在Training Phase开始的时候由这个yaml文件构建`class distiller.scheduler.CompressSceduler`对象, 并且在Traning Phase的不同阶段调用不同的函数来完成Network Compression的不同操作, 不论是哪种Network Compression方法, 外部调用的接口都是一样的, CompressionScheduler会根据在yaml中的定义来调用distiller库中不同的函数来实现具体的Network Compression算法

在这个章节中, 我们会分两部分来介绍使用distiller库的方法, 一方面是在Training Phase中要调用的接口函数, 另一方面是如何撰写yaml来在训练过程中加入不同的Network Compression算法

2.3.5.1 Distiller APIs

distiller.config.file_config*(model, optimizer, filename)*

定义位置

- `./distiller/config.py(143)`

简要描述

- Read the compression schedule from file

参数列表

参数名	必选	类型	说明	in/out
model	Yes	class models.C2SRCNN.model.Net	模型结构	in
optimizer	Yes	class torch.optim.optimizer.optimizer	训练时的优化器	in
filename	Yes	str	yaml文件路径	in

返回参数说明

参数名	类型	说明
compression_scheduler	class distiller.scheduler.CompressSceduler	Network CompressionScheduler 对象

调用例程

```
import distiller
distiller.config.file_config(model, optimizer, './models/C2SRCNN/agp_prune.yaml')
```

distiller.scheduler.CompressionScheduler.on_epoch_begin*(epoch, optimizer)*

定义位置

- `./distiller/scheduler.py(105)`

简要描述

- 在一个epoch开始的时候进行的compression schedule操作, 依compression的方法不同而不同

参数列表

参数名	必选	类型	说明	in/out
epoch	Yes	int	epoch id, 当前是training过程的第几个epoch	in
optimizer	No	class torch.optim.optimizer.optimizer	训练时的优化器(defualt:None)	in

调用例程

参考Trainer.Run

distiller.scheduler.CompressionScheduler.on_minibatch_begin*(epoch, minibatch_id, minibatches_per_epoch, optimizer)*

定义位置

- `./distiller/scheduler.py(121)`

简要描述

- 在一个batch开始的时候进行的compression schedule操作, 依compression的方法不同而不同

参数列表

参数名	必选	类型	说明	in/out
epoch	Yes	int	epoch id, 当前是training过程的第几个epoch	in
minibatch_id	Yes	int	batch id, 当前是某个epoch中的第几个batch	in
minibatches_per_epoch	Yes	int	一个epoch中的batch数量	in
optimizer	No	class torch.optim.optimizer.optimizer	训练时的优化器(defualt:None)	in

调用例程

参考Trainer.Run

distiller.scheduler.CompressionScheduler.before_backward_pass*(epoch, minibatch_id, minibatches_per_epoch, loss, optimizer, return_loss_components)*

定义位置

- `./distiller/scheduler.py(138)`

简要描述

- 在计算loss之后进行的compression schedule操作, 依compression的方法不同而不同因为做了network compression之后, loss会变化, 用这个操作去更新loss的值

参数列表

参数名	必选	类型	说明	in/out
epoch	Yes	int	epoch id, 当前是training过程的第几个epoch	in
minibatch_id	Yes	int	batch id, 当前是某个epoch中的第几个batch	in
minibatches_per_epoch	Yes	int	一个epoch中的batch数量	in
loss	Yes	float	未compression之前的loss值	in
optimizer	No	class torch.optim.optimizer.optimizer	训练时的优化器(defualt:None)	in
return_loss_components	No	bool	是否将compression前后的loss都显示出来(default:False)	in

调用例程

参考Trainer.Run

distiller.scheduler.CompressionScheduler.on_minibatch_end*(epoch, minibatch_id, minibatches_per_epoch, optimizer)*

定义位置

- `./distiller/scheduler.py(174)`

简要描述

- 在一个batch结束时进行的compression schedule操作, 依compression的方法不同而不同

参数列表

参数名	必选	类型	说明	in/out
epoch	Yes	int	epoch id, 当前是training过程的第几个epoch	in
minibatch_id	Yes	int	batch id, 当前是某个epoch中的第几个batch	in
minibatches_per_epoch	Yes	int	一个epoch中的batch数量	in
optimizer	No	class torch.optim.optimizer.optimizer	训练时的优化器(defualt:None)	in

