Contents

[Contents](#_Toc1539910139)

[常用工具](#_Toc1615440655)

[命令](#_Toc107622010)

[标注工具](#_Toc2014657552)

[公开数据集](#_Toc699364361)

[图形化DL训练工具DIGITS](#_Toc1756159679)

[硬件及平台](#_Toc670405241)

[GPU](#_Toc949966398)

[GPU SDK](#_Toc191874932)

[CUDA核数](#_Toc771217819)

[内存大小](#_Toc2121560112)

[deploy优化](#_Toc33573601)

[TensorRT](#_Toc1887871772)

[基本概念](#_Toc1056105144)

[优化方法：](#_Toc1047862961)

[1. 网络优化](#_Toc1474633072)

[Typical development workflow:](#_Toc1939491988)

[安装](#_Toc2113926161)

[方法1: Command Line Wrapper](#_Toc197278520)

[方法2: python接口](#_Toc1544403418)

[方法2.1: TensorRT Lite （高度抽象层）](#_Toc1890150487)

[方法2.2: Using TensorRT to Optimize Caffe Models in Python](#_Toc584106716)

[方法3： C++接口 （主要用于自定义网络）](#_Toc767940974)

[方法4： TensorRT + Caffe](#_Toc751010058)

[模型压缩](#_Toc1029103042)

[模型剪枝 （有损）](#_Toc803897453)

[模型量化 （有损）](#_Toc621493579)

[Deep learning framework](#_Toc516953161)

[Caffe](#_Toc2002910870)

[Install](#_Toc1227064845)

[tutorial](#_Toc2041290369)

[Blob storage and communication](#_Toc1395337361)

[Layer computation and connections](#_Toc695021852)

[Net = layers + layers + ...](#_Toc1428731)

[Model](#_Toc1262511265)

[Solver](#_Toc1394386214)

[可视化.prototxt](#_Toc1757588410)

[Interfaces--Command Line (train, test, time)](#_Toc1932916507)

[Interfaces--Python](#_Toc196868964)

[安装](#_Toc1949463343)

[1. 用代码来生成train.prototxt, test.prototxt, solver.prototxt文件](#_Toc556650678)

[2. 剖析模型](#_Toc170945429)

[3. 模型Finetune](#_Toc1983036944)

[4. 训练--绘制loss and accuracy曲线](#_Toc297038802)

[5. 测试](#_Toc1227050573)

[Interfaces--C++](#_Toc883416258)

[Caffe源代码](#_Toc1771671875)

[Caffe IDE](#_Toc1019058913)

[Caffe的整体流程图](#_Toc849858771)

[Caffe test case调试](#_Toc1968950395)

[caffe.proto](#_Toc415978683)

[Blobs](#_Toc592525611)

[Layers](#_Toc405573464)

[自定义层](#_Toc1183919657)

[实现涉及的文件](#_Toc1343535669)

[全通过层AllPassLayer(仅仅作为教学)](#_Toc1434676506)

[标注数据层AnnotationDataLayer](#_Toc1987817110)

[Caffe2](#_Toc1965029249)

[Install](#_Toc1951629668)

[Tutorial](#_Toc1843244332)

[Blob：存储数据](#_Toc1044610446)

[Workspace：工作空间(等效matlab, caffe中没有),](#_Toc1845436389)

[Operator：输入blob，输出blob，定义了计算规则](#_Toc1091098046)

[Caffe2 Operator与Caffe Layer区别？](#_Toc1739632299)

[Net：由多个operator组合实现](#_Toc1846865120)

[手写体分类](#_Toc206125663)

[A. 训练](#_Toc986534865)

[B. 测试](#_Toc1456969883)

[C. deploy](#_Toc2139042170)

[剖析模型](#_Toc1183403829)

[Caffe2 DB](#_Toc1258949578)

[minidb](#_Toc548209200)

[Keras](#_Toc1354349258)

[install](#_Toc1094502874)

[tutorial](#_Toc845248003)

[张量Tensor](#_Toc433916183)

[层Layer](#_Toc1977919132)

[模型= Sequential模型 / 函数式模型Model](#_Toc469436230)

[数据预处理](#_Toc1452975096)

[性能评估](#_Toc680294256)

[权重初始化，正则项，约束项](#_Toc290902977)

[训练过程的网络状态--通过回调函数Callbacks](#_Toc1868953779)

[Key points](#_Toc1272819867)

[Keras后端](#_Toc696476441)

[自定义层](#_Toc905389788)

[tensorflow](#_Toc468871888)

[Install](#_Toc2131152948)

[High Level APIs](#_Toc745723251)

[使用Estimators(模型)](#_Toc286417489)

[Estimators <- Keras models](#_Toc1935298968)

[自定义Estimators](#_Toc441483935)

[保存和恢复模型 Estimators](#_Toc1331027936)

[数据](#_Toc1633251709)

[Low Level APIs --TensorFlow Core](#_Toc1532581981)

[Tensor (含属性：data type and shape)](#_Toc923176587)

[节点Op](#_Toc1332633181)

[计算图tf.Graph](#_Toc1738707645)

[运行计算图（tf.Session）](#_Toc1909711452)

[TensorBoard: 可视化学习](#_Toc642119416)

[保存和加载模型](#_Toc1730266167)

[分类](#_Toc945631633)

[例子：手写体](#_Toc1901068994)

常用工具

命令

Monitor GPU resource

$ watch -n 1 nvidia-smi

标注工具

ROI

<https://github.com/tzutalin/labelImg>

sudo apt-get install pyqt4-dev-tools

sudo pip install lxml

$git clone <https://github.com/tzutalin/labelImg.git>

labelImg$ make qt4py2

labelImg$ python labelImg.py

导入图像文件夹 -> Create ROI -> drag mouse to draw ROI -> Ctrl+s

xml 转json

<https://pypi.python.org/pypi/xmljson>

$ pip install xmljson

from xmljson import parker, Parker

from xml.etree.ElementTree import fromstring, fromstringlist

with open(root + '/Case-3-D-6-1.xml', 'r') as fh:

contents = fh.readlines()

anno = parker.data(fromstringlist(contents)) #json数据

print(json.dumps(anno, indent=4))

another lib

import xmltodict

json\_dict = xmltodict.parse(xml\_str)

xmlstr = xmltodict.unparse(json\_dict)

分动分割

<https://github.com/wkentaro/labelme>

# Ubuntu 14.04

sudo apt-get install python-qt4 pyqt4-dev-tools

sudo pip install labelme # python2 works

公开数据集

Pascal VOC <http://host.robots.ox.ac.uk/pascal/VOC/>

coco <http://cocodataset.org/#home>

kitti <http://www.cvlibs.net/datasets/kitti/eval_object.php>

openimages <https://github.com/openimages/dataset>

图形化DL训练工具DIGITS

<https://github.com/NVIDIA/DIGITS>

深度学习GPU训练系统，支持Caffe, Torch and Tensorflow

硬件及平台

GPU

**Tesla** 人工智能，高性能计算和超大规模数据中心

Tesla V100（16/32GB），P100, P4/P40

**GEFORCE GTX** 游戏体验

**GTX 1080 Ti**， 1080， 1070， 1060, 1050

NVIDIA CUDA® Cores:3584 Boost Clock (MHz): 1582

Memory Speed: 11 Gbps Standard Memory Config:11 GB GDDR5X

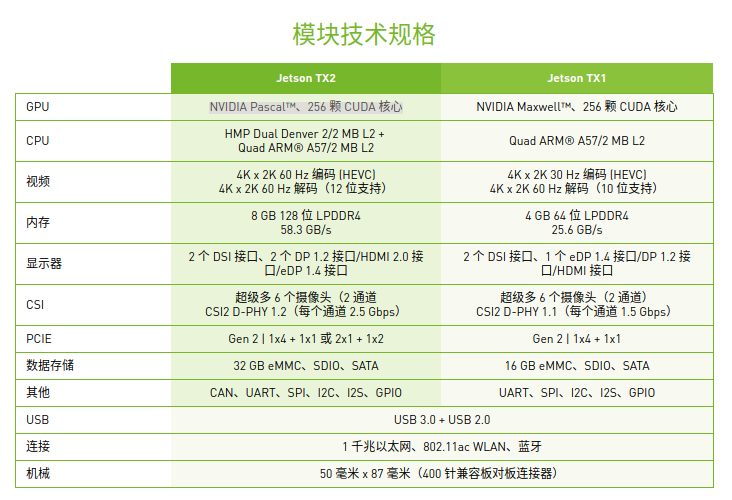
Memory Interface Width:352-bit Memory Bandwidth (GB/sec): 484

Maximum Digital Resolution:7680x4320@60Hz

Maximum GPU Temperature (in C): 91 Graphics Card Power (W):250 W

**JETSON TX2**

性能强大，但外形小巧，节能高效，非常适合机器人、无人机、智能摄像机和便携医疗设备等智能终端设备。



Jetson Development Pack (JetPack)

一体化软件包，捆绑并安装了适用于 NVIDIA Jetson 嵌入式平台的所有开发用软件工具。 JetPack 包括:

* 深度学习: TensorRT、cuDNN、NVIDIA DIGITS™ 工作流程
* 计算机视觉: NVIDIA VisionWorks、OpenCV
* GPU 计算: NVIDIA CUDA、CUDA 库
* 多媒体: ISP 支持、摄像头图像、视频 CODEC
* 同时，它还包括 ROS 兼容性、OpenGL、高级开发者工具等等。
* support for TensorFlow models, up to 15% perf/W improvement for DL applications, out-of-the-box kernel support for Docker, and support for Ubuntu 16.04 on your host PC.

2018京东价格：

Jetson TX2 核心板： 3799元

Jetson TX1 TX2 嵌入式开发套件 TX2套件： 5580

Quadro 用于专业绘图设计

Tegra 专为移动和嵌入式设备设计

NVIDIA® NVS™ 多显示器商用显卡的标准

架构：

2017 NVIDIA Volta 架构: 如Tesla V100

2016 NVIDIA Pascal 架构： 如GTX 10XX系列， JETSON TX2

2014 NVIDIA Maxwell 架构： 如GTX 9XX系列

2012 NVIDIA Kepler™ 架构： 如GTX 7XX，

2010 NVIDIA Fermi架构

GeForce GTX 1080 vs Tesla P100

<https://alisha17.github.io/machine-learning/2017/12/15/benchmarks.html>

Larger mini-batches are more efficient to compute and lead to better convergence in fewer epochs

a larger memory in a GPU and take a larger batch size, donot mean that it will give better validation accuracy

the time taken for training with a larger batch also increases significantly

2000 steps and the batch size as 32 was 51 minutes, the time taken for training with same number of steps but with the batch size as 85, was 120 minutes

GPU SDK

CUDA核数

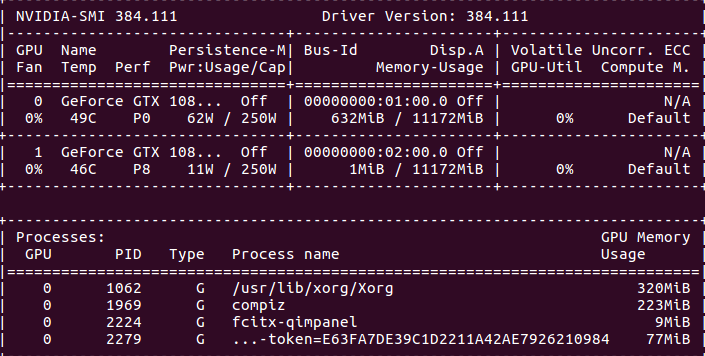
内存大小

峰值计算性能 TFLOPS： 万亿次浮点指令每秒

1T=1024G， floating point operations per second

内存带宽

$ watch -n 1 nvidia-smi



第一栏的Fan：N/A是风扇转速，从0到100%之间变动，这个速度是计算机期望的风扇转速，实际情况下如果风扇堵转，可能打不到显示的转速。有的设备不会返回转速，因为它不依赖风扇冷却而是通过其他外设保持低温（比如我们实验室的服务器是常年放在空调房间里的）。

第二栏的Temp：是温度，单位摄氏度。

第三栏的Perf：是性能状态，从P0到P12，P0表示最大性能，P12表示状态最小性能。

第四栏下方的Pwr：是能耗，上方的Persistence-M：是持续模式的状态，持续模式虽然耗能大，但是在新的GPU应用启动时，花费的时间更少，这里显示的是off的状态。

第五栏的Bus-Id是涉及GPU总线的东西，domain:bus:device.function

第六栏的Disp.A是Display Active，表示GPU的显示是否初始化。

第五第六栏下方的Memory Usage是显存使用率。

第七栏是浮动的GPU利用率。

第八栏上方是关于ECC的东西。

第八栏下方Compute M是计算模式。

下面一张表示每个进程占用的显存使用率。

显存占用和GPU占用是两个不一样的东西，显卡是由GPU和显存等组成的，显存和GPU的关系有点类似于内存和CPU的关系。我跑caffe代码的时候显存占得少，GPU占得多，师弟跑TensorFlow代码的时候，显存占得多，GPU占得少。

cuda 版本

cat /usr/local/cuda/version.txt

cudnn 版本

cat /usr/local/cuda/include/cudnn.h | grep CUDNN\_MAJOR -A 2

deploy优化

主要指标：Power efficiency and speed of response

TensorRT

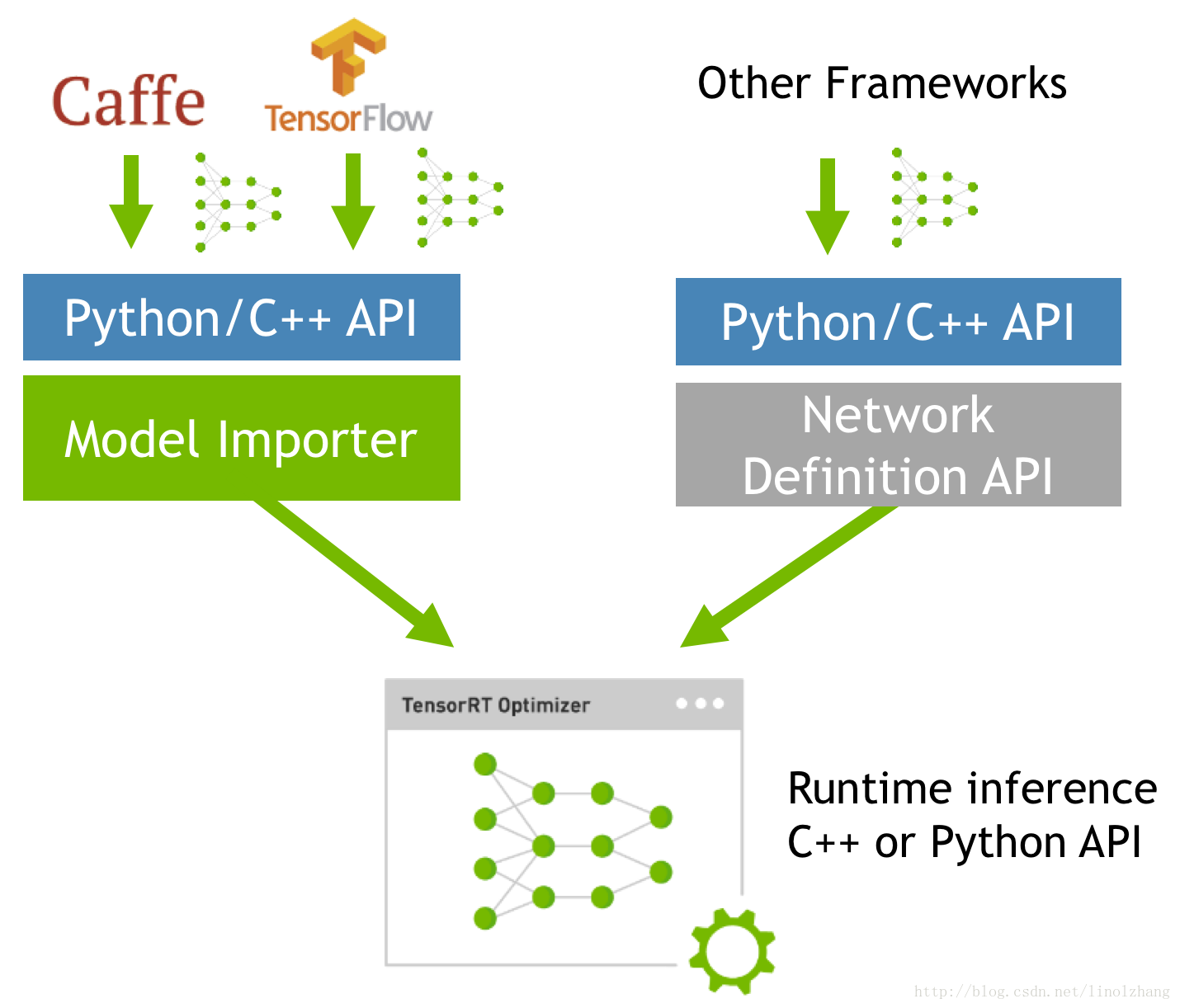
<https://developer.nvidia.com/tensorrt>

<https://blog.csdn.net/fengbingchun/article/details/78469551>

基本概念

TensorRT是为优化生产环境中部署的深度学习模型而创建的库(C++库), 自动优化训练过的神经网络。使用TensorRT，你无需在部署硬件上安装并运行深度学习框架。

TensorRT 4 provides an ONNX parser so you can easily import ONNX models from frameworks such as Caffe 2, Chainer, Microsoft Cognitive Toolkit, MxNet and PyTorch into TensorRT



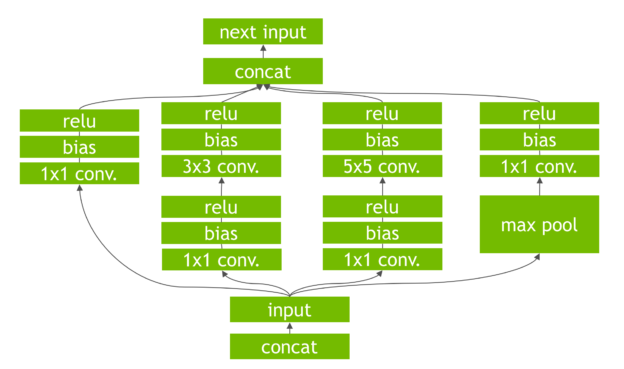
术语

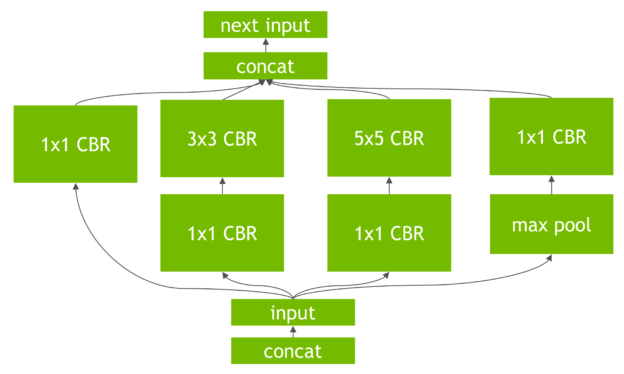
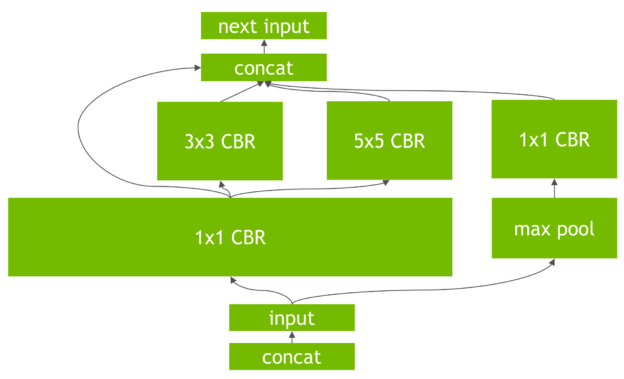
* **ONNX** (Open Neural Network Exchange)
* **UFF (Universal Framework Format )**: is a data format that describes an execution graph for a DNN (Deep Neural Network)
* **PLAN file**: is the serialized data that the runtime engine uses to execute the network

优化方法：

1. 网络优化

* 消除未使用的输出的层以避免不必要的计算；
* 在可能的情况下，convolution、bias和ReLU层被融合以形成单个层，包括垂直层融合和水平层融合。

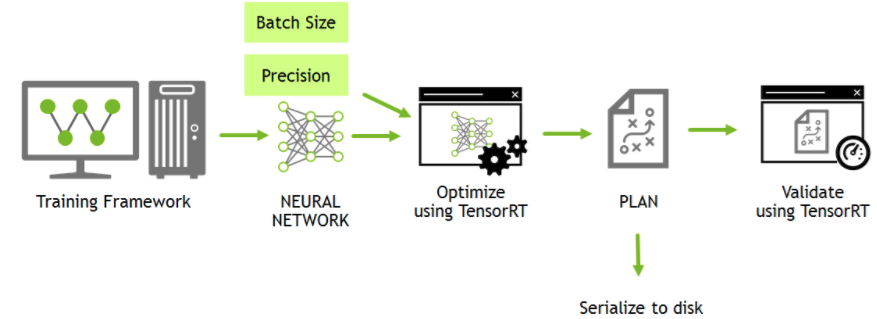


垂直层融合 水平层融合

1. 数据布局可以使用半精度FP16 和INT8 （有损）
2. 卷积的优化，使用Winograd（提升3倍） 等算法或者特定硬件方式实现；
3. cuDNN优化，根据不同的batchsize设置不同的计算模式或者GPU clock；
4. 内存优化,GPU汇编指令,提高GPU利用率(批量处理尽可能并行处理,在cuda中 使用warp对齐,提高GPU命令命中率)
5. CPU和GPU互补：CPU做一部分工作，GPU做一部分工作；

Typical development workflow:



two ways to create a TensorRT network:

1. Create from scratch using the Builder API, or
2. Import an existing NVCaffe, ONNX or Tensorflow network using the Parser API

TensorRT支持的层

Activation (ReLU,tanh,sigmoid), BatchGemm, Concatenation, Constant,

Convolution, Deconvolution, ElementWise, Flatten, FullyConnected, Gather,

LRN, MatrixMultiply, Padding, Pooling (maximum, average), Ragged SoftMax,

Reduce, RNN, RNNv2, Scale, Shuffle, SoftMax, Squeeze, TopK, Unary,

Plugin

The Plugin Layer allows you to integrate custom layer implementations that TensorRT does not natively support.

对于TensorRT 不支持的层，可以先将支持的层跑完，然后将输出作为caffe的输入，用caffe再跑，V1不支持TensorRT 和caffe同时工作，V2支持。

**Some framework models are not supported by TensorRT**. How to do?

* use the Plugin API
* modify the original network to use an equivalent supported model.

The TensorRT API allows developers to import, calibrate, generate and deploy optimized networks. Networks can be imported directly from NVCaffe, or from other frameworks via the UFF or ONNX formats. They may also be created programmatically by instantiating individual layers and setting parameters and weights directly.

A workflow is provided for those looking to implement custom layers for use in Python in tensorrt/examples/custom\_layers.

Build Phase

* elimination of layers whose outputs are not used
* fusion of convolution, bias and ReLU operations
* aggregation of operations with sufficiently similar parameters and the same source tensor (for example, the 1x1 convolutions in GoogleNet v5's inception module)
* merging of concatenation layers by directing layer outputs to the correct eventual destination

Execution Phase

* The runtime executes the optimized engine.
* The engine runs inference tasks using input and output buffers on the GPU.