调用例程

参考Trainer.Run

distiller.scheduler.CompressionScheduler.on_epoch_end*(epoch, optimizer)*

定义位置

- `./distiller/scheduler.py(199)`

简要描述

- 在一个epoch结束时进行的compression schedule操作, 依compression的方法不同而不同

参数列表

参数名	必选	类型	说明	in/out
epoch	Yes	int	epoch id, 当前是training过程的第几个epoch	in
optimizer	No	class torch.optim.optimizer.optimizer	训练时的优化器(defualt:None)	in

调用例程

参考Trainer.Run

2.3.5.2 Compression Scheduler yaml

distiller库中有许多Network Compression的方法, 包括Pruning、Regularization、Quantization、Knowledge Distillation以及Conditional Computation等(参考Neural Network Distiller), 这些方法都可以以yaml文件的形式从外部定义, 由distiller库解析yaml文件来选择具体算法执行, 我们主要介绍AGP Pruning, Filter Pruning和DorefaNet Quantization三种算法以及如何在yaml中加入这些算法

- Automated Gradual Pruner(AGP)

AGP Pruning的算法原型是To prune, or not to prune: exploring the efficacy of pruning for model compression, 基本思路是通过level pruner的手段达到weighting的稀疏化, 所谓level pruner就是指定要稀疏化的比例, 然后根据weighting的绝对值来决定哪些weighting的作用小, 直接置0, 下图说明了在pruning前后的weighting值分布的变化

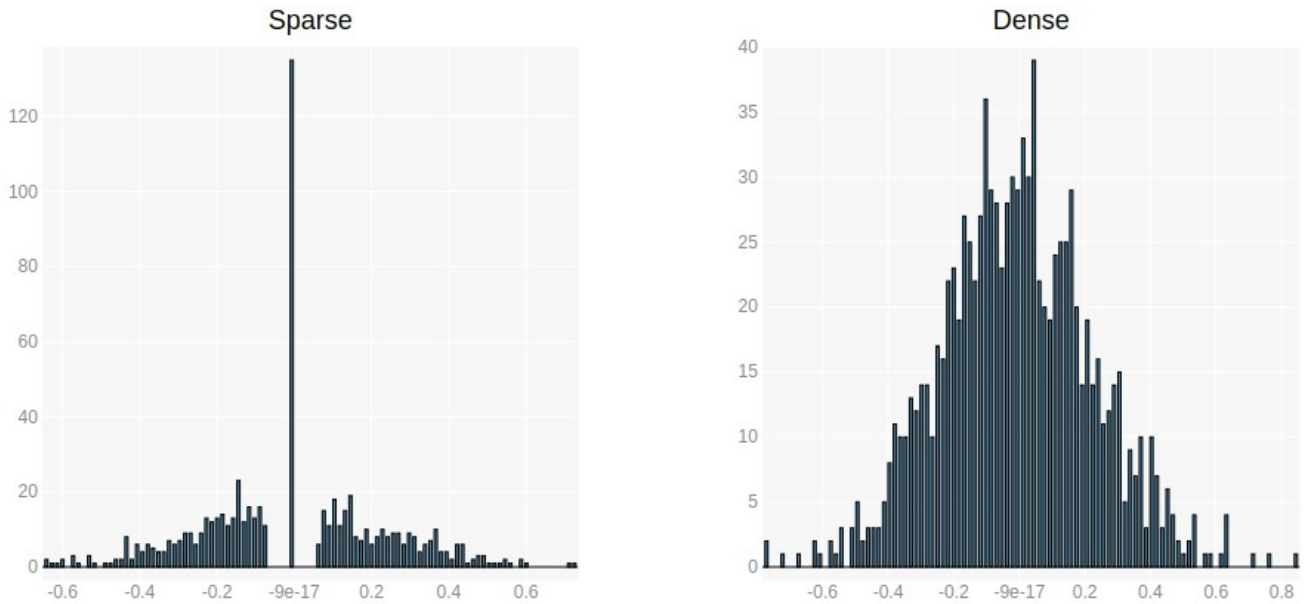

Figure 4. Weighting Distribution between sparsity & dense model

就是说, level pruner就是把值特别小的weighting直接置0, 达到非结构性稀疏化weighting数据的效果. 而Automated Gradual的意思是不会一步就稀疏到指定的稀疏化程度, 而是逐步增加稀疏化的程度, 每一个步增加的稀疏比例按指数递减, 如下公式所示, 其中s_i表示初始的稀疏化比例, s_f表示最终要达到的稀疏化比例, s_t表示每一步新增的稀疏比例

$$s_t = s_f + (s_i - s_f)\left(1 - \frac{t - t_0}{n\Delta t}\right)^3 \quad \text{for} \quad t \in \{t_0,\ t_0 + \Delta t,\ \ldots,\ t_0 + n\Delta t\}$$

Automated Gradual Pruning在实际操作的过程中是稀疏过程和训练过程迭代, 一边稀疏一边训练, 逐步达到预设的稀疏化目标, 基本的流程如下图所示

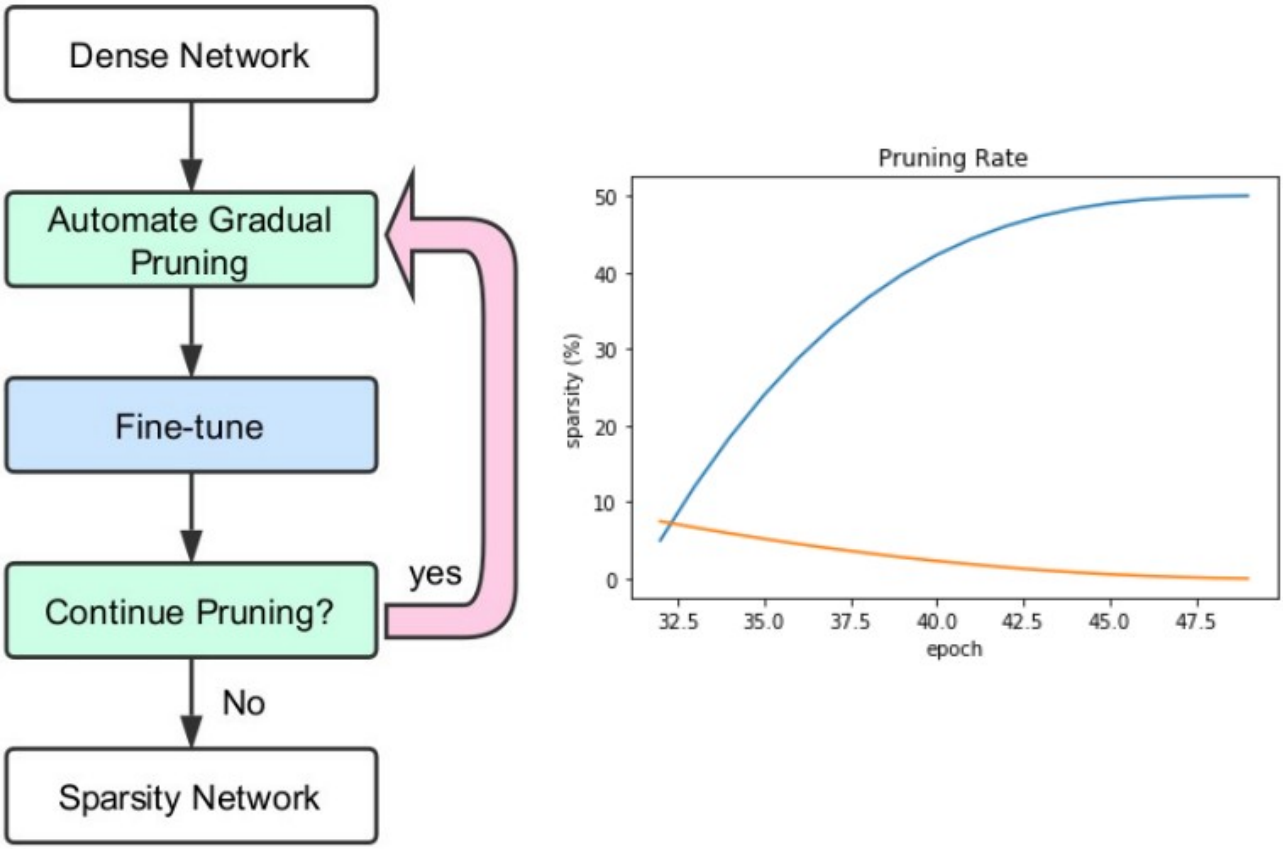

Figure 5. AGP Pruning Scheme

右侧蓝色曲线表明在迭代过程中权重Sparsity在逐步提高, 而黄色曲线表明每一步稀疏较前一次增加的稀疏比例, 逐渐降低, 当到达目标稀疏比例s_f后黄色曲线会降低到0

完成AGP Pruner的yaml定义如下所示

```
version: 1
pruners:
  conv_pruner:
    class: 'AutomatedGradualPruner'
    initial_sparsity : 0.05
    final_sparsity: 0.4
    weights: ['mid_part.0.0.weight', 'mid_part.1.weight', 'mid_part.3.weight',
'mid_part.5.weight', 'mid_part.7.weight', 'mid_part.9.0.weight']

policies:
  - pruner:
      instance_name : 'conv_pruner'
    starting_epoch: 0
    ending_epoch: 300
    frequency: 1
```

`pruners`定义了pruner的具体算法, `AutomatedGradualPruner`是pruner具体执行的类, `initial_sparsity`和`final_sparsity`分别是初始稀疏比例和目标稀疏比例, 而`weights`定义了要稀疏的权重的名称, 这个可以在定义模型的时候具体看每个module权重的名称是什么, 要稀疏哪些module就在这里填什么