安装

<https://zhuanlan.zhihu.com/p/35468450>

<https://blog.csdn.net/xll_bit/article/details/78376320>

下载（链接见第一篇blog）

TensorRT 4.0.0.3 for Ubuntu 16.04 and CUDA 8.0 tar package

注意: 版本要求cuda 8 or 9, cudnn 7， 一定要细看doc

添加路径

$sudo gedit ~/.bashrc

export TensorRT\_Root="/home/qzlin/Documents/dl/TensorRT4"

export LD\_LIBRARY\_PATH="$TensorRT\_Root/lib:$LD\_LIBRARY\_PATH"

$source ~/.bashrc

TensorRT\_Root$pip install python/tensorrt\*.whl

$ which tensorrt 测试 tensorRT是否安装成功

若出现fatal error: cuda.h: No such file or directory

注意：不能加sudo, 具体见网上解释关于安装pycuda

$pip install uff-0.1.0rc0-py2.py3-none-any.whl

$ which convert-to-uff 测试convert-to-uff是否安装成功

1. Caffe -> TensorRT

方法1: Command Line Wrapper

giexec --deploy=mnist.prototxt

--model=mnist.caffemodel

--output=prob

方法2: python接口

**方法2.1: TensorRT Lite （高度抽象层）**

the Lite engine creates the logger, TensorRT engine, runtime and context for you, then allocates the GPU memory for the engine as well.

def build\_engine(engine\_file):

MODEL\_PROTOTXT = './data/mnist/mnist.prototxt'

CAFFE\_MODEL = './data/mnist/mnist.caffemodel'

**engine = trt.lite.Engine**(framework="caffe",

deployfile=MODEL\_PROTOTXT,

modelfile=CAFFE\_MODEL,

max\_batch\_size=10,

input\_nodes={"data": (1, 28, 28)},

output\_nodes=['prob'])

**engine.save(engine\_file)**

def infer(engine\_file):

**engine = trt.lite.Engine(PLAN=engine\_file)**

for i in range(10):

img, rand\_file = generate\_data()

img = np.reshape(img, (1, 1, 28, 28))

**output = engine.infer(img)[0]**

print("Test Case: " + str(rand\_file))

print ("Prediction: " + str(np.argmax(output)))

同样支持caffe, tensorflow, and other framework(要自己构建)

可以包含输入预处理和输出后处理

def normalize(data):

for i in range(len(data)):

data[i] = 1.0 - data[i] / 255.0

return data.reshape(1,28,28)

argmax = lambda res: np.argmax(res.reshape(10))

mnist\_engine = tensorrt.lite.Engine(framework="tf", #Source framework

path=DATA\_DIR + "/mnist/lenet5\_mnist\_frozen.pb", #Model File

max\_batch\_size=10, #Max number of images to be processed at a time

input\_nodes={"in":(1,28,28)}, #Input layers

output\_nodes=["out"], #Ouput layers

preprocessors={"in":normalize}, #Preprocessing functions

postprocessors={"out":argmax}) #Postprocesssing functions

方法2.2: Using TensorRT to **Optimize Caffe Models** in Python

# 模型 -> plan

import tensorrt as trt

G\_LOGGER = trt.infer.ConsoleLogger(trt.infer.LogSeverity.ERROR)

def create\_engine(engine\_file):

MODEL\_PROTOTXT = './data/mnist/mnist.prototxt'

CAFFE\_MODEL = './data/mnist/mnist.caffemodel'

engine = trt.utils.caffe\_to\_trt\_engine(G\_LOGGER,

MODEL\_PROTOTXT,

CAFFE\_MODEL,

1,

1 << 20,

['prob'],

trt.infer.DataType.FLOAT)

trt.utils.write\_engine\_to\_file(engine\_file, engine.serialize())

engine.destroy()

# 使用plan进行inference

PyCUDA handles the CUDA operations needed to allocate memory on your GPU and to transfer data to the GPU and results back to the CPU.

NumPy is a well used tool to store and move data.

from tensorrt.parsers import caffeparser

import pycuda.driver as cuda

import pycuda.autoinit

import numpy as np

from random import randint

def inputs():

# 导入图像，(28,28) -> (28\*28, 1)

DATA = '/data/mnist/'

rand\_file = randint(0,9)

path = DATA + str(rand\_file) + '.pgm'

im = Image.open(path)

arr = np.array(im)

img = arr.ravel()

print("Test Case: " + str(rand\_file))

# 去均值

IMAGE\_MEAN = '/data/mnist/mnist\_mean.binaryproto'

INPUT\_H = 28

INPUT\_W = 28

parser = caffeparser.create\_caffe\_parser()

mean\_blob = parser.parse\_binary\_proto(IMAGE\_MEAN)

parser.destroy()

mean = mean\_blob.get\_data(INPUT\_W \*\* 2)

data = np.empty([INPUT\_W \*\* 2])

for i in range(INPUT\_W \*\* 2):

data[i] = float(img[i]) - mean[i]

mean\_blob.destroy()

return img, rand\_file

def infer\_by\_engine(context):

# convert input data to Float32

img = img.astype(np.float32)

# create output array to receive data

output = np.empty(OUTPUT\_SIZE, dtype=np.float32)

# allocate memory on the GPU with PyCUDA and

# register it with the engine.

d\_input = cuda.mem\_alloc(1 \* img.size \* img.dtype.itemsize)

d\_output = cuda.mem\_alloc(1 \* output.size \* output.dtype.itemsize)

bindings = [int(d\_input), int(d\_output)]

# create a cuda stream to run inference

stream = cuda.Stream()

cuda.memcpy\_htod\_async(d\_input, img, stream) #transfer input data to device

context.enqueue(1, bindings, stream.handle, None) #execute model

cuda.memcpy\_dtoh\_async(output, d\_output, stream) #transfer predictions back

stream.synchronize() #syncronize threads

return output

def user\_engine(engine\_file):

engine = trt.utils.load\_engine(G\_LOGGER, engine\_file)

runtime = trt.infer.create\_infer\_runtime(G\_LOGGER)

context = engine.create\_execution\_context()

img, label = inputs()

predict = infer\_by\_engine(context)

print("Test Case: " + str(label))

print ("Prediction: " + str(np.argmax(predict)))

engine.destroy()

context.destroy()

runtime.destroy()

def main():

engine\_file = "./data/mnist/new\_mnist.engine"

create\_engine(engine\_file)

user\_engine(engine\_file)

方法3： C++接口 （主要用于自定义网络）

TensorRT基本处理过程：caffe model -> GIE的model; 运行GIE引擎(数据提前copy到GPU中), 提取结果。

1. **简单例子见TensorRT/samples/sampleMNIST**

导入到netbeans,然后debug

（建议与python作对比）

自定义网络 （通过plugin实现)

功能：将最后一层全连接层name=”ip2”替换成自定义业务逻辑

1. **samplePlugin.cpp** （详细代码请参考TensorRT/samples/samplePlugin)

客户端使用自定义层

PluginFactory pluginFactory;

caffeToGIEModel("\*.prototxt", "\*.caffemodel", "prob", 1, &pluginFactory, "\*.engine");

pluginFactory.destroyPlugin();

uint8\_t img[INPUT\_H\*INPUT\_W]; // read image and substract mean

...

engine\_data, size = readFromfile(engine\_file); // read engine file

// deserialize the engine

IRuntime\* runtime = createInferRuntime(gLogger);

ICudaEngine\* engine = runtime->deserializeCudaEngine(

(const void\*)engine\_data.get(), size, &pluginFactory);

IExecutionContext \*context = engine->createExecutionContext();

// run inference

float prob[OUTPUT\_SIZE];

doInference(\*context, img, prob, 1);

void caffeToGIEModel(...){

// caffe -> network

IBuilder\* builder = createInferBuilder(gLogger);

**INetworkDefinition\* network = builder->createNetwork();**

ICaffeParser\* parser = createCaffeParser();

parser->setPluginFactoryExt(pluginFactory);

bool fp16 = builder->platformHasFastFp16();

const IBlobNameToTensor\* blobNameToTensor = parser->parse(

deployFile.c\_str(), modelFile.c\_str(), \*network,

fp16 ? DataType::kHALF : DataType::kFLOAT);

for (auto& s : outputs) **//标记某一层为输出层，每一层均可标记为输出**

network->markOutput(\*blobNameToTensor->find(s.c\_str()));

// Build the engine

builder->setMaxBatchSize(maxBatchSize);

**// MaxWorkspaceSize 设置小于需要的，则部分算法无法执行导致崩溃或结果不确定**

**// 可以通过如下方式获取最大工作空间engine->getWorkspaceSize();**

builder->setMaxWorkspaceSize(1 << 20);

builder->setHalf2Mode(fp16);

**ICudaEngine\* engine = builder->buildCudaEngine(\*network);**

// serialize the engine, then close everything down

write2file(engine->serialize(), engine\_file)

}

void doInference(IExecutionContext& context, float\* input, float\* output, int batchSize)

{

const ICudaEngine& engine = context.getEngine();

void\* buffers[2];

int inputIndex = engine.getBindingIndex(INPUT\_BLOB\_NAME),

outputIndex = engine.getBindingIndex(OUTPUT\_BLOB\_NAME);

cudaMalloc(&buffers[inputIndex], batchSize \* INPUT\_H \* INPUT\_W \* sizeof(float));

cudaMalloc(&buffers[outputIndex], batchSize \* OUTPUT\_SIZE \* sizeof(float));

cudaStream\_t stream;

cudaStreamCreate(&stream);

// DMA the input to the GPU, execute the batch asynchronously, and DMA it back:

input\_size = batchSize \* INPUT\_H \* INPUT\_W \* sizeof(float) ;

output\_size = batchSize \* OUTPUT\_SIZE\*sizeof(float);

cudaMemcpyAsync(buffers[inputIndex], input, input\_size, cudaMemcpyHostToDevice, stream);

**context.enqueue(batchSize, buffers, stream, nullptr);**

cudaMemcpyAsync(output, buffers[outputIndex], output\_size, cudaMemcpyDeviceToHost, stream);

cudaStreamSynchronize(stream);

}

void write2file(gieModelStream, engine\_file) {

std::ofstream outfile(engine\_file.c\_str(), std::ios::out | std::ios::binary);

unsigned char\* p = (unsigned char\*)gieModelStream->data();

outfile.write((char\*)p, gieModelStream->size());

}

void readFromfile(engine\_file) {

std::ifstream in\_file(engine\_file.c\_str(), std::ios::in | std::ios::binary);

std::size\_t size = fun(in\_file.tellg(), in\_file.seekg())

std::unique\_ptr<unsigned char[]> engine\_data(new unsigned char[size]);

in\_file.read((char\*)engine\_data.get(), size);

}

自定义层

class **FCPlugin**: public IPluginExt

{

public:

**FCPlugin(const Weights \*weights, int nbWeights, int nbOutputChannels)**

: mNbOutputChannels(nbOutputChannels){

// 获取权重和偏置blob

mKernelWeights = weights[0];

mBiasWeights = weights[1];

...allocate CPU memeory of mKernelWeights and mBiasWeights

...mKernelWeights.vlaues, mBiasWeights.values <- weights[0].values, weights[1].values

mNbInputChannels = int(weights[0].count / nbOutputChannels);

}

**FCPlugin(const void\* data, size\_t length)** { // create the plugin at runtime from a byte stream }

**~FCPlugin()** { //释放CPU资源 }

#----------------------required by creating the network---------------------#

int **getNbOutputs**() const override { return 1; }

Dims **getOutputDimensions**(int index, const Dims\* inputs, int nbInputDims) override

{ return DimsCHW(mNbOutputChannels, 1, 1); }

#----------------------required by builder---------------------------------#

void **configureWithFormat**(..., DataType type, ...) override { mDataType = type; }

bool **supportsFormat**(DataType type, PluginFormat format) const override

{ return (type == DataType::kFLOAT || type == DataType::kHALF) &&

format == PluginFormat::kNCHW; }

virtual size\_t **getWorkspaceSize**(int maxBatchSize) const override { return 0; }

#----------------------required by runtime----------------------------------#

int **initialize**() override {

// copy data from CPU to GPU

mDeviceKernel, mDeviceBias <- mKernelWeights, mBiasWeights

}

virtual int **enqueue**(int batchSize, const void\*const \* inputs, void\*\* outputs, void\* workspace, **cudaStream\_t stream**) override {

// 实现自定义层业务逻辑

cublasSetStream(mCublas, stream);

cudnnSetStream(mCudnn, stream);

if (mDataType == DataType::kFLOAT) { cublasSgemm(...); }

else { cublasHgemm(...); }

if (mBiasWeights.count) { cudnnAddTensor(...); }

}

virtual void **terminate**() override { //释放GPU资源 }

#----------------------required by serialization----------------------------#

virtual size\_t **getSerializationSize**() override {

return sizeof(mNbInputChannels) + sizeof(mNbOutputChannels) + sizeof(mBiasWeights.count) + sizeof(mDataType) + (mKernelWeights.count + mBiasWeights.count) \* type2size(mDataType); }

virtual void **serialize**(void\* buffer) override {

char\* d = static\_cast<char\*>(buffer), \*a = d;

write(d, mNbInputChannels);

write(d, mNbOutputChannels);

write(d, mBiasWeights.count);

write(d, mDataType);

convertAndCopyToBuffer(d, mKernelWeights);

convertAndCopyToBuffer(d, mBiasWeights);

}

private:

int mNbOutputChannels, mNbInputChannels;

Weights mKernelWeights, mBiasWeights;

void\* mDeviceKernel{nullptr};

void\* mDeviceBias{nullptr};

};

// integration for serialization

class **PluginFactory** : public nvinfer1::IPluginFactory, public nvcaffeparser1::IPluginFactoryExt

{

public:

// caffe parser plugin implementation，注意自定义层替代最后一层name=”ip2”

bool **isPlugin**(const char\* name) override { return **!strcmp(name, "ip2");** }

virtual IPlugin\* **createPlugin**(char\* layerName, const Weights\* weights, int nbWeights) override

{

assert(isPlugin(layerName) && nbWeights == 2);

const int NB\_OUTPUT\_CHANNELS = 10;

mPlugin = new FCPlugin(weights, nbWeights, NB\_OUTPUT\_CHANNELS);

return mPlugin.get();

}

// deserialization plugin implementation

IPlugin\* **createPlugin**(char\* layerName, const void\* serialData, size\_t serialLength) override

{

mPlugin = new FCPlugin(serialData, serialLength);

return mPlugin.get();

}

// the application has to destroy the plugin when it knows it's safe to do so

void **destroyPlugin**() { mPlugin.release(); }

std::unique\_ptr<FCPlugin> **mPlugin**{ nullptr };

};

方法4： TensorRT + Caffe

假定复杂网络拆解成三个网络： (如何拆解见caffe--剖析模型)

net1.prototxt, net1.caffemodel; 并优化成net1.engine (如何转换见tensorrt.lite)

net2.prototxt, net2.caffemodel; 假定很复杂无法转化为tensorrt, 直接用caffe

net3.prototxt, net3.caffemodel; 并优化成net3.engine

注意，用tensorrt.lite.Engine，不能同时优化几个网络，一个进程只能优化一个网络，否则第二个网络无法通过

级联使用engine + caffe + engine

engine\_visual = trt.lite.Engine(PLAN='step1.engine')

input\_name = 'Pooling2'

net2 = caffe.Net('net2.prototxt', 'net2.caffemodel', caffe.TEST)

engine\_ip = trt.lite.Engine(PLAN='step3.engine')

for i in range(10):

img, rand\_file = generate\_data()

img = np.reshape(img, (1, 1, 28, 28))

Pooling2 = engine\_visual.infer(img)

Pooling2 = Pooling2[0]

# net2.blobs[input\_name].reshape(\*Pooling2.shape)

net2.blobs[input\_name].data[...] = Pooling2

net2.forward()

InnerProduct1 = net2.blobs['InnerProduct1'].data

InnerProduct1 = np.reshape(InnerProduct1, (1, 1, 1, InnerProduct1.shape[-1]))

prob = engine\_ip.infer(InnerProduct1)

print("Test Case: " + str(rand\_file))

print("Prediction: " + str(np.argmax(prob)))

举例：

RefineDet

deploy.prototxt = backbone.prototxt + refinedet.prototxt

deploy.caffemodel = backbone.caffemodel + refinedet.caffemodel

**backbone.prototxt**

name: "backbone"

input: "data"

input\_shape { dim: 1 dim: 3 dim: 320 dim: 320 }

layer {

name: "conv1\_1"

type: "Convolution"

...

}

**refinedet.prototxt**

name: "refinedet"

input: "data"

input\_shape { dim: 1 dim: 3 dim: 320 dim: 320 }

input: "conv4\_3"

input\_shape { dim: 1 dim: 512 dim: 40 dim: 40 }

input: "conv5\_3"

input\_shape { dim: 1 dim: 512 dim: 20 dim: 20 }

input: "fc7"

input\_shape { dim: 1 dim: 1024 dim: 10 dim: 10 }

input: "conv6\_2"

input\_shape { dim: 1 dim: 512 dim: 5 dim: 5 }

layer {

name: "TL6\_1"

type: "Convolution"

bottom: "conv6\_2"

top: "TL6\_1"

...

}

Inference (图像大小： 每秒帧数)

# load model

engine\_visual = trt.lite.Engine(PLAN=engine\_backbone)

net2 = caffe.Net(prototxt\_refinedet, weights\_refinedet, caffe.TEST)

net2.blobs['data'].reshape(\*input\_size)

net2.blobs['conv4\_3'].reshape(1, 512, 40, 40)

net2.blobs['conv5\_3'].reshape(1, 512, 20, 20)

net2.blobs['fc7'].reshape(1, 1024, 10, 10)

net2.blobs['conv6\_2'].reshape(1, 512, 5, 5)

descriptor = []

for (i, im\_name) in enumerate(im\_names):

image\_file = 'examples/images/' + im\_name

image = caffe.io.load\_image(image\_file)

transformed\_image = transformer.preprocess('data', image)

ts = time()

output = engine\_visual.infer(transformed\_image)

net2.blobs['data'].data[...] = transformed\_image

net2.blobs['conv4\_3'].data[...] = output[0]

net2.blobs['conv5\_3'].data[...] = output[1]

net2.blobs['fc7'].data[...] = output[2]

net2.blobs['conv6\_2'].data[...] = output[3]

detections = net2.forward()['detection\_out']

te = time()

descriptor.append({

image.shape: 1.0/(te-ts)

})

1. Faster RCNN

TensorRT仅支持图像Portable PixMap (PPM)

Faster R-CNN 作了些改变

* The RPN and the ROI pooling layer is fused and replaced by a custom layer named RPROIFused .
* The reshape layer is replaced with two custom layers: ReshapeCTo2 and ReshapeCTo18 and are defined in the sample.

1. **Tensorflow -> TensorRT**
2. Unsupported Framework -> TensorRT

例子: PyTorch -> TensorRT

模型压缩

更精细模型的设计

如SqueezeNet、MobileNet等，使用更加细致、高效的模型设计，能够很大程度的减少模型尺寸，并且也具有不错的性能

模型剪枝 （有损）

将不重要的connection或者filter进行裁剪来减少模型的冗余

比如去除低阈值权重，然后稀疏编码

模型量化 （有损）

半精度浮点

为什么8bits对深度神经网络够用?

低功耗以及移动设备

八位数值的存取相对浮点数而言内存带宽降到 25%

后向传播和梯度需要浮点精度。推断时可以使用低精度

TensorFlow 自带对八位运算的生产级支持。它也能把浮点模型转换为等价的使用量化计算进行推断的图 (Graph；TensorFlow 里用来表达计算过程和内部状态的结构)

bazel -bin/tensorflow/contrib/quantization/tools/quantize\_graph \

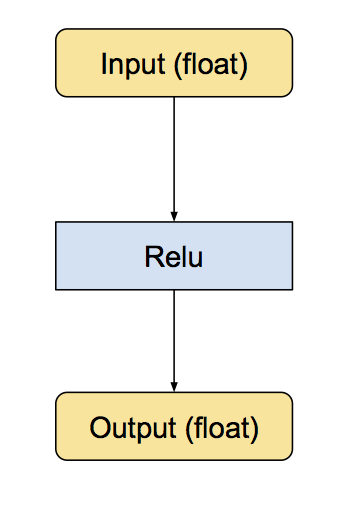
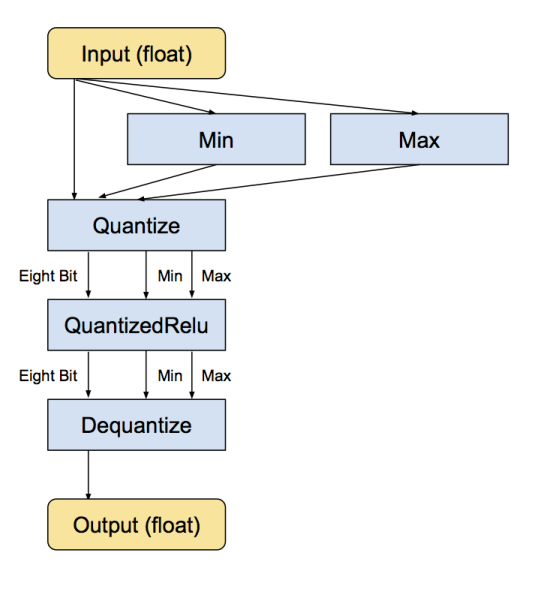
--input=/tmp/classify\_image\_graph\_def.pb \

--output\_node\_names="softmax" --output=/tmp/quantized\_graph.pb \

--mode=eightbit

量化是如何进行的？

通过把常见操作转换为等价的八位版本达到的。涉及的操作包括卷积，矩阵乘法，活化函数，池化操作，以及拼接

良好训练的神经网络必须能应对训练数据中的无关信息，这成就了神经网络对输入噪声和计算误差的强壮性

深度网络的一个魔力特性就是能够很好地应对较大的输入噪声。比如为了识别照片中的物体，网络必须忽略所有的 CCD 噪声、光照变化，以及其它与之前训练样本之间的非本质差异，而只关注重要的相似之处。这种能力意味着神经网络似乎把低精度计算视为另一种噪声来源，而在数值格式精度较低的情况下仍能给出准确结果。

通过实验也知道神经网络对噪声不敏感，所以量化噪声对结果总体来说影响不会很大

如何决定范围？如果你对八位输入进行任何算术计算，你会自然地累积得到超过八位精度能表达的数；需要把超过八位的输出缩减传递给下一个操作。对矩阵乘法而言，一个办法是根据可能的极端输入计算其输出的范围

舍入如何进行？神经网络对噪声不敏感，但如果对舍入操作不小心，会导致偏差往某个方向累积，最终影响精度

# Deep learning framework

|  |  |  |  |
| --- | --- | --- | --- |
|  | By | interface | description |
| Keras |  | Python | neural networks library  running on top of either TensorFlow or Theano, fast **experimentation** |
| Caffe2 | the Berkeley Vision and Learning Center (BVLC) | C, C++, Python, MATLAB, cmd | deep learning framework  工业级  支持多主机多GPU |
| TensorFlow | Google’s Machine Intelligence research organization | C++, Python | software library  numerical computation using data flow graphs |
| CNTK | The Microsoft Cognitive Toolkit | Python, C++, C# and cmd | unified deep-learning toolkit  train and combine popular model types across multiple GPUs and servers |
| theano |  | Python | math expression compiler  defines, optimizes, and evaluates mathematical expressions |
| Torch |  | C, C++, Lua | scientific computing framework  offers wide support for machine learning algorithms. **experimentation** |

Theano: a compiler for mathematical expressions in Python

TensorFlow: a python library for fast numerical computing

Keras: library addresses these concerns by providing a wrapper for both Theano and Tensorflow

scikit-learn library：general purpose machine learning framework in Python built on top

of SciPy

若内存足够大，一次将所有训练数据导入内存

model.fit(X\_train, Y\_train, batch\_size=batch\_size, nb\_epoch=12)

若CPU内存不大，无法一次性将所有训练数据导入，怎么办？

批次导入训练数据

存在多种方案：

方案一：先导入subset1, 训练12次，然后导入subset2,训练12次，...直至遍历所有训练集

方案二：先导入subset1, 训练1次，然后导入subset2, 训练1次，...遍历所有训练集，然后重复以上12次

以上是对mnist数据集的比较结果：建议采用方案二

0.989 total, 12 epoch 397s

0.986 11\*subset, 12 epoch / subset 396s

0.989 12 \* (11\*subset + 1 epoch / subset) 393s

keras 代码

from psutil import virtual\_memory

mem = virtual\_memory()

free\_memory = ratio \* float(mem.total) / 1024 \*\* 3

case\_num\_memory = f(free\_memory, case\_size)

subset\_num = ceil(all\_case\_num/subset\_num)

for j in range(epoches):

for i in range(subset\_num):

history = model.fit(train.subset(i), batch\_size=batch\_size, 1)

tensorflow代码

batch\_num = ceil(all\_case\_num/batch\_size)

for j in range(epoches):

for i in range(batch\_num):

batch = train.next\_batch(batch\_size)

sess.run([...], feed\_dict={X:batch[0], Y:batch[1]}

Caffe

<http://caffe.berkeleyvision.org/installation.html>

<https://www.cnblogs.com/go-better/p/7161006.html>

Install

cuda8, cudnn (keras有详细的说明）

$ pip install opencv-contrib-python 安装opencv

$ pkg-config --modversion opencv 获取opencv版本

sudo apt-get install libprotobuf-dev libleveldb-dev libsnappy-dev libopencv-dev libhdf5-serial-dev protobuf-compiler

sudo apt-get install --no-install-recommends libboost-all-dev

sudo apt-get install libopenblas-dev

$git clone https://github.com/BVLC/caffe.git

$cd caffe

caffe$ sudo cp Makefile.config.example Makefile.config

caffe$ sudo gedit Makefile.config

INCLUDE\_DIRS := $(PYTHON\_INCLUDE) /usr/local/include /usr/include/hdf5/serial

LIBRARY\_DIRS := $(PYTHON\_LIB) /usr/local/lib /usr/lib /usr/lib/x86\_64-linux-gnu /usr/lib/x86\_64-linux-gnu/hdf5/serial

若opencv3版本

OPENCV\_VERSION := 3

$ make all -j8

$ make test –j8

$ make runtest –j8

编译pycaffe

$ cd ~/caffe

caffe$ sudo apt-get install gfortran

$ cd ./python

caffe/python$ for req in $(cat requirements.txt); do pip install $req; done

caffe$ make pycaffe -j8

若出现numpy找不到，需要设置Makefile.config

PYTHON\_INCLUDE := /usr/include/python2.7 \

/usr/lib/python2.7/dist-packages/numpy/core/include \

/usr/local/lib/python2.7/dist-packages/numpy/core/include

配置环境变量，以便python调用

sudo gedit ~/.bashrc

export PYTHONPATH=/home/caffe/python:$PYTHONPATH

source ~/.bashrc

tutorial

caffe基本概念：

blob：存储数据和权值

layer：输入数据blob 形式，输出数据blob形式，层定义了计算

net：由多个layers组成，构成整体的网络

solver：定义了训练规则

**Blob** storage and communication

a blob is an N-dimensional array stored in a C-contiguous fashion.