`policies`是在训练过程中执行pruner的具体方法, 程序会在那些compression scheduler的操作(比如前面说的`on_epoch_begin()`、`on_minibatch_begin()`等)中选择你定义的pruner类来调用具体的函数, 并且规定在训练的哪些部分执行这些pruner操作, 在这个例子中就是从0 epoch开始到300 epoch结束, 每个epoch都执行pruner操作, 300 epoch之后的训练部分就不会执行pruner操作, 而继续普通的训练过程

- Filter Pruning

Filter Pruning的算法原型是Pruning Filters for Efficient ConvNets, 和上面提到的AGP Pruninng相比, Filter Pruning同样是根据权重的绝对值来稀疏化, 但是稀疏时不是逐个权重参数去判断权重的重要性, 而是以filter作为整体结构来判断权重的重要性, 具体方法是求一个filter包含的所有参数的L1 norm, 所以这个方法又叫L1 Ranked Structure Parameter Pruner

$$W_{importance} = /left/|W/right/|_{l1}$$

和AGP Pruner不同, Filter Pruner是一步根据目标稀疏比例砍掉importance比较低的filter, 然后根据砍掉的filter去给model做瘦身(thinning), 就是从结构上去掉这些filter, 比如某个layer的weighting dims是[64, 3, 3, 3], filter pruning的目标是25%, 那经过thinning后, 这个layer的weighting dims就变成[48, 3, 3, 3], 因为砍掉了25%的filter, 所以output channel也就相应少掉了25%. thinning之后, 再进行fine tune, 基本的流程如下图所示