* batches of images

number N x channel K x height H x width W

Blob memory is row-major in layout, so the last / right most dimension changes fastest

* model parameters

For a convolution layer with 96 filters of 11 x 11 spatial dimension and 3 inputs the blob is 96 x 3 x 11 x 11.

For an inner product / fully-connected layer with 1000 output channels and 1024 input channels the parameter blob is 1000 x 1024

* derivatives for optimization

Blob stores two chunks of memories, data and diff. The former is the normal data that we pass along, and the latter is the gradient computed by the network.

要运行caffe，需要先创建一个model=layers + layer+ ...，每一屋又由许多参数组成。熟练使用caffe最重要的就是学会配置文件.prototxt的编写。

**Layer** computation and connections

A layer takes input through bottom connections and makes output through top connections.

数据层是每个模型的最底层，是模型的入口，不仅提供数据的输入，也提供数据从Blobs转换成别的格式进行保存输出。通常**数据的预处理（如减去均值, 放大缩小, 裁剪和镜像等**），也在这一层设置参数实现。数据来源可以来自高效的数据库（如LevelDB和LMDB），也可以直接来自于内存。如果不是很注重效率的话，数据也可来自磁盘的hdf5文件和图片格式文件

1. **Data Layers**

ImageData - read raw images. --图像,方便data aug

Data - read data from LEVELDB or LMDB. --分类问题

Input - typically used for networks that are being deployed.

HDF5Data - allows data of arbitrary dimensions. --多标签回归

HDF5 Output - write data as HDF5.

Window Data - read window data file.

Memory Data - read data directly from memory.

Dummy Data - for static data and debugging.

AnnotatedData --read data from LMDB --用于目标检测

Data layers load input and save output by converting to and from Blob to other formats

1. 数据库LMDB or LEVELDB使用

创建LMDB数据库见章节-- **训练和测试自己的图片**

layer {

name: "mnist" #表示该层名称

**type: "Data"** # loads leveldb or lmdb storage DBs

# top或bottom: 每一层用bottom来输入数据，用top来输出数据

top: "data" # the 1st top is the data itself: the name is only convention

top: "label" # the 2nd top is the ground truth

# a rule, 指定该层属于训练或测试等

include: { phase: TRAIN }

**data\_param** { # the Data layer configuration

**source**: "examples/mnist/mnist\_train\_lmdb" # path to the DB

# type of DB: LEVELDB or LMDB (LMDB supports concurrent reads)

**backend**: LMDB

**batch\_size**: 64 # batch processing improves efficiency.

}

# 数据的预处理

transform\_param { # common data transformations

scale: 0.00390625 #maps the [0, 255] MNIST data to [0, 1]

mean\_file: "cifar10/mean.binaryproto" # 用一个配置文件来进行均值操作

mirror: 1 # 1/ture表示开启镜像，0/false表示关闭

crop\_size: 227 # 剪裁 227\*227图块，训练阶段随机剪裁，测试阶段从中间裁剪

}

}

数据来自于图片

layer {

**type: "ImageData"**

**image\_data\_param** {

# 一个文本文件的名字，每一行给定一个图片文件的名称和标签（label)

**source**: "examples/\_temp/file\_list.txt"

**batch\_size**: 50

# 如果设置，则将图片进行resize

**new\_height**: 256

**new\_width**: 256

}

...

}

1. HDF5数据

train.txt, val.txt, test.txt

\*.jpg 0.31 0.85

产生对应的HDF5文件

import h5py

import numpy as np

with open('train.txt', 'r') as f:

lines = f.readlines()

np.random.shuffle(lines)

with h5py.File("data.h5", 'w') as h:

for i, line in enumerate(lines):

imagefile, reg0, reg1 = line[:-1].split()

img = transform(imagefile) //减均值，scale, ...

imgs[i], regs[i] = img, [reg0, reg1]

h.create\_dataset('data', data=imgs)

h.create\_dataset('reg', data=regs)

with open('data\_h5.txt', 'w') as f:

f.write('data.h5')

layer {

name: "data"

**type: "HDF5Data"**

top: "data"

top: "label"

**hdf5\_data\_param** {

**source: "data\_h5.txt"**

**batch\_size: 32**

}

}

图像深度特征提取路径,然后flatten成一维向量fc

layer {

type: "Convolution"

top: "conv1\_1"

bottom: "data"

...

}

...

layer {

type: "InnerProduct"

top: "fc\_out"

bottom:"fc"

inner\_product\_param {

num\_output: 2 预测回归

...

}

}

layer {

type: "Sigmoid"

top: "pred"

bottom: "fc\_out"

}

layer {

type: "EuclideanLoss" //回归损失函数

top: "loss"

bottom: "pred" //预测输出

bottom: "label" //groundtruth

}

1. **Vision Layers**

Vision layers usually take images as input and produce other images as output, other layers (with few exceptions) ignore the spatial structure of the input, effectively treating it as “one big vector” with dimension chw.

* Convolution Layer - convolves the input image with a set of learnable filters, each producing one feature map in the output image.

layer {

name: "conv1"

**type: "Convolution"**

# lr = lr\_mult\*base\_lr (学习率base\_lr在solver.prototxt设置）

param { lr\_mult: 1 } **#weight学习率系数调整**

param { lr\_mult: 2 } **#bias学习率系数调整**

convolution\_param {

**num\_output**: 20 # 卷积核（filter)的个数

**kernel\_size**: 5 # 卷积核的大小

stride: 1 # 卷积核的步长，默认为1

pad: 0 # 扩充边缘，默认为0，不扩充。 扩充的时候是左右、上下对称的，比如卷积核的大小为5\*5，那么pad设置为2，则四个边缘都扩充2个像素，即宽度和高度都扩充了4个像素,这样卷积运算之后的特征图就不会变小

# 权值初始化， 默认为“constant",值全为0， 还可以选择"xavier" / "gaussian"

weight\_filler { type: "xavier" }

# 偏置项的初始化。一般设置为"constant",值全为0

bias\_filler { type: "constant" }

}

**bottom**: "data"

**top**: "conv1"

}

注：

xavier algorithm: initialize based on the number of input/output neurons

输入：n\*c0\*w0\*h0

输出：n\*num\_output\*w1\*h1

w1=(w0+2\*pad-kernel\_size)/stride+1;

h1=(h0+2\*pad-kernel\_size)/stride+1;

一般设置stride=1, pad=(kernel\_size-1)/2, 那么宽度高度不变

* Pooling Layer - max, average, or stochastic pooling.

layer {

name: "pool1"

**type: "Pooling"**

pooling\_param {

**kernel\_size: 2** # 池化的核大小

pad: 0 # 边缘扩充, 默认为0

stride: 2# 池化的步长，默认为1。一般设为2，即不重叠

pool: MAX#池化方法，默认为MAX。MAX/AVE/STOCHASTIC

}

**bottom**: "conv1"

**top**: "pool1"

}

注：

输入：n\*c\*w0\*h0

输出：n\*c\*w1\*h1

和卷积层的区别就是其中的c保持不变

w1=(w0+2\*pad-kernel\_size)/stride+1;

h1=(h0+2\*pad-kernel\_size)/stride+1;

一般设置stride=2, 特征图池化后缩小一倍

* Spatial Pyramid Pooling (SPP)
* Crop - perform cropping transformation.
* Deconvolution Layer - transposed convolution.

1. **Activation / Neuron Layers**

* ReLU / Rectified-Linear and Leaky-ReLU - ReLU and Leaky-ReLU rectification.

layer {

name: "relu1"

**type: "ReLU"**

**bottom**: "ip1"

**top**: "ip1"

}

Since ReLU is an element-wise operation, we can do **in-place operations to save some memory**. This is achieved by simply giving the same name to the bottom and top blobs

* PReLU - parametric ReLU.
* ELU - exponential linear rectification.
* Sigmoid, TanH, Absolute Value
* Power - f(x) = (shift + scale \* x) ^ power.
* Exp - f(x) = base ^ (shift + scale \* x).
* Log - f(x) = log(x).
* BNLL - f(x) = log(1 + exp(x)).
* Threshold - performs step function at user defined threshold.
* Bias - adds a bias to a blob that can either be learned or fixed.
* Scale - scales a blob by an amount that can either be learned or fixed.

1. **Common Layers**

* Inner Product : 全连接层

layer {

name: "ip1"

**type: "InnerProduct"**

param { lr\_mult: 1 }

param { lr\_mult: 2 }

inner\_product\_param {

**num\_output:** 500

weight\_filler { type: "xavier" }

bias\_filler { type: "constant" }

}

**bottom**: "pool2"

**top**: "ip1"

}

* Dropout

layer {

name: "drop7"

type: "Dropout"

bottom: "fc7-conv"

top: "fc7-conv"

dropout\_param {

**dropout\_ratio:** 0.5

}

}

* Softmax

layers {

name: "prob"

**type: “Softmax"**

bottom: "cls3\_fc"

top: "prob"

}

* Reshape

layer {

name: "reshape"

**type: "Reshape"**

bottom: "input"

top: "output"

# 用于指定blob数据的各维的值（blob是一个四维的数据：n\*c\*w\*h）。

reshape\_param {

shape {

**dim**: 0 # 表示维度不变，即输入和输出是相同的维度。

**dim**: 0

**dim**: 14 # 将原来的维度变成14

**dim**: -1 # infer it from the other dimensions

}

}

}

注： 假设原数据为：64\*3\*28\*28， 输出数据为：64\*3\*14\*56

* Flatten: 将(n, c, h, w) -> (n, c\*h\*w, 1, 1)
* Batch Reindex,
* Split: 一输入 -> 多输出
* Concat:多输入 -> 一输出
* Slicing
* Eltwise - element-wise operations such as product or sum between two blobs.
* Filter / Mask - mask or select output using last blob.
* Parameter - enable parameters to be shared between layers.
* Reduction - reduce input blob to scalar blob using operations such as sum or mean.
* Silence - prevent top-level blobs from being printed during training.
* ArgMax
* Python - allows custom Python layers.
* Embed - for learning embeddings of one-hot encoded vector (takes index as input).
* Im2col

1. **Loss Layers**

* Softmax with Loss - computes the multinomial logistic loss of the softmax of its inputs. It’s conceptually identical to a softmax layer followed by a multinomial logistic loss layer, but provides a more numerically stable gradient.

layer {

name: "loss"

**type: "SoftmaxWithLoss"**

**bottom**: "ip2"

**bottom**: "label"

**top**: “loss”

}

to compute the loss function value, report it when backpropagation starts, and initiates the gradient with respect to ip2

* Multinomial Logistic Loss
* Infogain Loss - a generalization of MultinomialLogisticLossLayer.
* Sum-of-Squares / Euclidean - computes the sum of squares of differences of its two inputs, 12N∑Ni=1‖x1i−x2i‖22.
* Hinge / Margin - The hinge loss layer computes a one-vs-all hinge (L1) or squared hinge loss (L2).
* Sigmoid Cross-Entropy Loss - computes the cross-entropy (logistic) loss, often used for predicting targets interpreted as probabilities.
* Contrastive Loss

1. **Accuracy / Top-k layer**

scores the output as an accuracy with respect to target – it is not actually a loss and has no backward step.

layer {

name: "accuracy"

**type: "Accuracy"**

**bottom**: "ip2"

**bottom**: "label"

**top**: "accuracy"

**include { phase: TEST }** 仅在测试时，才计算accuracy

}

1. **Normalization Layers**

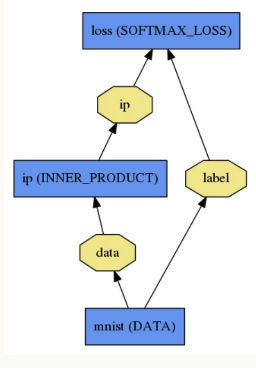
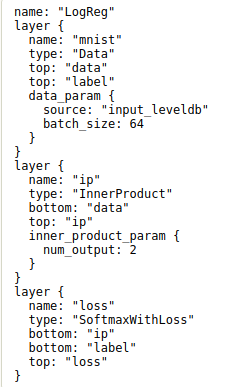
* Local Response Normalization (LRN) - performs a kind of “lateral inhibition” by normalizing over local input regions.
* Mean Variance Normalization (MVN) - performs contrast normalization / instance normalization.
* Batch Normalization - performs normalization over mini-batches.

1. **Recurrent Layers**

Recurrent, RNN, Long-Short Term Memory (LSTM)

**Net =** layers + layers + ...

The net is a set of layers connected in a computation graph – a directed acyclic graph (DAG) , A typical net begins with a data layer that loads from disk and ends with a loss layer that computes the objective for a task such as classification or reconstruction.

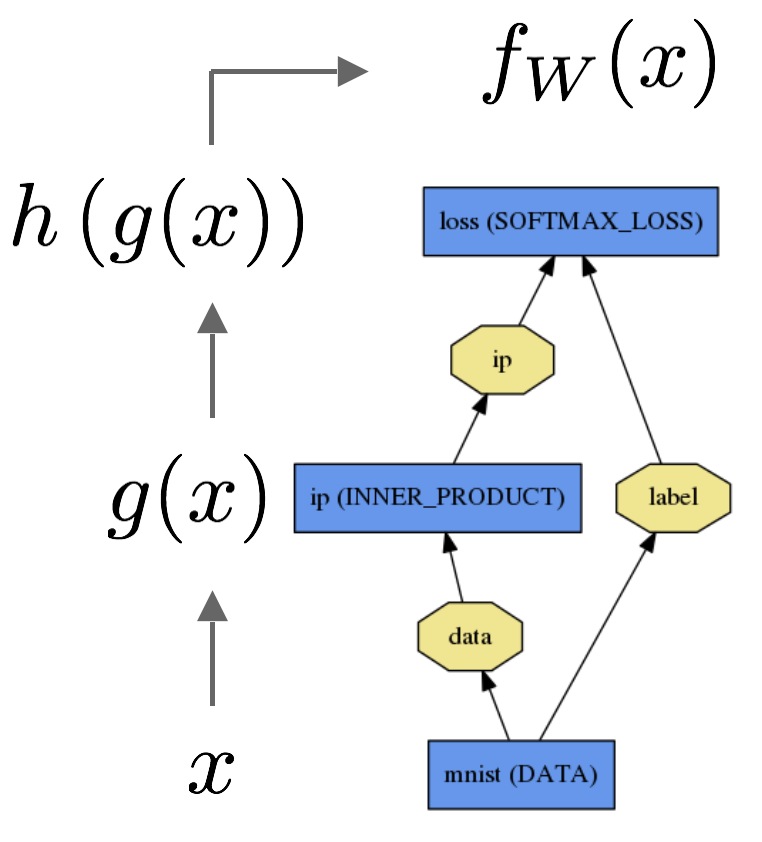
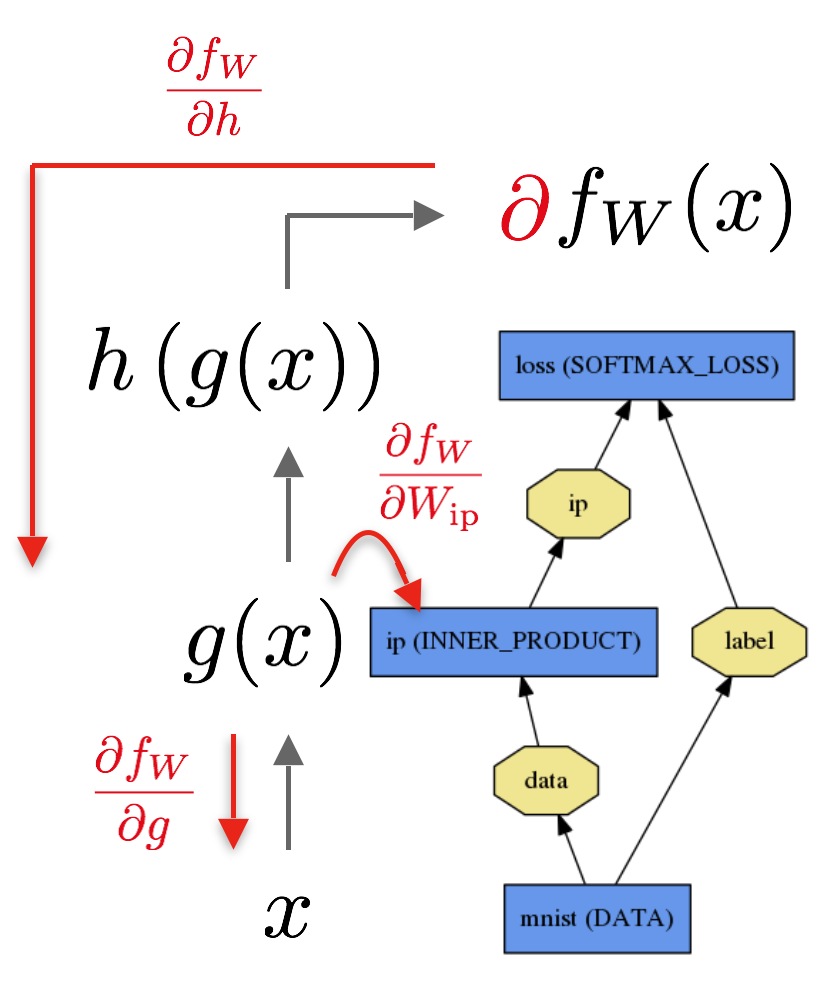
 

**Model**

models definition: **.prototxt** (in plaintext)

models learned: **.caffemodel** (serialized as binary).

**Forward / Backward**

The data x is passed through an inner product layer for g(x) then through a softmax for h(g(x)) and softmax loss to give fW(x).

The backward pass begins with the loss and computes the gradient with respect to the output ∂fW/∂h. The gradient with respect to the rest of the model is computed layer-by-layer through the chain rule. Layers with parameters, like the INNER\_PRODUCT layer, compute the gradient with respect to their parameters ∂fW/∂Wip during the backward step.

**Loss (error, cost, or objective)**

learning is driven by a loss function. By convention, Caffe layer types with the suffix Loss contribute to the loss function, but other layers are assumed to be purely used for intermediate computations. However, any layer can be used as a loss by adding a field loss\_weight: <float> to a layer definition for each top blob produced by the layer

# 添加正则化损失函数

loss := 0

for layer in layers:

for top, loss\_weight in layer.tops, layer.loss\_weights:

loss += loss\_weight \* sum(top)

在Deep Learning中，往往loss function是非凸的，没有解析解，我们需要通过优化方法来求解。solver的主要作用就是交替调用前向（forward)算法和后向（backward)算法来更新参数，从而最小化loss，实际上就是一种迭代的优化算法。

**Solver**

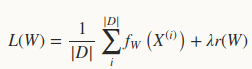
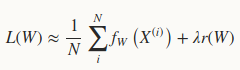
optimizes a model by

1. first calling forward to yield the output and loss,
2. then calling backward to generate the gradient of the model,
3. and then incorporating the gradient into a weight update that attempts to minimize the loss

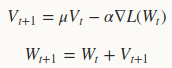
each iteration

1. calls network forward to compute the output and loss
2. calls network backward to compute the gradients
3. incorporates the gradients into parameter updates according to the solver method
4. updates the solver state according to learning rate, history, and method

损失函数: dataset D -> mini-batch N

 -> 

SGD:



learning rate α is the weight of the negative gradient.

The momentum μ is the weight of the previous update

./examples/imagenet/**alexnet\_solver.prototxt**

设置深度网络模型。每一个模型就是一个net，需要在一个专门的配置文件中对net进行配置，每个net由许多的layer所组成

注意的是：文件的路径要从caffe的根目录开始，其它的所有配置都是这样。

也可用train\_net和test\_net来对训练模型和测试模型分别设定。例如：

train\_net: "examples/hdf5\_classification/logreg\_auto\_train.prototxt"

test\_net: "examples/hdf5\_classification/logreg\_auto\_test.prototxt"

**net:** "examples/mnist/lenet\_train\_test.prototxt"

在caffe中的一次迭代iter指的是一个batch,而不是一张图片

//在测试的时候，需要迭代的次数; test\_iter = 测试样本总数/test\_batch\_size

**test\_iter**: 100

//经过多少次batch\_size的训练，然后进行一次测试

一般遍历一次全部训练数据(即一个epoch), 然后测试一次，则test\_interval = 训练样本总数/ train\_batch\_size

**test\_interval**: 500

**max\_iter**: 350000 # 350K次batch\_size训练， 建议epoch \* 训练集大小/batch\_size

iter\_size： X

每次循环都会以batch\_size大小计算梯度和loss，最后再取iter\_size次的平均。可以看成iter\_size\*batch\_size次更新一次参数;即处理batchsize\*itersize张图片后，才进行梯度下降

如何更新学习率？

lr\_policy可以设置为下面这些值，相应的学习率的计算为：

fixed #保持base\_lr不变.

step　 #step + stepsize + gamma

#返回 base\_lr \* gamma ^ (floor(iter / stepsize)),iter当前迭代次数

exp 　　 #返回base\_lr \* gamma ^ iter， iter为当前迭代次数

inv　　 #inv + power

#返回base\_lr \* (1 + gamma \* iter) ^ (- power)

multistep #multistep + stepvalue

#step是均匀等间隔变化，而multistep则是根据stepvalue值变化

poly 　 #学习率进行多项式误差, 返回 base\_lr (1 - iter/max\_iter) ^ (power)

sigmoid #学习率进行sigmod衰减

#返回 base\_lr ( 1/(1 + exp(-gamma \* (iter - stepsize))))

**base\_lr**: 0.01 # begin training at a learning rate of 0.01 = 1e-2

**lr\_policy**: "step" # learning rate policy: drop the learning rate

**gamma**: 0.1 # drop the learning rate by a factor of 10

**stepsize**: 100000 # drop the learning rate every 100K iterations

**momentum**: 0.9

type: SGD # 默认值就是SGD， AdaDelta/AdaGrad/Adam/Nesterov/RMSProp

**weight\_decay**: 0.0005 # 权重衰减项

**display**: 100 #每训练100次，在屏幕上显示一次。如果设置为0，则不显示

快照。将训练出来的model和solver状态进行保存，snapshot用于设置训练多少次后进行保存，默认为0，不保存。snapshot\_prefix设置保存路径

**snapshot**: 5000

**snapshot\_prefix**: "examples/mnist/lenet"

#保存的类型HDF5/BINARYPROTO ，默认为BINARYPROTO

snapshot\_format: "BINARYPROTO"

snapshot\_diff: false 是否保存梯度值，默认为false

solver\_mode: CPU CPU/GPU, 默认为GPU

If learning diverges (e.g., you start to see very large or NaN or inf loss values or outputs), try dropping the base\_lr (e.g., base\_lr: 0.001) and re-training, repeating this until you find a base\_lr value that works.

感觉训练时噪音太大时:

调大batch size到喂饱硬件 (显卡容易爆）

带learning rate下降的sgd，开始时依赖batch带来的噪音快速下降，接下来使用较低的learning rate消除这些噪音寻求稳定收敛

大batch size相当于小lr，反之亦然。可以根据收敛速度进行一定选择。

另外如果用了batchnorm，batch size别太小（大于64？）

如果不用batchnorm，可以考虑用小batch size甚至1 来得到最优的结果

可视化.prototxt

可视化编辑器

网址：<http://ethereon.github.io/netscope/#/editor>

内容copy到左边编辑框，然后shift+enter

Mouse指者对应层，可以显示对应层的信息

.prototxt -> 图片

$sudo apt-get install graphviz

$sudo pip install pydot

# **sudo python python/draw\_net.py** \*.prototxt \*.png --rankdir=BT

第一个参数：网络模型的prototxt文件

第二个参数：保存的图片路径及名字

第二个参数：--rankdir=x , x 有四种选项，分别是LR, RL, TB, BT 。用来表示网络的方向，分别是从左到右，从右到左，从上到小，从下到上。默认为ＬＲ。

**Interfaces--Command Line (train, test, time)**

$ sudo gedit ~/.bashrc

export PATH="/home/qzlin/Documents/dl/caffe/build/tools:$PATH"

$ **caffe train**

#模型的配置文件

**-solver** examples/finetuning/solver**.prototxt**

#预先训练好的权重来fine-tuning模型，不能和-snapshot同时使用

**-weights** models/bvlc\_reference\_caffenet/ bvlc**.caffemodel**

#从snapshot中恢复训练。可以在solver配置文件设置快照，保存solverstate

**-snapshot** model\_iter\_1000**.solverstate**

**-gpu all** # 指定用哪一块gpu运行，或-gpu all使用所有gpu运行

| tee \*.log #stdout 即屏幕输出信息输出到指定文件

建议写个脚本文件启动

jobfile.sh

cd /home/qzlin/Documents/dl/RefineDet/

./build/tools/caffe train \

--solver=".../solver.prototxt" \

--snapshot=".../\*.solverstate" \

--gpu 0,1 2>&1 | tee jobfile.log

train.py

import os, stat, subprocess

def run\_job(job\_file):

os.chmod(job\_file, stat.S\_IRWXU)

subprocess.call(job\_file, shell=True)

若solver.prototxt不仅仅配置train.prototxt, 还配置了test.prototxt,那么训练和测试交替进行

solver.prototxt

train\_net: "models/vgg16/VOC0712/refinedet\_vgg16\_320x320/train.prototxt"

test\_net: "models/vgg16/VOC0712/refinedet\_vgg16\_320x320/test.prototxt"

test\_iter: 4952 //在测试的时候，需要迭代的次数，一般test\_iter = test\_size/test\_batch\_size

**test\_interval:**  //经过多少次train\_batch\_size的训练，然后进行一次测试

...

**eval\_type: "detection"**

ap\_version: "11point"

caffe train可以用来当测试使用

修改solver.prototxt: max\_iter=0

$caffe train -solver solver.prototxt -weights final.caffemodel

直接进入solver.sovle() -> TestAll()

**绘制loss and accuracy曲线**

通过可视化log文件

$python caffe\tools\extra\plot\_training\_log.py 0 test\_acc\_vs\_iters.png \*.log

$python caffe\tools\extra\plot\_training\_log.py 2 test\_loss\_vs\_iters.png \*.log

参数：图的类型，生成图片的路径和log的路径

0: 测试准确率 vs. 迭代次数

1: 测试准确率 vs. 训练时间(秒)

2: 测试loss vs. 迭代次数

3: 测试loss vs. 训练时间

4: 学习率 vs. 迭代次数

5: 学习率 vs. 训练时间

6: 训练loss vs. 迭代次数

7：训练loss vs. 训练时间

$ **caffe test**

**-model** examples/mnist/lenet\_train\_test.prototxt /

**-weights** examples/mnist/lenet\_iter\_10000.caffemodel /

-gpu 0,1 -iterations 100

Iterations值 = 测试大小/batch\_size, 若不设置，默认值为50

若不设-gpu，默认用cpu测试

$ **caffe time**

-model examples/mnist/lenet\_train\_test.prototxt /

-weights examples/mnist/lenet\_iter\_10000.caffemodel /

-gpu 0 -iterations 10

**训练和测试自己的图片**

1. 图像数据 -> db（leveldb/lmdb)

# sudo vi examples/images/create\_filelist.sh

# sudo vi examples/images/create\_lmdb.sh

1. 计算均值并保存

图片减去均值再训练，会提高训练速度和精度

# sudo build/tools/compute\_image\_mean examples/myfile/img\_train\_lmdb examples/myfile/mean.binaryproto

caffe训练的时候减去pixel-mean和image-mean的区别？

假定训练集：(N, 3, H, W)

mean.binaryproto文件 = image mean (3, H, W): 把所有训练集在同一个空间位置上的像素的对应通道求了均值

(B\_mean,G\_mean,R\_mean) = pixel mean：不考虑空间位置，相当于把image mean再求了一次均值。

1. 创建模型并编写配置文件

# sudo cp models/bvlc\_reference\_caffenet/solver.prototxt examples/myfile/

# sudo cp models/bvlc\_reference\_caffenet/train\_val.prototxt examples/myfile/

1. 训练

# sudo build/tools/caffe train -solver examples/myfile/solver.prototxt

example:

下载mnist数据, 在 data/mnist/目录下有四个文件

train-images-idx3-ubyte: 训练集样本 (9912422 bytes)

train-labels-idx1-ubyte: 训练集对应标注 (28881 bytes)

t10k-images-idx3-ubyte: 测试集图片 (1648877 bytes)

t10k-labels-idx1-ubyte: 测试集对应标注 (4542 bytes)

caffe$ sudo sh data/mnist/get\_mnist.sh

解析手写体，从而获取图像，代码见

<https://www.jianshu.com/p/84f72791806f>

转换成LMDB数据

在 examples/mnist/目录下，生成两个文件夹

mnist\_train\_lmdb

data.mdb

lock.mdb

mnist\_test\_lmdb

caffe$ sudo sh examples/mnist/create\_mnist.sh

训练

$ sudo time sh examples/mnist/train\_lenet.sh

or

$ caffe train -solver examples/mnist/lenet\_solver.prototxt

若没有权限访问mnist

$sudo chmod -R 777 .

自己的数据集

图像分类

create test.txt/train.txt

test/coloinside/2404.jpg 1

test/bodyoutside/1194.jpg 0

...

images -> lmdb

$build/tools/convert\_imageset --resize\_height=224 --resize\_width=224 --shuffle=True --gray=False root\_dir .../train.txt .../lmdb/train

注意：

{root\_dir}/{line} line表示train.txt里的每一行，表示图像文件名+label,

需要先创建空文件夹lmdb

**Interfaces--Python**

安装

$ make pycaffe

$ sudo gedit ~/.bashrc

export PYTHONPATH=/path/to/caffe/python:$PYTHONPATH

**caffe.Net:** is the central interface for loading, configuring, and running models. **caffe.Classifier and caffe.Detector:** provide convenience interfaces for common tasks.

**caffe.SGDSolver:** exposes the solving interface.

**caffe.io:** handles input / output with preprocessing and protocol buffers.

**caffe.draw:** visualizes network architectures.

Caffe blobs are exposed as numpy ndarrays for ease-of-use and efficiency.

from caffe import layers, params

import caffe

**layers**里面包含了Caffe内置的层（比如卷积，ReLU等），

**params**包含了各种枚举值

net=caffe.NetSpec() //获取Caffe的一个Net

...填充这个net

...最后输出到文件即prototxt

1. **用代码来生成train.prototxt, test.prototxt, solver.prototxt文件**

**deploy文件最好手动生成**

# 添加caffe/python路径

import sys

sys.path.insert(0, "python")

import numpy as np

from caffe import layers as L, params as P

import caffe

data\_dir = 'examples/mnist'

file\_dir = 'examples/qzlin/mnist'

def create\_net(lmdb, batch\_size, include\_acc=False):

data, label = **L.Data**(source=lmdb, backend=P.Data.LMDB, batch\_size=batch\_size, ntop=2, transform\_param=dict(scale=0.00390625))

conv1 = **L.Convolution**(data, kernel\_size=5, stride=1, num\_output=16, pad=2, weight\_filler=dict(type='xavier'))

relu1 = **L.ReLU**(conv1, in\_place=True)

pool1 = **L.Pooling**(relu1, pool=P.Pooling.MAX, kernel\_size=3, stride=2)

conv2 = L.Convolution(pool1, kernel\_size=3, stride=1, num\_output=32, pad=1, weight\_filler=dict(type='xavier'))

relu2 = L.ReLU(conv2, in\_place=True)

pool2 = L.Pooling(relu2, pool=P.Pooling.MAX, kernel\_size=3, stride=2)

fc3 = **L.InnerProduct**(pool2, num\_output=1024, weight\_filler=dict(type='xavier'))

relu3 = L.ReLU(fc3, in\_place=True)

drop3 = **L.Dropout**(relu3, in\_place=True)

fc4 = L.InnerProduct(drop3, num\_output=10, weight\_filler=dict(type='xavier'))

loss = **L.SoftmaxWithLoss**(fc4, label)

if include\_acc:

acc = **L.Accuracy**(fc4, label)

return **caffe.to\_proto**(loss, acc)

else:

return caffe.to\_proto(loss)

def write\_net(train\_proto=file\_dir+'/train.prototxt',

test\_proto=file\_dir+'/test.prototxt',

train\_lmdb=data\_dir+'/mnist\_train\_lmdb',

test\_lmdb=data\_dir+'/mnist\_test\_lmdb'):

with open(train\_proto, 'w') as f:

f.write(str(create\_net(train\_lmdb, batch\_size=64)))

with open(test\_proto, 'w') as f:

f.write(str(create\_net(test\_lmdb, batch\_size=100, include\_acc=True)))

def create\_solver(train\_proto, test\_proto):

s = **caffe.proto.caffe\_pb2.SolverParameter**()

s.train\_net = train\_proto

s.test\_net.append(test\_proto)

# 每遍历完整个训练集,测试一次

s.test\_interval = int(np.ceil(50000.0/64))

# 20为等价于keras的epoches

s.max\_iter = s.test\_interval \* 20

# 测试集迭代次数 = 测试集/batch\_size

s.test\_iter.append(int(10000.0/100))

s.base\_lr = 0.01

s.momentum = 0.9

s.weight\_decay = 0.0005

s.lr\_policy = 'inv'

s.gamma = 0.0001

s.power = 0.75

s.display = s.test\_interval

s.snapshot = s.max\_iter/2

s.snapshot\_prefix = file\_dir+'shapshot'

s.solver\_mode = caffe.caffe\_pb2.SolverParameter.GPU

return s

def write\_solver(train\_proto=file\_dir+'/train.prototxt',

test\_proto=file\_dir+'/test.prototxt',

solver\_file=file\_dir+'/solver.prototxt'):

s = create\_solver(train\_proto, test\_proto)

with open(solver\_file, 'w') as f:

f.write(str(s))

deploy.prototxt

<https://github.com/BVLC/caffe/wiki/Using-a-Trained-Network:-Deploy>

1. Remove the data layer that was used for training
2. Remove any layer that is dependent upon data labels.
3. Set the network up to accept data.
4. Have the network output the result.

和test.prototxt文件差不多，只是头尾不相同。

deploy文件

第一层数据输入层只有维度信息

没有最后的Accuracy层，换成输出层(Softmax概率层)

name: "Mnist"

layer {

name: "data"

type: "Input"

top: "Data1"

input\_param { shape: { dim: 1 dim: 1 dim: 28 dim: 28 } }

}

**input: "Data1"**

**input\_dim: 1**

**input\_dim: 1**

**input\_dim: 28**

**input\_dim: 28**

输入层，以上左右框演示，任选一个

...

test.prototxt去除头尾层

...

layer {

name: "prob"

type: "Softmax"

bottom: "InnerProduct2"

top: "prob"

}

1. **剖析模型**

<https://github.com/BVLC/caffe/blob/master/examples/net_surgery.ipynb>

训练好的模型:

**.caffemodel**

这个文件里面存放的就是各层的参数，即net.params，里面没有数据(net.blobs)

**.solverstate**文件，这个和caffemodel差不多，但它多了一些数据，如模型名称、当前迭代次数; solverstate是用来恢复训练的，防止意外终止而保存的快照

导入模型

net = caffe.Net(deploy, caffemodel, caffe.TEST)

# 获取模型权重和偏置等参数

for layer\_name, param in net.params.items():

print layer\_name + '\t' + str(param[0].data.shape), str(param[1].data.shape)

根据blobs名称获取输出数据

net.blobs['data'].data #取出数据层blob 1\*c\*h\*w

net.blobs['conv'].data #取出conv层输出 1\*16\*h\*w

根据层名称获取层权重和偏置

net.params['conv'][0].data #取出conv权重 (16, 3, 5, 5)

#[‘conv’][1].data为偏置

# 获取各层**blobs**

net.blobs[input\_name].data[...] = input //首先输入数据

net.forward() //运行网络前向计算

for layer\_name, blob in net.blobs.items(): //获取各个层blobs

print layer\_name + '\t' + str(blob.data.shape)

可视化卷积核和特征图，见链接

用t-SNE或PCA降维可视化高维特征

from sklearn.manifold import TSNE

每个数据

ip = net.blobs['ip'].data 假定ip2是100维向量

model = TSNE(n\_compoents=2)

ip\_vis = model.fit\_transform(ip)

scatter画数据集

1. **模型Finetune**

**split and join模型**

训练后的prototxt and caffemodel， 层参数是一一对应的

创建网络时，prototxt层参数可以比caffemodel少，比如剪裁版prototxt, 那么创建的网络只导入prototxt层参数，然后保存该网络，只获取prototxt对应的caffemodel

例子：deploy.txt + deploy.caffemodel ->

net1.prototxt, net1.caffemodel

net2.prototxt, net2.caffemodel

net3.prototxt, net3.caffemodel

deploy.prototxt

net3.prototxt

net2.prototxt

net1.prototxt

name: "Mnist"

input: "data"

input\_shape { dim: 1 dim: 1 dim: 28 dim: 28}

layer {

name: "Convolution1"

type: "Convolution"

bottom: "Data1"

...

}

...

layer {

name: "Pooling2"

type: "Pooling"

bottom: "Convolution2"

top: "Pooling2"

...

}

...

layer {

name: "InnerProduct1"

type: "InnerProduct"

bottom: "Pooling2"

top: "InnerProduct1"

...

}

layer {

name: "InnerProduct2"

type: "InnerProduct"

bottom: "InnerProduct1"

top: "InnerProduct2"

...

}

layer {

name: "prob"

type: "Softmax"

bottom: "InnerProduct2"

top: "prob"

}

name: "net1"

input: "data"

input\_shape { dim: 1 dim: 1 dim: 28 dim: 28}

layer {

name: "Convolution1"

type: "Convolution"

bottom: "Data1"

...

}

...

layer {

name: "Pooling2"

type: "Pooling"

bottom: "Convolution2"

top: "Pooling2"

...

}

name: "net2"

input: "Pooling2"

input\_shape: { dim: 1 dim: 32 dim: 7 dim: 7 }

layer {

name: "InnerProduct1"

type: "InnerProduct"

bottom: "Pooling2"

...

}

...

layer {

name: "Dropout1"

type: "Dropout"

bottom: "InnerProduct1"

top: "InnerProduct1"

}

name: "net3"

input: "InnerProduct1"

input\_shape: { dim: 1 dim: 1dim: 1 dim: 1024 }

layer {

name: "InnerProduct2"

type: "InnerProduct"

bottom: "InnerProduct1"

top: "InnerProduct2"

...

}

layer {

name: "prob"

type: "Softmax"

bottom: "InnerProduct2"

top: "prob"

}

CAFFE\_MODEL = 'examples/qzlin/mnist/snapshot\_iter\_15640.caffemodel'

def create\_caffemodel\_by\_prototxt(keyname):

deploy = 'mnist/{}.prototxt'.format(keyname)

net = caffe.Net(deploy, CAFFE\_MODEL, caffe.TEST)

net.save('mnist/{}.caffemodel'.format(keyname))

for i in range(1, 4):

create\_caffemodel\_by\_prototxt("step{}".format(i))

级联使用net1.caffemodel, net2.caffemodel, net3.caffemodel

im = caffe.io.load\_image('mnist/0.png', False)

input = im[np.newaxis, :, :, :].transpose(0, 3, 1, 2) # hwc -> 1chw

net1 = dig\_net(input, input\_name='Data1',

deploy='mnist/step1.prototxt',

caffemodel='mnist/step1.caffemodel')

Pooling2 = net1.blobs['Pooling2'].data

net2 = dig\_net(Pooling2, input\_name='Pooling2', deploy, caffemodel)

InnerProduct1 = net2.blobs['InnerProduct1'].data

net3 = dig\_net(InnerProduct1, input\_name='InnerProduct1', deploy, caffemodel)

prob = net3.blobs['prob'].data[0]

**Finetune模型**

模型迁移：只要网络定义中出现的层的名字在caffemodel文件中能够找到就可以读取，否则就按一般情况初始化

冻结低层，只训练高层模型： 将低层学习率参数lr\_mult设置为0，那么低层就不会进行梯度更新，前向计算包括低层+高层,后向计算只含高层

训练可能需要11G显卡内存，因为动态规化法要保留各个层的特征图

测试可能只需要3G以下的显卡内存，只需要前向计算，不保留特征图

迁移学习冻结低层，只训练高层，需要显卡内存和测试差不多

1. **训练--绘制loss and accuracy曲线**

准备数据集

caffe\_root$ ./data/cifar10/get\_cifar10.sh 下载数据集

caffe\_root$ ./examples/cifar10/create\_cifar10.sh转化为lmdb格式(含均值文件)

def loss\_accuracy():

# 注意：数据路径,模型,优化等都包含在内

# 1. 无论模型中Slover类型是什么统一设置为SGD

solver = caffe.SGDSolver('examples/cifar10/cifar10\_quick\_solver.prototxt')

# 2. 根据solver的prototxt中solver\_type读取，默认为SGD

solver = caffe.get\_solver('/home/xxx/data/solver.prototxt')

#方式1: 训练模型

# solver.solve()

# 方式2: 迭代训练模型

niter = 4000

test\_interval = 200

train\_loss = np.zeros(niter)

test\_acc = np.zeros(int(np.ceil(niter/test\_interval)))

for it in range(niter):

#进行一次前向传播一次反向传播并根据梯度更新参数

#solver.net.forward() # 前向传播

#solver.net.backward() # 反向传播,计算梯度

solver.step(1)

train\_loss[it] = solver.net.blobs['loss'].data

solver.test\_nets[0].forward(start='conv1')

if it % test\_interval == 0:

acc = solver.test\_nets[0].blobs['accuracy'].data

test\_acc[it // test\_interval] = acc

保存模型

solver.net.save('mymodel.caffemodel')

1. **测试**

def evaluate():

#加载Model

net\_file = '.../deploy.prototxt'

caffe\_model = '....caffemodel'

net = caffe.Net(net\_file, caffe\_model, caffe.TEST)

# 图像预处理

# 设定图片的shape格式为网络data层格式

transformer = caffe.io.Transformer({'data': net.blobs['data'].data.shape})

#HWC -> CHW

transformer.set\_transpose('data', (2,0,1))

#去均值

transformer.set\_mean('data', np.load('\*\_mean.npy').mean(1).mean(1))

# rescale -> [0, 255]

transformer.set\_raw\_scale('data', 255)

# RGB -> BGR

transformer.set\_channel\_swap('data', (2,1,0))

# 加载图片

im\_test = caffe.io.load\_image(caffe\_root+'examples/images/cat.jpg')

# 执行预处理

net.blobs['data'].data[...] = transformer.preprocess('data',im\_test)

# 执行测试

out = net.forward()

# 排序softmax(即类别概率),显示对应label

imagenet\_labels\_filename = caffe\_root + 'data/ilsvrc12/synset\_words.txt'

labels = np.loadtxt(imagenet\_labels\_filename, str, delimiter='\t')

top\_k = net.blobs['prob'].data[0].flatten().argsort()[-1:-6:-1]

for i in np.arange(top\_k.size):

print top\_k[i], labels[top\_k[i]]

**Interfaces--C++**

Caffe源代码

data 用于存放下载的训练数据

docs 帮助文档

example 一些代码样例

matlab MATLAB接口文件

python Python接口文件

model 一些配置好的模型参数

scripts 一些文档和数据用到的脚本

tools 保存的源码是用于生成二进制处理程序的，训练时直接调用这些二进制文件

include Caffe的实现代码的头文件

src 实现Caffe的源文件

util 数据转换时用的一些代码

proto 数据存储格式Protobuf

layers

caffe核心代码

blob[.cpp .h] 基本的数据结构Blob类。

common[.cpp .h] 定义Caffe类

internal\_thread[.cpp .h] 使用boost::thread线程库

net[.cpp .h] 网络结构类Net

solver[.cpp .h] 优化方法类Solver

data\_transformer[.cpp .h] 输入数据的基本操作类DataTransformer

syncedmem[.cpp .h] 分配内存和释放内存类CaffeMallocHost，用于同步GPU，CPU数据

layer\_factory.cpp layer.h 层类Layer

Caffe IDE

<http://suanfazu.com/t/eclipse-caffe/13450>

<https://blog.csdn.net/mounty_fsc/article/details/51089864>

$nsight 打开 IDE

File -> New -> Project -> C/C++ -> Makefile Project with Existing Code

工程配置：

Project name： caffe

Existing code location：/home/qzlin/Documents/dl/caffe

Language：C and C++

Toolchain：Linux GCC

Run -> Debug Configurations -> C/C++ Application -> New

Name: RefineDet Default

Main: /home/qzlin/Documents/dl/RefineDet/build/tools/caffe

Arguments: train -solver=.../solver.prototxt

working directory: ...