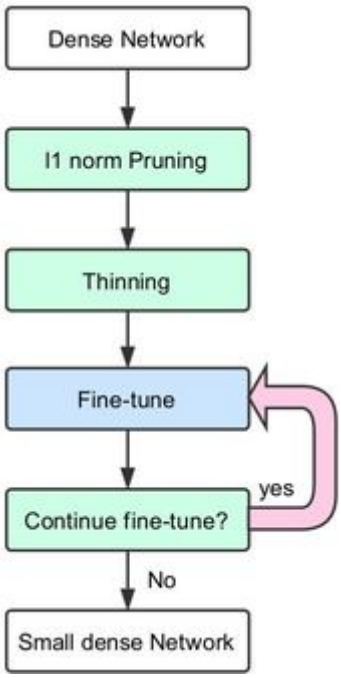

Figure 6. Filter Pruning Scheme

完成Filter Pruner的yaml定义如下所示:

```
version: 1
pruners:
  filter_pruner:
    class: 'L1RankedStructureParameterPruner'
    reg_regims:
      'mid_part.0.0.weight': [0.4, '3D']
      'mid_part.1.weight': [0.4, '3D']
      'mid_part.3.weight': [0.4, '3D']
      'mid_part.5.weight': [0.4, '3D']
```

```
      'mid_part.7.weight': [0.4, '3D']
      'mid_part.9.0.weight': [0.4, '3D']

extensions:
  net_thinner:
      class: 'FilterRemover'
      thinning_func_str: remove_filters
      arch: 'vgg19'
      dataset: 'imagenet'

policies:
  - pruner:
      instance_name: 'filter_pruner'
    epochs: [0]

  - extension:
      instance_name: 'net_thinner'
    epochs: [0]
```

`L1RankedStructureParameterPruner`是pruner最终具体要执行的类, 而在filter pruner的应用中, 权重的名称写在`reg_reims`下面, `0.4`同样是最终要稀疏化的目标比例, 而`3D`表明了结构化稀疏的方式, 目前`L1RankedStructureParameterPruner`支持两种结构化稀疏方式, `3D` -> filter wise sparsity / `Channels` -> input channel wise sparsity.

`extensions`定义了pruner之外的操作, 在filter pruner的应用里, 就是thinning, 就是从物理结构上去掉全部是0的filter

`policies`同样是在训练过程中执行pruner的具体方法, 可以看到, pruner和thinning操作在epoch 0里就全部完成了, 就是说, filter pruner是不需要与训练过程迭代交互的, 一步完成, 剩下的就是传统的fine-tune方式去继续训练prunning之后的模型

- DorefaNet Quantization

量化(Quantization)过程是指将权重(weighting)和激活值(activations)都转为指定位宽, 同时尽量维持model的performance, 这里, 我们用的是[DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients]()方法, 是将量化与训练结合的方法, 通过在训练中将performance的损失由其他weighting共同承担的方法来维持model performance不下降太多, 基本的思路如下图所示