编译： Project -> Projecties -> build command “make” 改为”make -j”

Project -> clean, Project -> Build All

注意：**将Makefile.config注释掉 DEBUG :=1** 这一行来打开调试信息并关闭优化(gcc flag -g -O0)

Caffe的整体流程图

<https://www.cnblogs.com/liuzhongfeng/p/7289956.html>

tools/caffe.cpp

void main(输入参数train -solover=solover.prototxt) {

GetBrewFunction("train")获取train()

}

int train() {

//从solver.prototxt读取solver\_param

SolverParameter solver\_param = ReadSolverParamsFromTextFileOrDie();

//从参数创建solve

**Solver solver = CreateSolver(solver\_param);**

// 从snapshot恢复模型

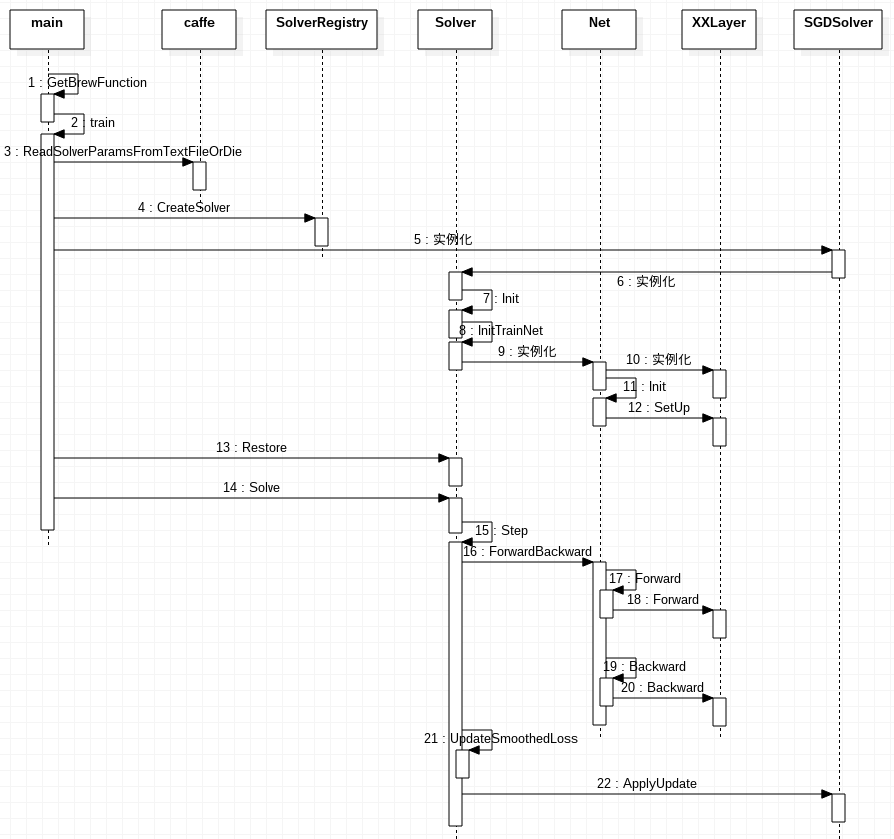
solver->Restore(snapshot);

//若采用finetuning，则拷贝weight到指定模型

CopyLayers(solver.get(), FLAGS\_weights);

**solver->Solve(); //开始训练网络**

}



初始化总体流程

1. 新建一个Solver对象
2. 在Solver的构造函数中新建Net类实例
3. 在Net类的构造函数中新建各个layer实例

设置每个Blob

void Solver::InitTrainNet() {

// 获取NetParameter from SolverParameter

NetParameter net\_param;

net\_param.CopyFrom(param\_.net\_param());

ReadNetParamsFromTextFileOrDie(param\_.net(), &net\_param);

// 实例化net

net =new Net(net\_param);

}

void Net::Init(const NetParameter& in\_param) {

FilterNet(in\_param, &filtered\_param); //过滤校验参数FilterNet

InsertSplits(filtered\_param, &param);//插入Splits层

...// 构建网络中输入输出存储结构

for (int layer\_id = 0; layer\_id < param.layer\_size(); ++layer\_id) {

//创建层, 然后调用REGISTER\_LAYER\_CLASS(AnnotatedData);从而实例化AnnotatedDataLayer

layers.push\_back(LayerRegistry::CreateLayer(layer\_param));

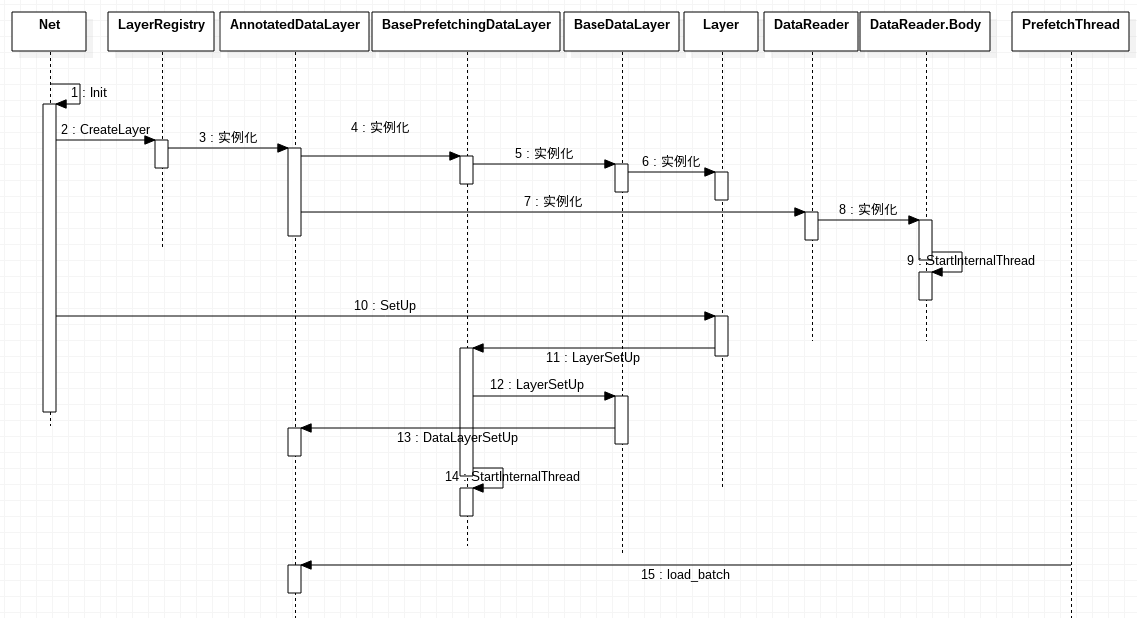
........//创建相关blob

//执行SetUp()

layers\_[layer\_id]->SetUp(bottoms[layer\_id], tops[layer\_id]);

}

}



训练

void Solver::Step(int iters) {

while (iter\_ < stop\_iter) {

net\_->ClearParamDiffs(); // 将net\_中的Bolb梯度参数置为零

// accumulate the loss and gradient

Dtype loss = 0;

for (int i = 0; i < param\_.iter\_size(); ++i) {

loss += net\_->ForwardBackward(); // 正向传导和反向传导，并计算loss

}

loss /= param\_.iter\_size();

// 为了输出结果平滑，将临近的average\_loss个loss数值进行平均，存储在成员变量smoothed\_loss\_

UpdateSmoothedLoss(loss, start\_iter, average\_loss);

ApplyUpdate(); // BP算法更新权重

++iter\_;

}

}

void SGDSolver::ApplyUpdate() {

Dtype rate = GetLearningRate(); // 获取当前学习速率

// 在计算当前梯度的时候，如果该值超过了阈值clip\_gradients，则将梯度直接设置为该阈值，

//此处阈值设为-1，即不起作用

ClipGradients();

// 逐层更新网络中的可学习层

for (int id = 0; id < this->net\_->learnable\_params().size(); ++id) {

Normalize(id); // 归一化

Regularize(id); // L2范数正则化添加衰减权重

ComputeUpdateValue(id, rate); // 随机梯度下降法计算更新值

}

this->net\_->Update(); // 更新权重

}

若solver.prototxt不仅仅配置train.prototxt, 还配置了test.prototxt,那么训练和测试交替进行

solver.prototxt

train\_net: "models/vgg16/VOC0712/refinedet\_vgg16\_320x320/train.prototxt"

test\_net: "models/vgg16/VOC0712/refinedet\_vgg16\_320x320/test.prototxt"

test\_iter: 4952 //在测试的时候，需要迭代的次数，一般test\_iter = test\_size/test\_batch\_size

**test\_interval: 1**  //经过多少次train\_batch\_size的训练，然后进行一次测试

...

**eval\_type: "detection"**

ap\_version: "11point"

src/caffe/sover.cpp

template <typename Dtype>

void Solver<Dtype>::Solve(const char\* resume\_file) {

Step(...);

UpdateSmoothedLoss(loss, start\_iter, average\_loss);

**TestAll();**

}

template <typename Dtype>

void Solver<Dtype>::TestDetection(const int test\_net\_id) {

map<int, map<int, vector<pair<float, int> > > > all\_true\_pos;

map<int, map<int, vector<pair<float, int> > > > all\_false\_pos;

map<int, map<int, int> > all\_num\_pos;

const shared\_ptr<Net<Dtype> >& test\_net = test\_nets\_[test\_net\_id];

const vector<Blob<Dtype>\*>& result = test\_net->Forward(&iter\_loss);

all\_num\_pos[imageid][label] += result\_vec[num\_det \* 5 + 2];

float score = result\_vec[k \* 5 + 2];

int tp = result\_vec[num\_det \* 5 + 3];

int fp = result\_vec[num\_det \* 5 + 4];

all\_true\_pos[imageid][label].push\_back(score, tp);

all\_false\_pos[imageid][label].push\_back(score, fp);

for (int i = 0; i < all\_true\_pos.size(); ++i) {

const map<int, vector<pair<float, int> > >& true\_pos = all\_true\_pos.find(i)->second;

const map<int, vector<pair<float, int> > >& false\_pos = all\_false\_pos.find(i)->second;

const map<int, int>& num\_pos = all\_num\_pos.find(i)->second;

//对于每幅图像

map<int, float> APs;

float mAP = 0.;

for (it = num\_pos.begin(); it != num\_pos.end(); ++it) {

//对于每个类别

int label = it->first;

int label\_num\_pos = it->second;

const vector<pair<float, int> >& label\_true\_pos = true\_pos.find(label)->second;

const vector<pair<float, int> >& label\_false\_pos = false\_pos.find(label)->second;

//计算每个类别检测框的precision, recall and AP

// **precision(检测框数量), recall(检测框数量)**, AP（数值)

//不确定是检测框数量还是？举例：prec[2076]?检测框数量不是<500吗？

vector<float> prec, rec;

**ComputeAP**(label\_true\_pos, label\_num\_pos, label\_false\_pos,

param\_.ap\_version(), &prec, &rec, &(APs[label]));

//对所有类别的AP求平均

mAP += APs[label];

}

mAP /= num\_pos.size();

}

}

Net

用容器的形式将多个Layer有序地放在一起，主要功能是**逐层对Layer初始化，以及更新网络参数（提供Update( )的接口）**，本身不能对参数进行有效地学习过程

vector<shared\_ptr<Layer<Dtype> > > layers\_; //构成该net的layers

vector<vector<Blob<Dtype>\*> > bottom\_vecs\_; //每一层layer中的bottom Blobs

vector<vector<Blob<Dtype>\*> > top\_vecs\_; //每一层layer中的top Blobs

vector<shared\_ptr<Blob<Dtype> > > params\_; //整个net中的learnable parameter

//根据NetParameter进行net初始化,简单的来说就是先把网络中所有层的bottom Blobs&top Blobs（无重复）实例化，并从输入层开始，逐层地进行Setup的工作，从而完成了整个网络的搭建，为后面的数据前后传输打下基础

void Init(const NetParameter& param);

//对整个网络的前向和方向传导，各调用一次就可以计算出网络的loss

vector<Blob\*>& Forward(const vector<Blob<Dtype>\* > & bottom,Dtype\* loss = NULL)

void Net<Dtype>::Backward()

Solver

Solver类中包含一个Net的指针，主要是实现了训练模型参数所采用的优化算法，根据优化算法的不同会派生不同的类，而基于这些子类就可以对网络进行正常的训练过程

shared\_ptr<Net<Dtype> > net\_; //net对象

//对已初始化后的网络进行固定次数的训练迭代过程

void Step(int iters)

// 不同的模型训练方法通过重载函数ComputeUpdateValue( )实现计算update参数的核心功能

ComputeUpdateValue();

net\_->Update();

若$ caffe test -model test.prototxt -weights final.caffemodel

int test() {

Caffe::SetDevice(gpus[0]);

Caffe::set\_mode(Caffe::GPU);

// Instantiate the caffe net.

Net<float> caffe\_net(FLAGS\_model, caffe::TEST, FLAGS\_level, &stages);

caffe\_net.CopyTrainedLayersFrom(FLAGS\_weights);

vector<int> test\_score\_output\_id;

vector<float> test\_score;

float loss = 0;

for (int i = 0; i < FLAGS\_iterations; ++i) {

float iter\_loss;

//获取最后一层输出即detection\_eval

const vector<Blob<float>\*>& result = caffe\_net.Forward(&iter\_loss);

loss += iter\_loss;

//统计并显示测试结果

}

loss /= FLAGS\_iterations;

...

}

Caffe test case调试

基于caffe train/test/time调试搭好nsight环境

RefineDet$ make -j

RefineDet$ make test -j

右击src/caffe/test/test\_detection\_evaluate\_layer.cpp -> Debug Configuration -> New C/C++ Application

Name: RefineDet-test-detection-evaluate-layer

Main:.../RefineDet/build/test/test\_detection\_evaluate\_layer.testbin

如下解决无法进入源代码断点：

<https://blog.csdn.net/adaptiver/article/details/72459323?locationNum=11&fps=1>

Select other... -> Legacy Create Process Launcher -> ok

Debugger -> Use full file path to set breakpoints

caffe.proto

<https://blog.csdn.net/zr459927180/article/details/50904938>

caffe中，数据的读取、运算、存储都是采用**Google Protocol Buffer (PB)**, PB是一种轻便、高效的结构化数据存储格式，可以用于结构化数据串行化，很适合做数据存储或 RPC 数据交换格式。它可用于通讯协议、数据存储等领域的语言无关、平台无关、可扩展的序列化结构数据格式。是一种效率和兼容性都很优秀的二进制数据传输格式，目前提供了 C++、Java、Python 三种语言的 API。Caffe采用的是C++和Python的API。caffemodel存储的数据也就是网络参数net\_param的PB

helloworld.proto

package lm

message helloworld

{

required int32 id = 1;

repeated string str = 2;

optional int32 opt = 3;

}

proto文件类似C++语言数据定义, Message等价于class

修饰符

required表明是必须包含该成员

optional表明可选成员

repeated等价于vector数组

-> PB编译器(protoc) -> c++

lm.helloworld.pb.h

lm.helloworld.pb.cc

#include "lm.helloworld.pb.h"

lm::helloworld msg1;

msg1.set\_id(101);

msg1.set\_str(“hello”);

fstream output("./log", ios::out | ios::trunc | ios::binary);

msg1.SerializeToOstream(&output)

fstream input("./log", ios::in | ios::binary);

msg1.ParseFromIstream(&input)

Blobs

数据封装包(输入数据、输出数据、权值)，并且在CPU与GPU之间具有同步处理能力。按C风格连续存储的N维数组(N, C, H, W), 同时保存data and diff(梯度）

template <typename Dtype>

class Blob {

public:

explicit Blob(const vector<int>& shape);

void Reshape(const vector<int>& shape); //改变blob维度，分配内存

void ReshapeLike(const Blob& other);

string shape\_string() const; //字符串形式显示大小

const vector<int>& shape() const; //获取数据维度

//拷贝数据作为数据或者作为梯度，reshape强迫与blob shape一致

void CopyFrom(const Blob<Dtype>& source, bool copy\_diff = false, bool reshape = false);

//反序列化: 从proto读数据

void FromProto(const BlobProto& proto, bool reshape = true);

//把blob数据保存到proto

void ToProto(BlobProto\* proto, bool write\_diff = false) const;

Dtype data\_at(const vector<int>& index) const ; //根据下标取值

Dtype diff\_at(const vector<int>& index) const ;

const Dtype\* cpu\_data() const; //获取数据指针 (不改变数据)

Dtype\* mutable\_cpu\_data(); //...(可改变数据)

const Dtype\* cpu\_diff() const;

Dtype\* mutable\_cpu\_diff();

const Dtype\* gpu\_data() const;

Dtype\* mutable\_gpu\_data();

const Dtype\* gpu\_diff() const;

Dtype\* mutable\_gpu\_diff();

Dtype asum\_data() const; //L1 norm

Dtype asum\_diff() const;

Dtype sumsq\_data() const; //L2 norm squared

Dtype sumsq\_diff() const;

void scale\_data(Dtype scale\_factor); //缩放

void scale\_diff(Dtype scale\_factor);

//传指针（注意：排他性)

void ShareData(const Blob& other);

void ShareDiff(const Blob& other);

protected:

shared\_ptr<SyncedMemory> data\_; //数据

shared\_ptr<SyncedMemory> diff\_; //梯度

shared\_ptr<SyncedMemory> shape\_data\_;

vector<int> shape\_;

};

Layers

vector<Dtype> loss\_ ; //每一层都会有一个loss值，但只有LossLayer才会产生非0的loss

vector<shared\_ptr<Blob<Dtype> > > blobs\_ ; //Layer所学习的参数，包括权值和偏差

//通过bottom Blob和LayerParameter(从prototxt读入)**确定Layer的学习参数的形状**

void LayerSetUp(const vector<Blob<Dtype>\*>& bottom, const vector<Blob<Dtype>\*>& top)

//通过bottom Blob对象的形状以及Layer的学习参数的形状来**确定top Blob对象的形状**

void Reshape(const vector<Blob<Dtype>\*>& bottom, const vector<Blob<Dtype>\*>& top)

// Layer内部数据正向传播，从bottom到top方向

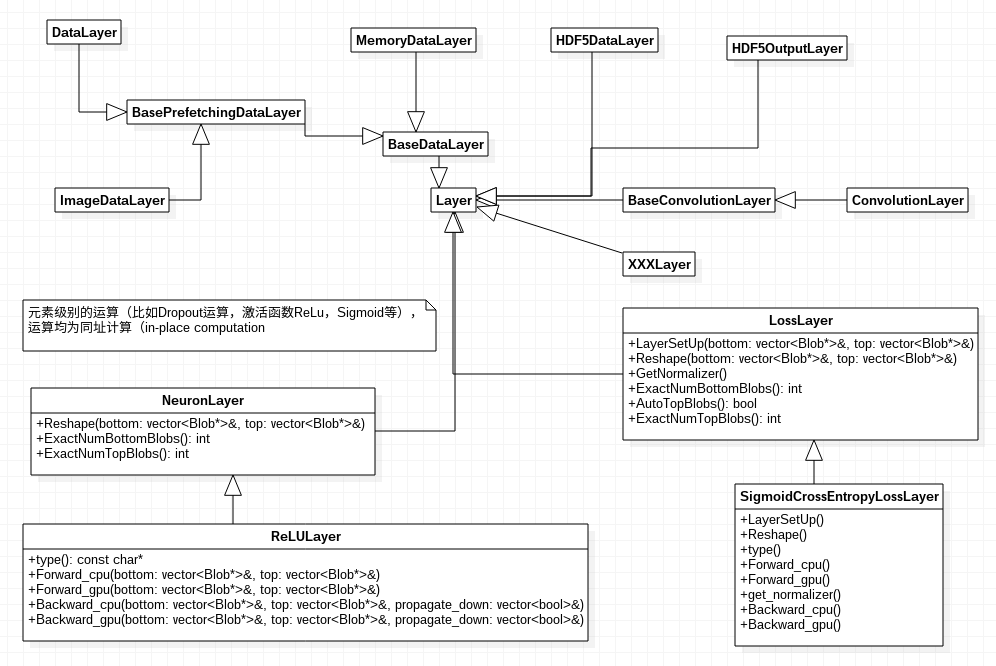
void Forward(const vector<Blob<Dtype>\*> &bottom, vector<Blob<Dtype>\*> \*top) = 0

// Layer内部梯度反向传播，从top到bottom方向

void Backward(const vector<Blob<Dtype>\*> &top,

const vector<bool> &propagate\_down,

vector<Blob<Dtype>\*> \*bottom) = 0



1. Data Layers

2.Vision Layers

3.Neuron Layers: 逐点运算层(如Activation)继承于Neuron Layer

4.Loss Layers

5.Common Layers (InnerProductLayer, SoftmaxLayer, Accuracy / Top-k layer , NormalizeLayer, RecurrentLayer, ...)直接继承于Layer

自定义层

实现涉及的文件

include/caffe/layers/your\_layer.hpp

src/caffe/layers/your\_layer.cpp

src/caffe/layers/your\_layer.cu (若实现Forward\_gpu,Backward\_gpu)

your\_layer继承common\_layer.hpp, data\_layer.hpp, loss\_layer.hpp, neuron\_layer.hpp, vision\_layer.hpp

// 为了在写net.prototxt时，layer{type："YourLayerName"}有所对应

virtual inline const char\* type() const { return "YourLayerName"; }

// **限制bottom和top的blob个数**

virtual inline int ExactNumBottomBlobs() const { return 0; }

virtual inline int MinBottomBlobs() const { return -1; }

virtual inline int MaxBottomBlobs() const { return -1; }

virtual inline int ExactNumTopBlobs() const { return 1; }

virtual inline int MinTopBlobs() const { return -1; }

virtual inline int MaxTopBlobs() const { return -1; }

virtual inline bool EqualNumBottomTopBlobs() const { return false; }

//If true, Net::Init will create enough "anonymous" top blobs

virtual inline bool AutoTopBlobs() const { return false; }

// Layer初始化: 读取layer的参数，**权重进行初始化等**

virtual void LayerSetUp(vector<Blob\*>& bottom, vector<Blob\*>& top);

// layer初始化: 根据bottom的shape，**修改top的shape**

virtual void Reshape(vector<Blob\*>& bottom, vector<Blob\*>& top) = 0;

// 前向传播计算loss和top，反向传播计算diff(梯度)

Dtype Forward\_cpu(vector<Blob\*>& bottom, vector<Blob\*>& top);

void Backward\_cpu(...);

在.cpp文件末尾注册Layer，便于**运行时统一创建**

INSTANTIATE\_CLASS(XXXLayer);

REGISTER\_LAYER\_CLASS(XXX);

在src/caffe/proto/caffe.proto添加your\_layer的message

如果想要在net.prototxt中设置你的layer的参数的话，你需要在caffe.proto中定义

message **AnnotatedDataParameter** {

...