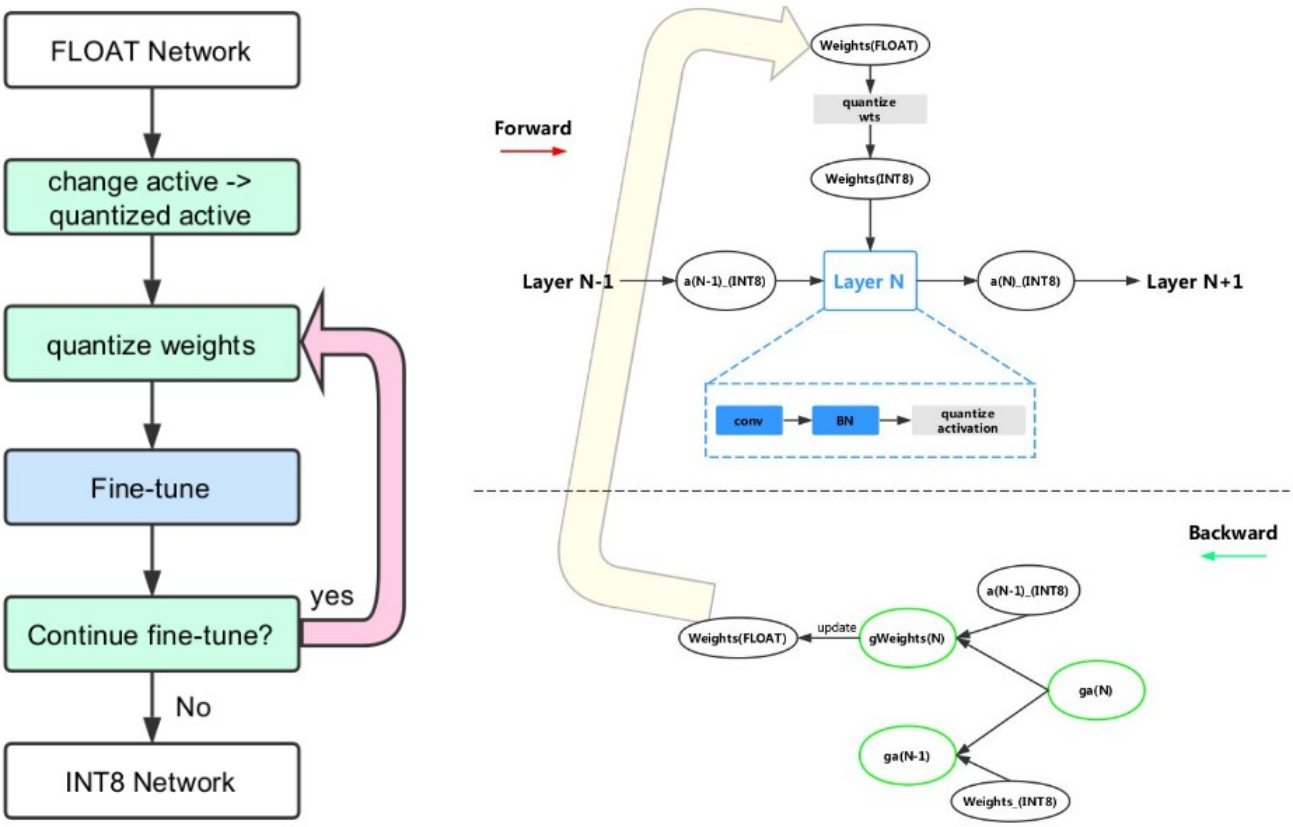

Figure 7. DorefaNet Quantization Scheme

左边的图是Quantization的流程图, 表明quantization过程是和training过程交替迭代的, 如果选择进行DorefaNet Quantization, 那在Compression Scheduler定义的那些操作中, 就会去找Quantization需要在`on_epoch_begin`、`on_minibatch_begin`等阶段需要调用的函数接口从而实现Quantizationsuanfa

右图是分别在Inference和Backpropogation的过程中, DorefaNet Quantization各种操作的时机. 每一次的量化操作, 从weight_fp开始, 先对weight_fp量化, 然后forward(), 在forward()之前, 已经将激活函数改为quantize activation, 然后得到输出, backward()时, 用实际计算的weight_q去算∇a, 用之前forward()得到的a+1去算∇weight_q, 然后用∇weight_q计算∇weight_fp(因为要考虑量化函数的梯度weight_q=quantizek(weight_fp)), 更新weight_fp, 再到下一次iteration, 以此形成闭环;

完成DorefaNet Quantization的yaml定义如下所示:

```
version: 1
```

```
quantizers:
  dorefa_quantizer:
    class: 'DorefaQuantizer'
    bits_activations: 8
    bits_weights: 8
    quantize_bias: True

lr_schedulers:
   training_lr:
     class: 'MultiStepLR'
     milestones: [500,1000,1500,2000]
     gamma: 0.5

policies:
    - quantizer:
        instance_name: 'dorefa_quantizer'
      starting_epoch: 0
      ending_epoch: 2500
      frequency: 1

    - lr_scheduler:
        instance_name: 'training_lr'
      starting_epoch: 0
      ending_epoch: 2500
      frequency: 1
```

`DorefaQuantizer`是pruner最终具体要执行的类, `bits_activations`是激活值要量化的位宽, `bits_weights`是权重要量化的位宽, `quatntize_bias`是是否要量化Bias参数.

在DorefaNet Quantization的这个例子中, 新出现了一个item叫`lr_schedulers`, 这是通过yaml定义的方法来控制learning rate在训练过程中的变化, 和`pytorch`中`torch.optim.lr_scheduler.py`定义的各种Learning rate scheduler类似

`policies`同样是在训练过程中执行pruner的具体方法, 类似于AGP Pruner, 在`starting_epoch`和`ending_epoch`之间, 会进行quantization和training的交互迭代.

2.3.6 Tools Collections

2.3.6.1 Sensitivity Analysis

apputils.platform_summaries.sensitivity_analysis*(model, sensitivity_file, test_func, sense_type)*

定义位置

- `./apputils/platform_summaries.py(54)`

简要描述

- 分析网络权重的敏感度, 就是剪枝后的performance loss

参数列表

参数名	必选	类型	说明	in/out
model	Yes	class models.C2SRCNN.model.Net	已训练好的模型	in
sensitivity_file	Yes	str	存储敏感度分析结果的文件路径	in
test_func	Yes	func	在Tester中定义的Test Process, 因为要得到performance, 需要每次剪枝后inference一次model	in
sense_type	Yes	str	敏感度分析的维度(one of 'element'/'filter'/'channel')	in

调用例程

```
from apputils.platform_summaries import sensitivity_analysis
sensitivity_analysis(model, 'sense_file.xlsx', test_func, 'element')
```

2.3.6.2 Sparsity Display

apputils.platform_summaries.sparsity_display*(model, sparsity_file)*

定义位置

- `./apputils/platform_summaries.py(79)`

简要描述

- 分析网络权重的稀疏性, 就是权重中0的占比

参数列表

参数名	必选	类型	说明	in/out
model	Yes	class models.C2SRCNN.model.Net	已训练好的模型	in
sensitivity_file	Yes	str	存储稀疏性分析结果的文件路径	in

调用例程

```
from apputils.platform_summaries import sparsity_display
sparsity_display(model, 'spars_file.xlsx')
```

2.3.6.3 MACs Display

apputils.platform_summaries.macs_display*(model, macs_file, dummy_input)*

定义位置

- `./apputils/platform_summaries.py(98)`

简要描述

- 分析网络的计算量MACs

参数列表

参数名	必选	类型	说明	in/out
model	Yes	class models.C2SRCNN.model.Net	已训练好的模型	in
sensitivity_file	Yes	str	存储计算量结果的文件路径	in
dummy_input	Yes	tensor	和网络输入相同维度的dummy数据, for example, torch.FloatTensor(1, 3, 32, 32), [batch, channel, height, width]	in

调用例程

```
from apputils.platform_summaries import macs_display
macs_display(model, 'macs_file.xlsx',  torch.FloatTensor(1, 3, 32, 32))
```

2.3.6.4 Draw Model

apputils.platform_summaries.draw_model_to_file*(model, png_fname, dummy_input)*

定义位置

- `./apputils/platform_summaries.py(29)`

简要描述

- 将模型结构存储到png图像文件

参数列表

参数名	必选	类型	说明	in/out
model	Yes	class models.C2SRCNN.model.Net	已训练好的模型	in
png_fname	Yes	str	图像文件路径	in
dummy_input	Yes	tensor	和网络输入相同维度的dummy数据, for example, torch.FloatTensor(1, 3, 32, 32), [batch, channel, height, width]	in

调用例程

```
from apputils.platform_summaries import draw_model_to_file
draw_model_to_file(model, 'arch.png', torch.FloatTensor(1, 3, 32, 32))
```

备注

- `draw_model_to_file()`只能在程序执行过程中, 画出`model`定义的模型到目标图像文件, 还有一个工具 Netron, 可以直接拖动相应的模型文件(.pth/.onnx/.prototxt/.pb)等到界面上, 画出的模型结构比`draw_model_to_file()`更清晰并且可以获得更多信息, 并支持基本所有的框架生成的模型文件, 在开发过程中, 可视化模型框架, 也可以直接选用这个工具

2.3.6.5 Transform Network to ONNX format

apputils.platform_summaries.transform_to_onnx*(model, onnx_file, dummy_input, quantize_flag)*

定义位置

- `./apputils/platform_summaries.py(119)`

简要描述

- 将网络模型和权重存储为onnx格式

参数列表

参数名	必选	类型	说明	in/out
model	Yes	class models.C2SRCNN.model.Net	已训练好的模型	in
onnx_file	Yes	str	onnx格式权重文件路径	in
dummy_input	Yes	tensor	和网络输入相同维度的dummy数据, for example, torch.FloatTensor(1, 3, 32, 32), [batch, channel, height, width]	in
quantize_flag	Yes	book	模型是否为量化后模型, 如果是, 需要在存onnx模型时将ClippedLinearQuantization module替换为ClippedOnly module	in

调用例程

```
from apputils.platform_summaries import transform_to_onnx
transform_to_onnx(model, 'model.onnx',  torch.FloatTensor(1, 3, 32, 32), False)
```