}

message LayerParameter {

**optional AnnotatedDataParameter annotated\_data\_param = 200;**

}

CAFFE\_ROOT$ make clean

CAFFE\_ROOT$ make -j

**举例：**

**全通过层AllPassLayer(仅仅作为教学)**

增加对应cpp、h、cu的声明和实现,修改caffe.proto文件，编译caffe库即可

1. **$CAFFE\_ROOT/include/caffe/layers/AllPassLayer.hpp**

#ifndef CAFFE\_ALL\_PASS\_LAYER\_HPP\_

#define CAFFE\_ALL\_PASS\_LAYER\_HPP\_

#include <vector>

#include "caffe/blob.hpp"

#include "caffe/layer.hpp"

#include "caffe/proto/caffe.pb.h"

#include "caffe/layers/neuron\_layer.hpp"

class AllPassLayer : public NeuronLayer<Dtype> {

public:

explicit AllPassLayer(const LayerParameter& param) : NeuronLayer<Dtype>(param) {}

virtual inline const char\* type() const { return "AllPass"; }

protected:

virtual void Forward\_cpu(...);

virtual void Forward\_gpu(...);

virtual void Backward\_cpu(...);

virtual void Backward\_gpu(...);

};

1. **$CAFFE\_ROOT/src/caffe/layers/AllPassLayer.cpp**

#include <algorithm>

#include <vector>

#include "caffe/layers/all\_pass\_layer.hpp"

#include <iostream>

using namespace std;

void Forward\_cpu(const vector<Blob<Dtype>\*>& bottom, const vector<Blob<Dtype>\*>& top) {

const Dtype\* bottom\_data = bottom[0]->cpu\_data();

Dtype\* top\_data = top[0]->mutable\_cpu\_data();

for (int i = 0; i < bottom[0]->count(); ++i) { top\_data[i] = bottom\_data[i]; }

}

void Backward\_cpu(const vector<Blob<Dtype>\*>& top, const vector<bool>& propagate\_down,

const vector<Blob<Dtype>\*>& bottom) {

if (propagate\_down[0]) {

const Dtype\* top\_diff = top[0]->cpu\_diff();

Dtype\* bottom\_diff = bottom[0]->mutable\_cpu\_diff();

for (int i = 0; i < bottom[0]->count(); ++i) { bottom\_diff[i] = top\_diff[i]; }

}

}

INSTANTIATE\_CLASS(AllPassLayer);

REGISTER\_LAYER\_CLASS(AllPass);

1. **$CAFFE\_ROOT/src/caffe/proto/caffe.proto**

message AllPassParameter {

optional float key = 1 [default = 0];

}

message LayerParameter {

...

optional AllPassParameter all\_pass\_param = 155;

}

1. **$CAFFE\_ROOT$ make clean && make all**

使用于deploy.prototxt

name: "AllPassTest"

layer {

name: "data"

type: "Input"

top: "data"

input\_param { shape: { dim: 10 dim: 3 dim: 227 dim: 227 } }

}

layer {

name: "ap"

type: "AllPass" #类名去掉 Layer 后的名称。

bottom: "data"

top: "conv1"

all\_pass\_param { key: 12.88 }

}

测试

$CAFFE\_ROOT$ ./build/tools/caffe.bin time -model deploy.prototxt

标注数据层AnnotationDataLayer

void DataLayerSetUp(const vector<Blob<Dtype>\*>& bottom, const vector<Blob<Dtype>\*>& top) {

const int batch\_size = this->layer\_param\_.data\_param().batch\_size();

const AnnotatedDataParameter& anno\_data\_param = this->layer\_param\_.annotated\_data\_param();

for (int i = 0; i < anno\_data\_param.batch\_sampler\_size(); ++i)

batch\_samplers\_.push\_back(anno\_data\_param.batch\_sampler(i));

label\_map\_file\_ = anno\_data\_param.label\_map\_file();

// Read a data point, and use it to initialize the top blob.

AnnotatedDatum& anno\_datum = \*(reader\_.full().peek());

// Use data\_transformer to infer the expected blob shape from anno\_datum.

vector<int> top\_shape = this->data\_transformer\_->InferBlobShape(anno\_datum.datum());

top\_shape[0] = batch\_size;

this->transformed\_data\_.Reshape(top\_shape);

top[0]->Reshape(top\_shape);

for (int i = 0; i < this->PREFETCH\_COUNT; ++i) this->prefetch\_[i].data\_.Reshape(top\_shape);

// label

vector<int> label\_shape(4, 1);

int num\_bboxes = 0; //计算检测框数量

for (int g = 0; g < anno\_datum.annotation\_group\_size(); ++g)

num\_bboxes += anno\_datum.annotation\_group(g).annotation\_size();

label\_shape[0] = 1;

label\_shape[1] = 1;

label\_shape[2] = std::max(num\_bboxes, 1);

label\_shape[3] = 8; // [item\_id, group\_label, instance\_id, xmin, ymin, xmax, ymax, diff]

top[1]->Reshape(label\_shape);

for (int i = 0; i < this->PREFETCH\_COUNT; ++i) this->prefetch\_[i].label\_.Reshape(label\_shape);

}

void load\_batch(Batch<Dtype>\* batch) {

const int batch\_size = this->layer\_param\_.data\_param().batch\_size();

const AnnotatedDataParameter& anno\_data\_param = this->layer\_param\_.annotated\_data\_param();

const TransformationParameter& transform\_param = this->layer\_param\_.transform\_param();

AnnotatedDatum& anno\_datum = \*(reader\_.full().peek());

//基于transform\_param配置参数: 强制图像通道->resize大小->剪切大小, 计算变换后图像维度

vector<int> top\_shape = this->data\_transformer\_->InferBlobShape(anno\_datum.datum());

this->transformed\_data\_.Reshape(top\_shape);

top\_shape[0] = batch\_size;

batch->data\_.Reshape(top\_shape);

**Dtype\* top\_data** = batch->data\_.mutable\_cpu\_data();

// Store transformed annotation.

map<int, vector<AnnotationGroup> > all\_anno;

int num\_bboxes = 0;

for (int item\_id = 0; item\_id < batch\_size; ++item\_id) {

// get a anno\_datum

AnnotatedDatum& anno\_datum = \*(reader\_.full().pop("Waiting for data"));

AnnotatedDatum distort\_datum;

distort\_datum.CopyFrom(anno\_datum);

data\_transformer\_->**DistortImage**(anno\_datum.datum(),distort\_datum.mutable\_datum());

AnnotatedDatum\* expand\_datum = new AnnotatedDatum();

data\_transformer\_->**ExpandImage**(distort\_datum, expand\_datum);

// Generate sampled bboxes from expand\_datum.

vector<NormalizedBBox> sampled\_bboxes;

**GenerateBatchSamples**(\*expand\_datum, batch\_samplers\_, &sampled\_bboxes);

// Randomly pick a sampled bbox and crop the expand\_datum.

int rand\_idx = caffe\_rng\_rand() % sampled\_bboxes.size();

AnnotatedDatum\* sampled\_datum = new AnnotatedDatum();

data\_transformer\_->**CropImage**(\*expand\_datum,sampled\_bboxes[rand\_idx],sampled\_datum);

// Apply data transformations (mirror, scale, crop...)

int offset = batch->data\_.offset(item\_id);

this->transformed\_data\_.set\_cpu\_data(top\_data + offset);

vector<AnnotationGroup> transformed\_anno\_vec;

// Transform datum and annotation\_group at the same time

transformed\_anno\_vec.clear();

data\_transformer\_->**Transform**(\*sampled\_datum,

&(this->transformed\_data\_), &transformed\_anno\_vec);

// Count the number of bboxes.

for (int g = 0; g < transformed\_anno\_vec.size(); ++g) {

num\_bboxes += transformed\_anno\_vec[g].annotation\_size();

}

all\_anno[item\_id] = transformed\_anno\_vec;

}

// Store "rich" annotation if needed.

vector<int> label\_shape(4);

label\_shape[0] = 1;

label\_shape[1] = 1;

label\_shape[3] = 8;

label\_shape[2] = num\_bboxes;

batch->label\_.Reshape(label\_shape);

**Dtype\* top\_label** = batch->label\_.mutable\_cpu\_data();

int idx = 0;

for (int item\_id = 0; item\_id < batch\_size; ++item\_id) {

//每幅图像

const vector<AnnotationGroup>& anno\_vec = all\_anno[item\_id];

for (int g = 0; g < anno\_vec.size(); ++g) {

//每个真实框

const AnnotationGroup& anno\_group = anno\_vec[g];

for (int a = 0; a < anno\_group.annotation\_size(); ++a) {

const Annotation& anno = anno\_group.annotation(a);

const NormalizedBBox& bbox = anno.bbox();

top\_label[idx++] = item\_id;

top\_label[idx++] = anno\_group.group\_label();

top\_label[idx++] = anno.instance\_id();

top\_label[idx++] = bbox.xmin();

top\_label[idx++] = bbox.ymin();

top\_label[idx++] = bbox.xmax();

top\_label[idx++] = bbox.ymax();

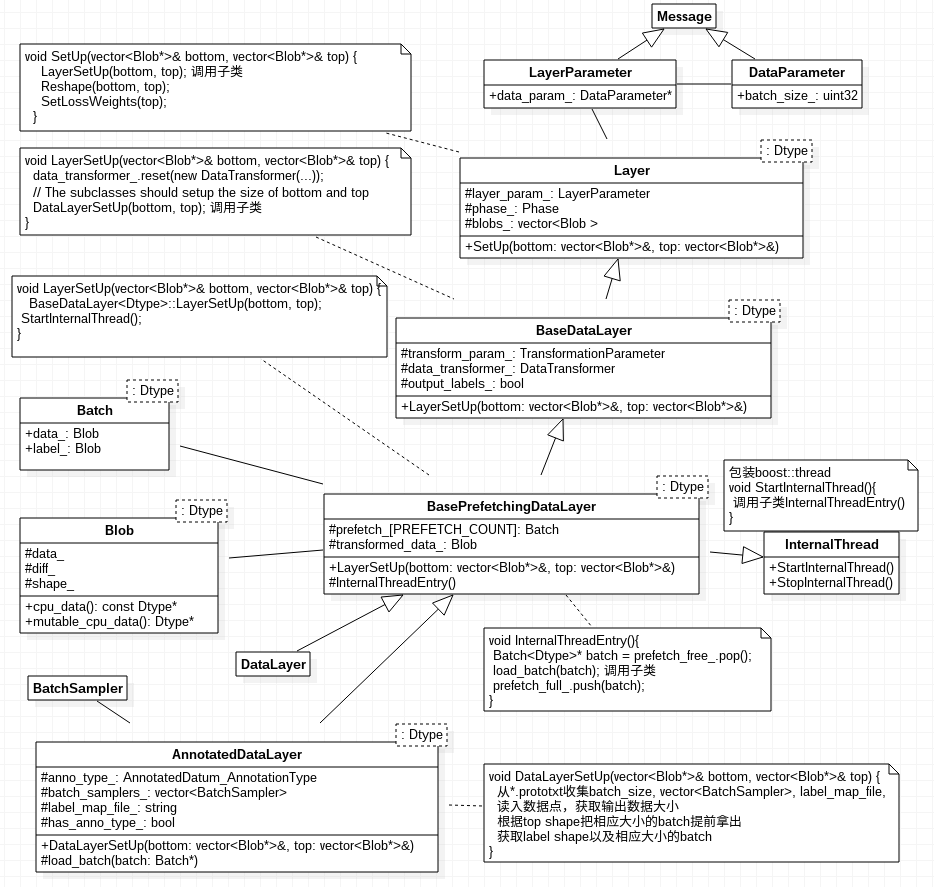
top\_label[idx++] = bbox.difficult();

}

}

}

}



数据增广

transform\_param {

mirror: true 默认为false

mean\_file: "" 或者均值mean\_value: 104.0 mean\_value: 117.0 mean\_value: 123.0

scale: XXX 默认为1

crop\_size: XXX 训练随机剪切，测试固定中心点剪切

crop\_h: XXX

crop\_w: XXX

force\_color: XXX 强制图像彩色 or 灰度

force\_gray: XXX

resize\_param { 图像大小缩放

prob: 1.0 使用缩放策略的概率，默认为1

resize\_mode: WARP 缩放模式, 还有FIT\_SMALL\_SIZE/FIT\_LARGE\_SIZE\_AND\_PAD

height: 512 缩放大小

width: 512

height\_scale: XXX 默认为0, 仅在resize\_mode=FIT\_SMALL\_SIZE使用

width\_scale: XXX 默认为0

//仅在Pad=BE\_SMALL\_SIZE\_AND\_PAD且object centering

pad\_mode: XXX 默认为CONSTANT,MIRRORED/REPEAT\_NEAREST

pad\_value: XXX

interp\_mode: LINEAR 插值模式：同OpenCV, AREA/NEAREST/CUBIC/LANCZOS4

}

emit\_constraint { emit annotation条件

emit\_type: CENTER 默认为CENTER, MIN\_OVERLAP

emit\_overlap: XXX 仅在emit\_type=MIN\_OVERLAP时使用

}

distort\_param { image distortion

brightness\_prob: 0.5 亮度

brightness\_delta: 32.0 值范围，推荐delta = 32

contrast\_prob: 0.5 对比度

contrast\_lower: 0.5

contrast\_upper: 1.5

hue\_prob: 0.5 色调

hue\_delta: 18.0 推荐delta=36

saturation\_prob: 0.5 饱和度

saturation\_lower: 0.5

saturation\_upper: 1.5

random\_order\_prob: 0.0 随机顺序图像通道概率，默认为0

}

expand\_param { image expansion

prob: 0.5

max\_expand\_ratio: 4.0 默认值1

}

noise\_param { 默认全为false

prob: XXX

hist\_eq: XXX 是否直方图均衡化，

inverse: XXX 是否色反转

decolorize: XXX 是否灰度化

gauss\_blur: XXX 是否高斯滤波

jpeg: XXX JPEG压缩度,默认为-1表示不压缩

posterize: XXX 是否光栅化

erode: XXX 是否erode

saltpepper: XXX 是否椒盐噪声

saltpepper\_param: {

fraction: XXX percentage of pixels 默认为0

value: XXX

value: XXX

}

clahe: XXX 是否局部直方图均衡化

convert\_to\_hsv: XXX 是否转换到hsv

convert\_to\_lab: XXX

}

}

src/caffe/data\_transformer.cpp

void **DistortImage**(const Datum& datum, Datum\* distort\_datum);

src/caffe/util/im\_transforms.cpp

**transform\_param中的distort\_param**

1. // Do random brightness distortion.

均匀分布[-brightness\_delta, brightness\_delta]采样作为shift

第二个参数type=-1: 表示输出与输入同矩阵类型

第三个参数scale: 表示比例因子

第三个参数shift: 表示平移量

in\_img.convertTo(out\_img, -1, 1, shift);

1. // Do random contrast distortion.

均匀分布[contrast\_lower, contrast\_upper]采样作为scale

in\_img.convertTo(out\_img, -1, scale, 0);

1. // Do random saturation distortion.

均匀分布[saturation\_lower, saturation\_upper]采样作为scale

BGR转HSV，在Saturation通道进行调节

hsv[1].convertTo(hsv[1], -1, scale, 0);

HSV转回BGR

1. // Do random hue distortion.

均匀分布[-hue\_delta, hue\_delta]采样作为shift

BGR转HSV，在Vue通道进行调节

hsv[0].convertTo(hsv[0], -1, 1, shift);

HSV转回BGR

1. // Do random reordering of the channels.

随机调整三通道顺序

void **ExpandImage**(const AnnotatedDatum& anno\_datum,AnnotatedDatum\* expanded\_anno\_datum);

// Expand the image.

均匀分布[1, max\_expand\_ratio]采样作为expand\_ratio

h\_e, w\_e = h\*expand\_ratio, w\*expand\_ratio

扩大后图像expand\_img(h\_e, w\_e, 3) = mean

均匀分布[0, h\_e-h]采样作为h\_off; 均匀分布[0, w\_e-w]采样作为w\_off

置图像于扩大图像expand\_img[w\_off, h\_off, w, h]处

从原图像归一化[0, 0, 1, 1] -> 扩大图像归一化坐标

expand\_bbox[-w\_off, -h\_off, w\_e-w\_off, h\_e-h\_off]/(w, h)=[-0.04,-0.05,1.07,1.06]

// Transform the annotation according to expand image.

过滤每个真实框

若emit\_type=CENTER 确保真实框中心在扩大后的图像里

若emit\_type= MIN\_OVERLAP 确保真实框与扩大后图像IoU > emit\_overlap

将每个真实框bbox比如[0.18,0.63,0.47, 0.77] 投影到扩大图像坐标系

xmin\_p = (bbox.xmin() - expand\_bbox.xmin()) / expand\_bbox.width

ymin\_p = (bbox.ymin() - expand\_bbox.ymin()) / expand\_bbox.height

xmax\_p = (bbox.xmax() - expand\_bbox.xmax()) / expand\_bbox.width

ymax\_p = (bbox.ymax() - expand\_bbox.ymax()) / expand\_bbox.height

annotated\_data\_param {

batch\_sampler {

max\_sample: 1

max\_trials: 1

}

batch\_sampler {

sampler { 默认值全为1

min\_scale: 0.3

max\_scale: 1.0

min\_aspect\_ratio: 0.5

max\_aspect\_ratio: 2.0

}

sample\_constraint {

min\_jaccard\_overlap: 0.1 {0.1, 0.3, 0.5, 0.7, 0.9, 1.0}

}

max\_sample: 1

max\_trials: 50

}

...

label\_map\_file: "data/VOC0712/labelmap\_voc.prototxt"

}

// Generate samples from AnnotatedDatum using the BatchSampler.

// All sampled bboxes which satisfy the constraints defined in BatchSampler

// is stored in sampled\_bboxes.

void **GenerateBatchSamples**(const AnnotatedDatum& anno\_datum,

const vector<BatchSampler>& batch\_samplers,

vector<NormalizedBBox>\* sampled\_bboxes);

for (int i = 0; i < batch\_samplers.size(); ++i) { //对于每个batch\_sampler

for (int j = 0; j < batch\_sampler.max\_trials(); ++j) { //对于每个trial

// Generate sampled\_bbox in the normalized space [0, 1].

均匀分布[min\_scale, max\_scale]采样scale

均匀分布[min\_aspect\_ratio, max\_aspect\_ratio]采样ar

限制ar = min(max(ar, scale^2), 1/scale^2)

从而确定采样patch

大小: scale\*sqrt(ar), scale/sqrt(ar)

左上角: 均匀分布[0, 1-大小]采样

// Determine if the sampled bbox is positive or negative by the constraint.

约束条件：patch与任意真实框IoU > min\_jaccard\_overlap

且该batch\_sampler下的patch数目<=max\_sample

sampled\_bboxes.push\_back(NormalizedBBox(左上角，大小)

vector<NormalizedBBox> sampled\_bboxes;//生成个数<=batch\_samplers.size() \* max\_sample

void **CropImage**(const AnnotatedDatum& anno\_datum, const NormalizedBBox& bbox,

AnnotatedDatum\* cropped\_anno\_datum);

随机选择一个sampled\_bbox

将图像进行剪裁， 并更新标注

结果存于AnnotatedDatum sampled\_datum //注意图像和真实框都已更新

(注意：此时剪裁的是真实图像 -> distort -> expand后的图像）

对annotated data进行transform\_param指定变换

第一个参数：AnnotatedDatum containing the data and annotation

第二个参数：变换后的图像

第三个参数：变换后的标注

void **Transform**(const AnnotatedDatum& anno\_datum,

Blob<Dtype>\* transformed\_blob, vector<AnnotationGroup>\* transformed\_anno\_vec);

变换图像

cv\_resized\_image = ApplyResize(cv\_img, param\_.resize\_param());

cv\_noised\_image = ApplyNoise(cv\_resized\_image, param\_.noise\_param());

基于crop\_h, crop\_w随机剪裁cv\_noised\_image -> cv\_cropped\_image,归一化的剪裁窗口crop\_bbox

减均值，并scale -> transformed\_data即为最终的图像

变换标注

真实框bbox -> 调整到320×320坐标系NormalizedBBox resize\_bbox

UpdateBBoxByResizePolicy(param\_.resize\_param(), img\_width, img\_height, &resize\_bbox);

然后投影到最终的剪裁窗口crop\_bbox -> proj\_bbox即为最终的标注框

NormalizedBBox proj\_bbox;

ProjectBBox(crop\_bbox, resize\_bbox, &proj\_bbox)

Caffe2

Install

1. Install Dependencies

sudo apt-get update

sudo apt-get install -y --no-install-recommends \

build-essential \

cmake \

git \

libgoogle-glog-dev \

libgtest-dev \

libiomp-dev \

libleveldb-dev \

liblmdb-dev \

libopencv-dev \

libopenmpi-dev \

libsnappy-dev \

libprotobuf-dev \

openmpi-bin \

openmpi-doc \

protobuf-compiler \

python-dev \

python-pip

sudo pip install \

future \

numpy \

protobuf

# for Ubuntu 16.04

sudo apt-get install -y --no-install-recommends libgflags-dev

1. Clone & Build

git clone --recursive https://github.com/caffe2/caffe2.git && cd caffe2

# This will build Caffe2 in an isolated directory so that Caffe2 source is unaffected

mkdir build && cd build

# This configures the build and finds which libraries it will include in the

# Caffe2 installation. The output of this command is very helpful in debugging

cmake ..

# This actually builds and installs Caffe2 from makefiles generated from the

# above configuration step

sudo make install

Note:

**If download from github slowly, add IP of github into hosts (view gitcmd.txt)**

**If load github without complete, check and load submodule again**

Caffe2$git submodule update --init --recursive

(refer to: <http://blog.csdn.net/u013553529/article/details/78307072)>

$Caffe2\_ROOT$ make

注意：make -j 用多核编译，会出问题

1. Environment Variables

$sudo gedit ~/.bashrc

export Caffe2\_ROOT="/home/qzlin/Documents/dl/caffe2"

export PYTHONPATH="$Caffe2\_ROOT/build:$PYTHONPATH"

1. Test the Caffe2 Installation

$cd ~ && python -c 'from caffe2.python import core' 2>/dev/null && echo "Success" || echo "Failure"

1. Install with GPU Support

same as keras

then test GPU works

$python caffe2/python/operator\_test/relu\_op\_test.py

Note: ImportError: No module named hypothesis ##缺少hypothesis模块

$ pip install hypothesis

Tutorial

the primary idea of Caffe2 API: use Python to conveniently compose nets to train your model, pass those nets to C++ code as serialized protobuffers, and then let the C++ code run the nets with full performance.

**Blob**：存储数据

在Python中，他们被转换为numpy 矩阵

X = workspace.FetchBlob("X") #X是numpy矩阵

**Workspace**：工作空间(等效matlab, caffe中没有),

便于存储和管理net and blob

**from caffe2.python import workspace**

workspace.HasBlob("X")

workspace.FeedBlob("X", X)

workspace.FetchBlob("X")

**#存储和管理blob: 输入数据、权值、输出数据**

for name in workspace.Blobs():

print("{}:\n{}".format(name, workspace.FetchBlob(name)))

**#存储和管理net**

workspace.RunNetOnce(m.param\_init\_net)

workspace.CreateNet(m.net)

workspace.RunNet(m.name, 10) # run for 10 times

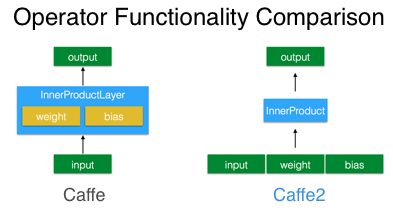
用多个名字定义多个workspace

workspace.CurrentWorkspace() **#访问当前工作空间**

workspace.SwitchWorkspace("another\_workspace", True) **#切换工作空间**

**Operator**：输入blob，输出blob，定义了计算规则

Caffe2 Operator与Caffe Layer区别？



从图上可知，Layer自包含参数，Operator仅仅是算子，参数需要输入

caffe2中的权值初始化、前传、反传、梯度更新都可以用operator实现

Operator类用来定义来数据如何计算

* 使用core.CreateOperator来直接创造

operator接受一串输入，产生一串输出

**from caffe2.python import core**

# Create an operator.

op = core.CreateOperator(

"Relu", # The type of operator that we want to run

["X"], # A list of input blobs by their names

["Y"], # A list of output blobs by their names

)

在python中创建一个operator，只是定义了一个operator，其实并没有运行这个operator。op实际上是一个protobuf对象。

在创造op之后，我们在当前的工作区中添加输入X，然后使用RunOperatorOnce运行这个operator

workspace.FeedBlob("X", np.random.randn(2, 3).astype(np.float32))

workspace.RunOperatorOnce(op)

print("Y:\n{}\n".format(workspace.FetchBlob("Y")))

operator接受无参数的输入来输出数据，从而用来生成数据，常用来初始化权值

op = core.CreateOperator(

"GaussianFill",

[], # GaussianFill does not need any parameters.

["W"],

shape=[100, 100], # shape argument as a list of ints.

mean=1.0, # mean as a single float

std=1.0, # std as a single float

)

W = net.GaussianFill([], ["W"], mean=0.0, std=1.0, shape=[5, 3], run\_once=0)

等效于

op = core.CreateOperator( "GaussianFill", [], ["W"], shape=[100, 100], mean=1.0, std=1.0)

net.Proto().op.append(op)

* 使用core.Net来访问创建operator
* 使用modelHelper来访问创建operators

**from caffe2.python import model\_helper**

model = model\_helper.ModelHelper(name="train")

# Create the input data, labels for the data as integers [0, 9]

data = np.random.rand(16, 100).astype(np.float32)

label = (np.random.rand(16) \* 10).astype(np.int32)

workspace.FeedBlob("data", data) #给计算图里提供data

workspace.FeedBlob("label", label)

方法1： 自己组合Operator: model.net.Op(...) #400+ Operator

model.param\_init\_net.XavierFill([], 'w', shape=[dim\_out, dim\_in]) # initialize your weight

model.param\_init\_net.ConstantFill([], 'b', shape=[dim\_out, ]) # initialize your bias

model.net.FC([‘data’, ‘w’, ‘b’], blob\_out) # finally building FC

方法2：Helper函数 ----封装函数创建layer： 自包含参数初始化, Op, ...

from caffe2.python.helpers import fc

fc\_1 = fc.fc(model, "data", 'fc\_1', 100, 10) #默认用XavierFill and ConstantFill初始化

方法3（推荐）：**brew 模块** (封装Helper函数)

**from caffe2.python import brew**

fc\_1 = brew.fc(m, 'data', 'fc\_1', 100, 10) #默认用XavierFill and ConstantFill初始化

Caffe2已提供的Operator见

<https://caffe2.ai/docs/operators-catalogue.html>

**Net**：由多个operator组合实现

**#定义计算图（由operator构成）**

net = core.Net("my\_first\_net") #创建网络

X = net.GaussianFill([], ["X"], mean=0.0, std=1.0, shape=[2, 3], run\_once=0)

W = net.GaussianFill([], ["W"], mean=0.0, std=1.0, shape=[5, 3], run\_once=0)

b = net.ConstantFill([], ["b"], shape=[5,], value=1.0, run\_once=0)

Y = net.FC([X, W, b], ["Y"])

#初始化net参数 (将param\_init\_net的protobuf传递给C++代码进行执行)

workspace.RunNetOnce(net)

#创建net

workspace.CreateNet(net)

#run 100\*10 iterations

for i in range(0, 100):

workspace.RunNet(net.Proto().name, 10)

为什么caffe2模型包含初始化网络和训练网络？

在caffe中，我们在训练网络中定义好了参数的初始化方式，网络加载时，程序会根据网络定义，自动初始化权值，我们只需要对这个网络，使用solver不断的前传和反传，更新参数即可。

在caffe2中，我们要把所有网络的搭建、初始化、梯度生成、梯度更新都使用operator这样一个方式来实现，所有的数据的生成、流动都要在图中反映出来

RunNetOnce: 用来运行生成权值和数据的网络，常用于初始化，这样的网络一次生成完，权值输出或数据就存在当前的workspace中，网络本身就没有存在的必要了，就直接销毁

RunNet: 可以用来重复训练网络，一开始使用CreateNet，不断迭代调用RunNet就可以不断运行网络更新参数了

封装自定义Helper function

def my\_super\_layer(model, blob\_in, blob\_out, \*\*kwargs):

...

brew.Register(my\_super\_layer)

brew.my\_super\_layer(model, blob\_in, blob\_out)

手写体分类

假定已转换成lmdb数据库

mnist-train-nchw-lmdb

data.mdb

lock.mdb

mnist-test-nchw-lmdb

data.mdb

lock.mdb

获取输入, 注意是blob

data(batch\_size, 1, 28, 28)

label(batch\_size, 1)

#注意：经过AddInput后，此时workspace含有data\_uint8, label, data, 可以运行网络获取，具体见模型剖析

def AddInput(model, batch\_size, db, db\_type):

data\_uint8, label = model.**TensorProtosDBInput**([], ["data\_uint8", "label"],

batch\_size=batch\_size, db=db, db\_type=db\_type)

data = model.Cast(data\_uint8, "data", to=core.DataType.FLOAT) # cast the data to float

data = model.**Scale**(data, data, scale=float(1./256)) # scale data from [0,255] down to [0,1]

data = model.StopGradient(data, data) # don't need the gradient for the backward pass

return data, label

#1\*28\*28 -> 20\*24\*24 -> 20\*12\*12 -> 100\*8\*8 -> 100\*4\*4 -> 500 -> 10

#注意：经过AddLeNetModel后，此时workspace含有数据blob, 还含有参数blob

def AddLeNetModel(model, data):

conv1 = brew.**conv**(model, data, 'c1', dim\_in=1, dim\_out=20, kernel=5)

pool1 = brew.**max\_pool**(model, conv1, 'p1', kernel=2, stride=2)

conv2 = brew.conv(model, pool1, 'c2', dim\_in=20, dim\_out=100, kernel=5)

pool2 = brew.max\_pool(model, conv2, 'p2', kernel=2, stride=2)

fc3 = brew.fc(model, pool2, 'fc3', dim\_in=100 \* 4 \* 4, dim\_out=500)

relu = brew.**relu**(model, fc3, fc3)

pred = brew.fc(model, relu, 'pred', 500, 10)

softmax = brew.**softmax**(model, pred, 'softmax')

return softmax

def AddTrainingOperators(model, softmax, label):

brew.**accuracy**(model, [softmax, label], "accuracy") #计算accuracy

# 计算loss

xent = model.**LabelCrossEntropy**([softmax, label], 'xent')

loss = model.**AveragedLoss**(xent, "loss")

model.**AddGradientOperators**([loss])

# SGD参数

ITER = brew.**iter**(model, "iter")

LR = model.**LearningRate**(ITER, "LR", base\_lr=-0.1, policy="step", stepsize=1, gamma=0.999 )

ONE = model.param\_init\_net.ConstantFill([], "ONE", shape=[1], value=1.0)

# for each parameter, we do the gradient updates.

for param in model.params:

# Note how we get the gradient of each parameter - ModelHelper keeps track of that.

param\_grad = model.**param\_to\_grad**[param]

# The update is a simple weighted sum: param = param + param\_grad \* LR

model.**WeightedSum**([param, ONE, param\_grad, LR], param)

def AddBookkeepingOperators(model):

model.Print('accuracy', [], to\_file=1)

model.Print('loss', [], to\_file=1)

for param in model.params:

model.Summarize(param, [], to\_file=1)

model.Summarize(model.param\_to\_grad[param], [], to\_file=1)

1. **训练**

#模型，建议输入是db输入, 输出是softmax

#注意：训练完模型之后，模型参数保存在workspace中

train\_model = ModelHelper(name="mnist\_train")

data, label = AddInput(model, batch\_size=64, db='mnist-train-nchw-lmdb', db\_type='lmdb')

softmax = AddLeNetModel(train\_model, X)

#训练方案: 计算loss, 更新每个参数梯度, 建议输入是softmax and label

**AddTrainingOperators**(train\_model, softmax, label)

# 诊断信息：训练过程中的内部信息输出

**AddBookkeepingOperators**(train\_model)

# 初始化参数

workspace.RunNetOnce(train\_model.param\_init\_net)

# 迭代训练

total\_iters = 200

accuracy, loss = np.zeros(total\_iters), np.zeros(total\_iters)

workspace.CreateNet(train\_model.net, overwrite=True)

for i in range(total\_iters):

workspace.RunNet(train\_model.net)

accuracy[i] = workspace.FetchBlob('accuracy')

loss[i] = workspace.FetchBlob('loss')

可以对训练过程的accuracy and loss可视化

1. **测试**

#测试模型, 建议输入db, 输出是accuracy,

注意模型千万不要初始化，即init\_params=False, 模型参数使用workspace里训练模型的同名参数, 特别注意：测试模型共享训练模型，通过workspace里的同名参数

test\_model = ModelHelper(name="mnist\_test", **init\_params=False)**

data, label = AddInput(test\_model, batch\_size=100, db='mnist-test-nchw-lmdb', db\_type='lmdb')

softmax = AddLeNetModel(test\_model, data)

**AddAccuracy**(test\_model, softmax, label)

workspace.RunNetOnce(test\_model.param\_init\_net) # run a test pass on the test net

workspace.CreateNet(test\_model.net, overwrite=True)

test\_accuracy = np.zeros(100)

for i in range(100):

workspace.RunNet(test\_model.net.Proto().name)

test\_accuracy[i] = workspace.FetchBlob('accuracy')

1. **deploy**

#仅需要模型部分

deploy\_model = ModelHelper(name="mnist\_deploy", **init\_params=False**)

**AddLeNetModel**(deploy\_model, "data")

# 注意输入输出需要指定

pe\_meta = **pe.PredictorExportMeta**(predict\_net=deploy\_model.net.Proto(),

parameters=[str(b) for b in deploy\_model.params],

inputs=["data"], outputs=["softmax"])

# 保存模型

**pe.save\_to\_db**("minidb", "mnist\_model.minidb", pe\_meta)

# 导入模型

predict\_net = **pe.prepare\_prediction\_net**("mnist\_model.minidb", "minidb")

# predict

workspace.FeedBlob("data", blob) #输入数据

**workspace.RunNetOnce(predict\_net)** #运行网络

softmax = workspace.FetchBlob("softmax") #获取输出结果

剖析模型

.caffemodel (Caffe) and **.pb** (Caffe2 ): these are the models

**.pbtxt**: human-readable form of the Caffe2 pb file

**deploy.prototxt**: the network architecture for deployment

**train\_val.prototxt**: the network architecture for training

**solver.prototxt**: describes the variables used during training, including learning rates, regularization, etc.

模型里输入，输出，参数等都在workspace中，通过名字获取

for name in workspace.Blobs():

blobs[name] = workspace.FetchBlob(name)

比如：

brew.conv(model, data, 'conv1', dim\_in=1, dim\_out=20, kernel=5)

输入数据是data, 输出数据是workspace.FetchBlob(‘conv1’)

权重参数是workspace.FetchBlob(‘conv1\_w’)

偏置参数是workspace.FetchBlob(‘conv1\_b’)

若是blob，无法查看shape,比如输入数据：

data\_uint8, label = model.**TensorProtosDBInput**([], ["data\_uint8", "label"],

batch\_size=batch\_size, db=db, db\_type=db\_type)

可以通过运行一次网络，从而获取信息

workspace.RunNetOnce(model.param\_init\_net)

workspace.RunNetOnce(model.net)

这时候就可以通过workspace.FetchBlob(‘data\_uint8’)查看具体信息了

查看网络图

with open("train\_net.pbtxt", 'w') as f:

f.write(str(train\_model.net.Proto()))

with open("train\_init\_net.pbtxt", 'w') as f:

f.write(str(train\_model.param\_init\_net.Proto()))

下载预训练模型

$ sudo python build/caffe2/python/models/download.py -i squeezenet

注意：-i表示保存到caffe2/build/caffe2/python/models

Caffe2模型含init\_net.pb and predict\_net.pb文件

squeezenet

init\_net.pb

predict\_net.pb

#读入protobuf文件

with open("init\_net.pb") as f:

init\_net = f.read()

with open("predict\_net.pb") as f:

predict\_net = f.read()

#从protobuf导入blobs于workspace

p = workspace.Predictor(init\_net, predict\_net)

#运行网络

results = p.run([img])

Caffe2 DB

假定数据集

(train\_features, train\_labels), (test\_features, test\_labels)

minidb

* 将numpy数据写入db中

def write\_db(**db\_type, db\_name, features, labels**):

db = core.C.create\_db(db\_type, db\_name, core.C.Mode.write)

transaction = db.new\_transaction()

for i in range(features.shape[0]):

feature\_and\_label = caffe2\_pb2.TensorProtos()

feature\_and\_label.protos.extend([

utils.NumpyArrayToCaffe2Tensor(features[i]),

utils.NumpyArrayToCaffe2Tensor(labels[i])])

transaction.put('train\_%03d'.format(i), feature\_and\_label.SerializeToString())

del transaction

del db

write\_db("minidb", "iris\_train.minidb", train\_features, train\_labels)

write\_db("minidb", "iris\_test.minidb", test\_features, test\_labels)

* 读取db数据 (通过TensorProtosDBInput加载)

def read\_db(**db\_type, db**):

net = core.Net("example\_reader")

dbreader = net.CreateDB([], "dbreader", db=db, db\_type=db\_type)

net.TensorProtosDBInput([dbreader], ["X", "Y"], batch\_size=16)

workspace.CreateNet(net)

# Let's run it to get batches of features.

for i in range(2):

workspace.RunNet(net.Proto().name)

print("The {}th batch of feature is:".format(i))

print(workspace.FetchBlob("X"))

print("The {}th batch of label is:".format(i))

print(workspace.FetchBlob("Y"))

read\_db("minidb", "iris\_train.minidb")

Multi-machine Multi-GPU 训练

基本思路：

* each GPU will execute exactly same code to run their share of the mini-batch.
* Between mini-batches, we average the gradients of each GPU and each GPU executes the parameter update in exactly the same way.
* At any point in time the parameters have same values on each GPU

Using 8 GPUS to run a batch of 32 each is equivalent to one GPU running a mini-batch of 256

for Resnet-50, we get ~7x speedup on 8 M40 GPUs over 1 GPU.

**from caffe2.python import data\_parallel\_model**, model\_helper

**import caffe2.python.predictor.predictor\_exporter as pred\_exp**

train\_model = model\_helper.ModelHelper(name="resnet50")

**data\_parallel\_model.Parallelize\_GPU**(train\_model,

input\_builder\_fun=add\_image\_input,

forward\_pass\_builder\_fun=create\_resnet50\_model\_ops,

param\_update\_builder\_fun=add\_parameter\_update\_ops,

devices=gpus, # list of integers such as [0, 1, 2, 3]

optimize\_gradient\_memory=False/True,

)

workspace.RunNetOnce(train\_model.param\_init\_net)

workspace.CreateNet(train\_model.net)

epoch = 0

while epoch < num\_epochs:

epoch\_iters = int(args.epoch\_size / total\_batch\_size / num\_shards)

for i in range(epoch\_iters):

workspace.RunNet(train\_model.net.Proto().name)

# Save the model for each epoch

predictor\_export\_meta = pred\_exp.PredictorExportMeta(

predict\_net=train\_model.net.Proto(),

parameters=data\_parallel\_model.GetCheckpointParams(train\_model),

inputs=[prefix + "/data"], outputs=[prefix + "/softmax"],

shapes={...})

pred\_exp.save\_to\_db(db\_type="minidb", db\_destination, predictor\_export\_meta)

def add\_image\_input(model):

reader = **train\_model.CreateDB**("reader", db, db\_type,num\_shards,shard\_id)

#The image input operator loads image and label data from the reader and

#applies transformations to the images (random cropping, mirroring, ...).

data, label = **brew.image\_input**(model, reader, ["data", "label"],

batch\_size=batch\_size, output\_type=dtype,

use\_gpu\_transform=True, use\_caffe\_datum=True,

mean=128., std=128., scale=256, crop=img\_size, mirror=1,is\_test=is\_test)

data = model.StopGradient(data, data)

# Model building functions

def create\_resnet50\_model\_ops(model, loss\_scale):

initializer = ...

with brew.arg\_scope([brew.conv, brew.fc], WeightInitializer=initializer, BiasInitializer=initializer):

pred = resnet.create\_resnet50(model, "data", num\_input\_channels, num\_labels)

softmax, loss = model.SoftmaxWithLoss([pred, 'label'], ['softmax', 'loss'])

loss = model.Scale(loss, scale=loss\_scale)

brew.accuracy(model, [softmax, "label"], "accuracy")

return [loss]

def add\_optimizer(model):

stepsz = int(30 \* epoch\_size / total\_batch\_size / num\_shards)

optimizer.add\_weight\_decay(model, args.weight\_decay)

opt = optimizer.build\_multi\_precision\_sgd(model, base\_learning\_rate, momentum=0.9, nesterov=1,policy="step", stepsize=stepsz, gamma=0.1)

return opt

def add\_post\_sync\_ops(model):

"""Add ops applied after initial parameter sync."""

for param\_info in model.GetOptimizationParamInfo(model.GetParams()):

if param\_info.blob\_copy is not None:

model.param\_init\_net.HalfToFloat(param\_info.blob,...)

自定义Operator

<https://caffe2.ai/docs/custom-operators.html>

## Keras

<https://keras-cn.readthedocs.io/en/latest/> Keras的核心数据结构是“模型”, Keras的底层库使用Theano或TensorFlow(“符号主义”的库)

与传统的Python代码区别？

**符号主义的计算首先定义各种变量，然后建立“计算图”，计算图规定了各个变量之间的计算关系。建立好的计算图需要编译已确定其内部细节，然而，此时的计算图还是一个“空壳子”，里面没有任何实际的数据，只有当你把需要运算的输入放进去后，才能在整个模型中形成数据流，从而形成输出值**

深度学习的优化算法，说白了就是梯度下降。每次的参数更新有两种方式：

* Batch gradient descent（批梯度下降）：遍历全部数据集算一次损失函数，然后算函数对各个参数的梯度，更新梯度。缺点：计算量开销大，计算速度慢，不支持在线学习
* stochastic gradient descent（随机梯度下降）：速度快，但收敛性不好，可能在最优点附近晃来晃去，hit不到最优点。两次参数的更新也有可能互相抵消掉，造成目标函数震荡的比较剧烈
* mini-batch gradient decent（小批的梯度下降）：数据分为若干批，按批来更新参数，批中的一组数据共同决定了本次梯度的方向，下降起来就不容易跑偏，减少了随机性

张量可以看作是向量、矩阵的自然推广，我们用张量来表示广泛的数据类型, 张量的阶数有时候也称为维度，或者轴

Ubuntu 16.04 LTS是Nvidia官方以及绝大多数深度学习框架默认开发环境

### install

安装tensorflow-gpu

# 安装CUDA开发环境: nvidia driver, cuda toolkit, cuDNN

<https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html>

<https://www.tensorflow.org/install/install_linux>

<http://www.python36.com/install-tensorflow-using-official-pip-pacakage/>

若已装有最新的版本，比如cuda9.1,需要remove，否则装不上

sudo apt-get purge nvidia\*

sudo apt-get auto-remove

重启机器

注意：cuda 9.1需要R390 nvidia driver

注意：cuda 9.0需要387 nvidia driver

$ cat /proc/driver/nvidia/version 查看nvidia版本号

sudo add-apt-repository ppa:graphics-drivers/ppa && sudo apt update

sudo apt-get install nvidia-390\*

or sudo apt-get install nvidia-387

Google: cuda tookit 9.0

CUDA Toolkit9.0

sudo dpkg -i cuda-repo-ubuntu1604-9-0-local\_9.0.176-1\_amd64.deb

sudo apt-key add /var/cuda-repo-9-0-local/7fa2af80.pub

sudo apt-get update

sudo apt-get install cuda-9.0

sudo dpkg -i libcudnn7\_7.0.5.15-1+cuda9.0\_amd64.deb

$ sudo gedit ~/.bashrc

export CUDA\_HOME=/usr/local/cuda-9.0

export PATH=$CUDA\_HOME/bin:$PATH

export LD\_LIBRARY\_PATH=$CUDA\_HOME/lib64:$LD\_LIBRARY\_PATH

$ source ~/.bashrc

Old version

http://shomy.top/2016/12/29/gpu-tensorflow-install/

$ sudo dpkg -i cuda-repo-ubuntu1604-8-0-local\_8.0.44-1\_amd64.deb

$ sudo apt-key add /var/cuda-repo-8-0-local/7fa2af80.pub

$ sudo apt update

$ sudo apt install cuda

下载并解压cudnn v7版本：cudnn-8.0-linux-x64-v7.tgz

include -> /usr/local/cuda/include

lib64 -> /usr/local/cuda/lib64

# 将CUDA路径添加至环境变量

$ sudo gedit ~/.bashrc

export CUDA\_HOME=/usr/local/cuda-8.0

export PATH=$CUDA\_HOME/bin:$PATH

export LD\_LIBRARY\_PATH=$CUDA\_HOME/lib64:$LD\_LIBRARY\_PATH

$ source ~/.bashrc

$ nvcc -V 测试nVidia cuda版本号

/usr/local/cuda/samples$sudo make all –j8 测试是否成功，运行某个sample

若出现nvidia driver 不匹配，要安装对应nvidia-driver

安装Keras

# 系统升级

$ sudo apt update

$ sudo apt upgrade

# 安装python基础开发包

$ sudo apt install -y python-dev python-pip python-nose gcc g++ git gfortran

# 安装运算加速库

$ sudo apt install -y libopenblas-dev liblapack-dev libatlas-base-dev

#安装依赖包

$ sudo pip install pip setuptools wheel

$ sudo pip install numpy scipy matplotlib scikit-learn scikit-image

$ sudo pip install h5py pyyaml

$ sudo pip install theano

$ sudo pip install tensorflow-gpu

#安装keras

$ sudo pip install keras

(note: 或者下载源代码keras$ sudo python setup.py install)

Keras默认使用TensorFlow作为后端来进行张量操作,

若有tensorflow-gpu，直接使用gpu

**Note: tensorflow-gpu对cuda版本 and cuCNN版本有要求**

若cuda-8.0 and cndnn5.1,则tensorflow-gpu-1.2

若装错了

$sudo pip uninstall tensorflow-gpu

$sudo pip install -U--pre tensorflow-gpu-1.2

$ python 验证

>>> import tensorflow

>>> import keras

Keras环境设置

修改默认keras后端: gedit ~/.keras/keras.json

若配置theano文件: gedit ~/.theanorc

[global]

openmp=False

device = gpu

floatX = float32

allow\_input\_downcast=True

[lib]

cnmem = 0.8

[blas]

ldflags= -lopenblas

[nvcc]

fastmath = True

加速测试

keras/examples/$ python mnist\_mlp.py

注意:

下载的数据集统一放于$HOME/.keras/datasets/

下载的模型统一放于$HOME/.keras/models/

tutorial

**张量Tensor**

from keras.layers import Input, Dense

from keras.models import Model

# This returns a tensor

inputs = Input(shape=(784,))

**层对象接受张量为参数，返回一个张量。**

# a layer instance is callable on a tensor, and returns a tensor

x = Dense(64, activation='relu')(inputs)

x = Dense(64, activation='relu')(x)

predictions = Dense(10, activation='softmax')(x)

**层Layer**

layer = Dense(32)

reconstructed\_layer = Dense.from\_config(config)层也可以借由配置信息重构

layer.get\_weights()：返回层的权重（numpy array）

layer.set\_weights(weights)：从numpy array中将权重加载到该层中，要求numpy array的形状与\* layer.get\_weights()的形状相同

config = layer.get\_config()：返回当前层配置信息的字典

获取层输入输出信息

非共享层

layer.input 输入张量

layer.output 输出张量

layer.input\_shape 输入数据的形状

layer.output\_shape 输出数据的形状

层有多个计算节点

layer.**get\_input\_at**(node\_index)

layer.**get\_output\_at**(node\_index)

layer.**get\_input\_shape\_at**(node\_index)

layer.**get\_output\_shape\_at**(node\_index)

**共享层**

把一个相同的Conv2D应用于一个大小为(3,32,32)的数据，然后又将其应用于一个(3,64,64)的数据，那么此时该层就具有了多个输入和输出的shape

a = Input(shape=(3, 32, 32))

b = Input(shape=(3, 64, 64))

conv = Conv2D(16, (3, 3), padding='same')

conved\_a = conv(a)

# Only one input so far, the following will work:

assert conv.input\_shape == (None, 3, 32, 32)

conved\_b = conv(b)

# now the `.input\_shape` property wouldn't work, but this does:

assert conv.get\_input\_shape\_at(0) == (None, 3, 32, 32)

assert conv.get\_input\_shape\_at(1) == (None, 3, 64, 64)

* **网络层**

**from keras.layers** import Dense, Dropout, Flatten, Reshape, Permute, RepeatVector, Lambda, ActivityRegularizer, Masking, GaussianNoise, GaussianDropout, AlphaDropout, BatchNormalization,TimeDistributed, Bidirectional

*Dense(units, activation, use\_bias, kernel\_initializer, bias\_initializer, kernel\_regularizer, bias\_regularizer, activity\_regularizer, kernel\_constraint, bias\_constraint)*

kernel\_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。

bias\_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。

kernel\_regularizer：施加在权重上的正则项，为Regularizer对象

bias\_regularizer：施加在偏置向量上的正则项，为Regularizer对象

activity\_regularizer：施加在输出上的正则项，为Regularizer对象

kernel\_constraints：施加在权重上的约束项，为Constraints对象

bias\_constraints：施加在偏置上的约束项，为Constraints对象

*Dropout(rate, noise\_shape=None, seed=None)*

Dropout将在训练过程中每次更新参数时随机断开一定百分比（rate）的输入神经元，Dropout层用于防止过拟合。

*Flatten()*

Flatten层用来将输入“压平”，即把多维的输入一维化

*Reshape(target\_shape)*

Reshape层用来将输入shape转换为特定的shape

# as first layer in a Sequential model

model = Sequential()

model.add(Reshape((3, 4), input\_shape=(12,)))

# now: model.output\_shape == (None, 3, 4)

# note: `None` is the batch dimension

# as intermediate layer in a Sequential model

model.add(Reshape((6, 2)))

# now: model.output\_shape == (None, 6, 2)

# also supports shape inference using `-1` as dimension

model.add(Reshape((-1, 2, 2)))

# now: model.output\_shape == (None, 3, 2, 2)

*Permute(dims)*

**Permute层将输入的维度按照给定模式进行重排**，例如，当需要将RNN和CNN网络连接时，可能会用到该层。

model = Sequential()

model.add(Permute((2, 1), input\_shape=(10, 64)))

# now: model.output\_shape == (None, 64, 10)

# note: `None` is the batch dimension

*RepeatVector(n)*

RepeatVector层将输入重复n次

model = Sequential()

model.add(Dense(32, input\_dim=32))

# now: model.output\_shape == (None, 32)

# note: `None` is the batch dimension

model.add(RepeatVector(3))

# now: model.output\_shape == (None, 3, 32)

*Lambda(function, output\_shape=None, mask=None, arguments=None)*

本函数用以对上一层的输出施以任何Theano/TensorFlow表达式

# add a x -> x^2 layer

model.add(Lambda(lambda x: x \*\* 2))

*ActivityRegularization(l1=0.0, l2=0.0)*

基于其激活值更新损失函数值,

l1：1范数正则因子（正浮点数）

l2：2范数正则因子（正浮点数）

*BatchNormalization*

**该层在每个batch上将前一层的激活值重新规范化，即使得其输出数据的均值接近0，其标准差接近1**

（1）加速收敛

（2）控制过拟合，可以少用或不用Dropout和正则

（3）降低网络对初始化权重不敏感

（4）允许使用较大的学习率

*GaussianNoise, GaussianDropout, AlphaDropout*

GaussianNoise(stddev)

为数据施加0均值，标准差为stddev的加性高斯噪声。该层在克服过拟合时比较有用，你可以将它看作是**随机的数据提升**。高斯噪声是需要对输入数据进行破坏时的自然选择。

一个使用噪声层的典型案例是构建去噪自动编码器，即Denoising AutoEncoder（DAE）。该编码器试图从加噪的输入中重构无噪信号，以学习到原始信号的鲁棒性表示

GaussianDropout(rate)

为层的输入施加以1为均值，标准差为sqrt(rate/(1-rate)的乘性高斯噪声

from keras.layers import *TimeDistributed, Bidirectional*

* **卷积层，反卷积，Cropping，UpSampling，ZeroPadding，Pooling**

**from keras.layers** import Conv1D, Conv2D, SeparableConv2D, Conv2DTranspose, Conv3D, Cropping1D, Cropping2D, Cropping3D, UpSampling1D, UpSampling2D, UpSampling3D, ZeroPadding1D, ZeroPadding2D, ZeroPadding3D

<http://deeplearning.net/software/theano_versions/dev/tutorial/conv_arithmetic.html#transposed-convolution-arithmetic>

一维卷积层（即时域卷积），用以在一维输入信号上进行邻域滤波

*Conv1D(...)*

二维卷积层，即对图像的空域卷积。该层对二维输入进行滑动窗卷积

*Conv2D*(filters, kernel\_size, strides=(1, 1), padding='valid', data\_format=None, dilation\_rate=(1, 1), activation=None, use\_bias=True, kernel\_initializer='glorot\_uniform', bias\_initializer='zeros', kernel\_regularizer=None, bias\_regularizer=None, activity\_regularizer=None, kernel\_constraint=None, bias\_constraint=None)

**filters：卷积核的数目**（即输出的维度）

**kernel\_size：卷积核的宽度和长度**。如为单个整数，则表示在各个空间维度的相同长度。

**strides：卷积的步长**。如为单个整数，则表示在各个空间维度的相同步长。

**padding：补0策略**，为“valid”, “same” 。“valid”代表只进行有效的卷积，“same”代表保留边界处的卷积结果

**activation：激活函数**

dilation\_rate：指定dilated convolution中的膨胀比例。

data\_format：字符串，“channels\_first”或“channels\_last”之一，代表图像的通道维的位置。该参数是Keras 1.x中的image\_dim\_ordering，**“channels\_last”对应原本的“tf”，“channels\_first”对应原本的“th”。**

use\_bias:布尔值，是否使用偏置项

kernel\_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器

bias\_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器

**kernel\_regularizer**：施加在权重上的正则项，为Regularizer对象

**bias\_regularizer**：施加在偏置向量上的正则项，为Regularizer对象

**activity\_regularizer**：施加在输出上的正则项，为Regularizer对象

kernel\_constraints：施加在权重上的约束项，为Constraints对象

bias\_constraints：施加在偏置上的约束项，为Constraints对象

输入shape

‘channels\_first’模式下，输入形如（samples, channels，rows，cols）的4D张量

**‘channels\_last’模式下，输入形如（samples，rows，cols，channels）的4D张量**

输出shape

‘channels\_first’模式下，为形如（samples，nb\_filter, new\_rows, new\_cols）的4D张量

**‘channels\_last’模式下，为形如（samples，new\_rows, new\_cols，nb\_filter）的4D张量**

输出的行列数可能会因为填充方法而改变

可分离卷积: SeparableConv2D(...)

**反卷积: Conv2DTranspose**(...)

*Conv3D*(...)

三维卷积对三维的输入进行滑动窗卷积，例如input\_shape = (3,10,128,128)代表对10帧128\*128的彩色RGB图像进行卷积

*Cropping1D*(cropping=(1, 1))

在时间轴（axis1）上对1D输入（即时间序列）进行裁剪

*Cropping2D*(cropping=((0, 0), (0, 0)), data\_format=None)

对2D输入（图像）进行裁剪，将在空域维度，即宽和高的方向上裁剪

cropping：长为2的整数tuple，分别为宽和高方向上头部与尾部需要裁剪掉的元素数

*Cropping3D*(...)

*UpSampling1D*(size=2): 在时间轴上，将每个时间步重复length次

*UpSampling2D*(size=(2, 2), data\_format=None)

**将数据的行和列分别重复size[0]和size[1]次**

*UpSampling3D*(size=(2, 2, 2), data\_format=None)

*ZeroPadding1D*(padding=1):对1D输入的首尾端（如时域序列）填充0

*ZeroPadding2D*(padding=(1, 1), data\_format=None)

对2D输入（如图片）的边界填充0，以控制卷积以后特征图的大小

**from keras.layers** import *MaxPooling1D, MaxPooling2D, MaxPooling3D, AveragePooling1D, AveragePooling2D, AveragePooling3D, GlobalMaxPooling1D, GlobalMaxPooling2D, GlobalAveragePooling2D*

* **局部连接层(locally-connected)，循环层(Recurrent),嵌入层(Embedding)**

from keras.layers import *LocallyConnected1D, LocallyConnected2D*

LocallyConnected1D层与Conv1D工作方式类似，唯一的区别是不进行权值共享。即施加在不同输入位置的滤波器是不一样的。

# apply a 3x3 unshared weights convolution with 64 output filters on a 32x32 image

# with `data\_format="channels\_last"`:

model = Sequential()

model.add(LocallyConnected2D(64, (3, 3), input\_shape=(32, 32, 3)))

# now model.output\_shape == (None, 30, 30, 64)

# notice that this layer will consume (30\*30)\*(3\*3\*3\*64) + (30\*30)\*64 parameters

from keras.layers.recurrent import *RNN, SimpleRNN, GRU, LSTM, ConvLSTM2D, SimpleRNNCell, GRUCell, LSTMCell, CuDNNGRU, CuDNNLSTM*

SimpleRNN(...): 全连接RNN网络

GRU(...): 门限循环单元

LSTM(...): 长短期记忆模型

* **嵌入层(Embedding), 融合层(Merge),激活层(Activation), 规范层(BatchNormalization), 噪声层(Noise),包装器(Wrapper)**

from keras.layers import *Embedding*

from keras.layers import *（类）Add, Substract, Multiply, Average, Maximum, Concatenate****,*** *Dot, (函数式接口）add, substract, multiply, average, maximum, concatenate, dot*

Merge层提供了一系列用于融合两个层或两个张量的层对象和方法。以大写首字母开头的是Layer类，以小写字母开头的是张量的函数。小写字母开头的张量函数在内部实际上是调用了大写字母开头的层。

Add(): 将Layer that adds a list of inputs.

Multiply(): 逐元素积的张量

Average(): 逐元素均值

Maximum(): 逐元素最大值

Concatenate(axis=-1): 按照给定轴相接构成的向量。

Dot(axes, normalize=False):两张量乘积

例如，如果两个张量a和b的shape都为（batch\_size, n），则输出为形如（batch\_size,1）的张量，结果张量每个batch的数据都是a[i,:]和b[i,:]的矩阵（向量）点积。

normalize: 布尔值，是否沿执行成绩的轴做L2规范化，如果设为True，那么乘积的输出是两个样本的余弦相似性

Add()([X1, X2]) 类使用

add([X1, X2]) 函数使用

* **激活层**

from keras.layers import *Activation*

预定义激活函数

**softmax：对输入数据的最后一维进行softmax**

elu

softplus

softsign

**relu**

tanh

**sigmoid**

hard\_sigmoid

linear

可以通过传递一个逐元素运算的Theano/TensorFlow函数来作为激活函数

from keras import backend as K

def tanh(x):

return K.tanh(x)

model.add(Dense(64, activation=tanh))

model.add(Activation(tanh)

高级激活函数

from keras.layers import *LeakyReLU, PReLU, ELU, ThresholdedReLU, Softmax*

当不激活时，LeakyReLU仍然会有非零输出值，从而获得一个小梯度，避免ReLU可能出现的神经元“死亡”现象

* **定制化层**

若无状态操作，建议用Lambda层

*Lambda(function, output\_shape=None, mask=None, arguments=None)*

本函数用以对上一层的输出施以任何Theano/TensorFlow表达式

# add a x -> x^2 layer

model.add(Lambda(lambda x: x \*\* 2))

若有状态操作（如含训练权重），则定制化层

from keras import backend as K

from keras.engine.topology import Layer

class MyLayer(Layer):

def \_\_init\_\_(self, output\_dim, \*\*kwargs):

self.output\_dim = output\_dim

super(MyLayer, self).\_\_init\_\_(\*\*kwargs)

def build(self, input\_shape): # 定制权重

self.kernel = ...

super(MyLayer, self).build(input\_shape)

def call(self, x): # 实现层的业务逻辑

return K.dot(x, self.kernel)

# 若层改变输出层shape,需要指明

def compute\_output\_shape(self, input\_shape):

return (input\_shape[0], self.output\_dim)

**模型= Sequential模型 / 函数式模型Model**

model.**summary**()：打印出模型概况

config = model.get\_config():返回包含模型配置信息的Python字典。

model = Model.from\_config(config): 模型也可以从它的config信息中重构回去

model.get\_layer()：依据层名或下标获得层对象

model.**get\_weights**()：返回模型权重张量的列表，类型为numpy array

model.**set\_weights**()：从numpy array里将权重载入给模型，要求数组具有与model.get\_weights()相同的形状。

json\_string =model.to\_json：返回代表模型的JSON字符串，仅包含网络结构，不包含权值。

model = model\_from\_json(json\_string): 可以从JSON字符串中重构原模型

model.save\_weights(filepath)：将模型权重保存到指定路径，文件类型是HDF5（后缀是.h5）

model.load\_weights(filepath, by\_name=False)：从HDF5文件中加载权重到当前模型中, 默认情况下模型的结构将保持不变。如果想将权重载入不同的模型（有些层相同）中，则设置by\_name=True，只有名字匹配的层才会载入权重

导入已训练模型

#反序列化时：自定义层名 -> 自定义层类

custom\_objects = {

'UpsampleLike' : layers.UpsampleLike,

'PriorProbability' : initializers.PriorProbability,

'RegressBoxes' : layers.RegressBoxes,

'NonMaximumSuppression' : layers.NonMaximumSuppression,

'Anchors' : layers.Anchors,

'ClipBoxes' : layers.ClipBoxes,

'\_smooth\_l1' : losses.smooth\_l1(),

'\_focal' : losses.focal(),

}

model = keras.models.load\_model('model.h5', custom\_objects)

Sequential模型

model.layers: 是添加到模型上的层的list

add(self, layer): 向模型中添加一个层, Layer对象

**compile**(self, optimizer, loss, metrics=None, sample\_weight\_mode=None)

**fit**(...)

**predict**(…)

evaluate(...)

predict\_classes(self, x, batch\_size=32, verbose=1)

predict\_proba(self, x, batch\_size=32, verbose=1)

**train\_on\_batch**(self, x, y, class\_weight=None, sample\_weight=None)

**predict\_on\_batch**(self, x)

**fit\_generator**(...)

evaluate\_generator(...)

predict\_generator(…)

函数式模型Model

广义的拥有输入和输出的模型，用Model来初始化一个函数式模型

from keras.models import Model

model = Model(inputs=[a1, a2], outputs=[b1, b3, b3])

**model.layers：组成模型图的各个层**

**model.inputs：模型的输入张量列表**

**model.outputs：模型的输出张量列表**

其余api同Sequential模型

**输入是张量，输出也是张量的一个框架就是一个模型，通过Model定义。**

# This creates a model that includes the Input layer and three Dense layers

model = Model(inputs=inputs, outputs=predictions)

model.compile(optimizer='rmsprop',loss='categorical\_crossentropy', metrics=['accuracy'])

model.fit(data, labels) # starts training

**重用已经训练好的模型**

把模型当作一个层一样，通过提供一个tensor来调用它。注意当你调用一个模型时，你不仅仅重用了它的结构，也重用了它的权重。

x = Input(shape=(784,))

# This works, and returns the 10-way softmax we defined above.

y = model(x)

**预训练权重的Keras模型**

模型的预训练权重将下载到~/.keras/models/并在载入模型时自动载入,这些模型可以用来进行预测、特征提取和finetune

ResNet50模型: from keras.applications.resnet50 import ResNet50

VGG16模型: from keras.applications.vgg16 import VGG16

model = VGG16(weights='imagenet', include\_top=False)

img = image.load\_img(img\_path, target\_size=(224, 224))

x = image.img\_to\_array(img)

x = preprocess\_input(x)

features = model.predict(x)

VGG19模型: from keras.applications.vgg19 import VGG19

InceptionV3模型: from keras.applications.inception\_v3 import InceptionV3

Xception模型: from keras.applications.xception import Xception

模型可视化 (依赖 pydot-ng 和 graphviz)

from keras.utils import vis\_utils, plot\_model

plot\_model(model, to\_file='model.png')

在ipython中展示图片

from IPython.display import SVG

from keras.utils.visualize\_util import model\_to\_dot

SVG(model\_to\_dot(model).create(prog='dot', format='svg'))

**多输入和多输出模型**



main\_input = Input(shape=(100,), dtype='int32', name='main\_input')

x = Embedding(output\_dim=512, ...)(main\_input)

lstm\_out = LSTM(32)(x)

auxiliary\_output = Dense(1, 'sigmoid', name='aux\_output')(lstm\_out)

auxiliary\_input = Input(shape=(5,), name='aux\_input')

x = keras.layers.concatenate([lstm\_out, auxiliary\_input])

for i in range(3)

x = Dense(64, activation='relu')(x)

x = Dense(64, activation='relu')(x)

x = Dense(64, activation='relu')(x)

main\_output = Dense(1, activation='sigmoid', name='main\_output')(x)

model = Model(inputs=[main\_input, auxiliary\_input],

**outputs=[main\_output, auxiliary\_output]**)

model.compile(optimizer='rmsprop',

**loss={'main\_output':** 'binary\_crossentropy',

**'aux\_output':** 'binary\_crossentropy'},

loss\_weights={'main\_output': 1.,

'aux\_output': 0.2})

model.fit({'main\_input': headline\_data, 'aux\_input': additional\_data},

**{'main\_output': labels, 'aux\_output': labels}**,

epochs=50, batch\_size=32)

#### 数据预处理

**from keras.preprocessing.image import ImageDataGenerator**

ImageDataGenerator(featurewise\_center=False,

samplewise\_center=False,

featurewise\_std\_normalization=False,

samplewise\_std\_normalization=False,

zca\_whitening=False,

rotation\_range=0.,

width\_shift\_range=0.,

height\_shift\_range=0.,

shear\_range=0.,

zoom\_range=0.,

channel\_shift\_range=0.,

fill\_mode='nearest',

cval=0.,

horizontal\_flip=False,

vertical\_flip=False,

rescale=None,

**preprocessing\_function**=None,

data\_format=K.image\_data\_format())

用以生成一个batch的图像数据，支持实时数据提升。训练时该函数会无限生成数据，直到达到规定的epoch次数为止。

featurewise\_center：布尔值，**使输入数据集去中心化**（均值为0）, 按feature执行

featurewise\_std\_normalization：布尔值，将输入除以数据集的标准差以完成标准化, 按feature执行

rotation\_range：整数，数据提升时图片随机转动的角度

width\_shift\_range：浮点数，图片宽度的某个比例，数据提升时图片水平偏移的幅度

height\_shift\_range：浮点数，图片高度的某个比例，数据提升时图片竖直偏移的幅度

shear\_range：浮点数，剪切强度（逆时针方向的剪切变换角度）

zoom\_range：浮点数或形如[lower,upper]的列表，随机缩放的幅度，若为浮点数，则相当于[lower,upper] = [1 - zoom\_range, 1+zoom\_range]

fit(x, augment=False, rounds=1)：计算依赖于数据的变换所需要的统计信息(均值方差等),只有使用featurewise\_center，featurewise\_std\_normalization或zca\_whitening时需要此函数。

flow(self, X, y, batch\_size=32, shuffle=True, seed=None, ...)：接收numpy数组和标签为参数,生成经过数据提升或标准化后的batch数据,并在一个无限循环中不断的返回batch数据

(x\_train, y\_train), (x\_test, y\_test) = cifar10.load\_data()

y\_train = np\_utils.to\_categorical(y\_train, num\_classes)

y\_test = np\_utils.to\_categorical(y\_test, num\_classes)

**datagen = ImageDataGenerator**(

featurewise\_center=True,

featurewise\_std\_normalization=True,

rotation\_range=20,

width\_shift\_range=0.2,

height\_shift\_range=0.2,

horizontal\_flip=True)

# compute quantities required for featurewise normalization

# (std, mean, and principal components if ZCA whitening is applied)

datagen.fit(x\_train)

# fits the model on batches with real-time data augmentation:

**model.fit\_generator(datagen.flow**(x\_train, y\_train, batch\_size=32),

steps\_per\_epoch=len(x\_train), epochs=epochs)

同时变换图像和mask

# we create two instances with the same arguments

data\_gen\_args = dict(featurewise\_center=True,

featurewise\_std\_normalization=True,

rotation\_range=90.,

width\_shift\_range=0.1,

height\_shift\_range=0.1,

zoom\_range=0.2)

image\_datagen = ImageDataGenerator(\*\*data\_gen\_args)

mask\_datagen = ImageDataGenerator(\*\*data\_gen\_args)

# Provide the same seed and keyword arguments to the fit and flow methods

seed = 1

image\_datagen.fit(images, augment=True, seed=seed)

mask\_datagen.fit(masks, augment=True, seed=seed)

image\_generator = image\_datagen.flow\_from\_directory(

'data/images',

class\_mode=None,

seed=seed)

mask\_generator = mask\_datagen.flow\_from\_directory(

'data/masks',

class\_mode=None,

seed=seed)

# combine generators into one which yields image and masks

train\_generator = zip(image\_generator, mask\_generator)

model.fit\_generator(train\_generator, steps\_per\_epoch=2000, epochs=50)

#### 性能评估

**from keras import losses**

真实的优化目标函数是在各个数据点得到的损失函数值之和的均值

可用的目标函数:

mean\_squared\_error或mse

mean\_absolute\_error或mae

mean\_absolute\_percentage\_error或mape

mean\_squared\_logarithmic\_error或msle

squared\_hinge

hinge

**binary\_crossentropy**（亦称作对数损失，logloss）

logcosh

**categorical\_crossentropy：亦称作多类的对数损失，注意使用该目标函数时，需要将标签转化为形如(nb\_samples, nb\_classes)的二值序列**

sparse\_categorical\_crossentrop：如上，但接受稀疏标签

kullback\_leibler\_divergence:从预测值概率分布Q到真值概率分布P的信息增益,用以度量两个分布的差异.

poisson：即(predictions - targets \* log(predictions))的均值

cosine\_proximity：即预测值与真实标签的余弦距离平均值的相反数

**from keras import optimizers**

optimizers.**SGD**(lr=0.01, momentum=0.0, decay=0.0, nesterov=False)

随机梯度下降法，支持动量参数，支持学习衰减率，支持Nesterov动量

optimizers.**RMSprop**(lr=0.001, rho=0.9, epsilon=1e-06)

除学习率可调整外，建议保持优化器的其他默认参数不变,递归神经网络时的一个良好选择

optimizers.Adagrad(lr=0.01, epsilon=1e-06): 建议保持优化器的默认参数不变

Adadelta(lr=1.0, rho=0.95, epsilon=1e-06): 建议保持优化器的默认参数不变

optimizers.**Adam**(lr=0.001, beta\_1=0.9, beta\_2=0.999, epsilon=1e-08):该优化器的默认值来源于参考文献

Nadam(lr=0.002, beta\_1=0.9, beta\_2=0.999, epsilon=1e-08, schedule\_decay=0.004)

**Adam本质上像是带有动量项的RMSprop，Nadam就是带有Nesterov 动量的Adam RMSprop**

**from keras import metrics**

metrics=[metrics.mae, metrics.categorical\_accuracy]

categorical\_accuracy:对多分类问题,计算所有预测值上的平均正确率

#### 权重初始化，正则项，约束项

**from keras import initializers**

不同的层可能使用不同的关键字来传递初始化方法

Dense(64, kernel\_initializer='random\_uniform', bias\_initializer='zeros')

kernel\_initializer=initializers.random\_normal(stddev=0.01)

kernel\_initializer='random\_normal'

预定义初始化方法

initializers.Zeros()

initializers.Ones()

Constant(value=0)

RandomNormal(mean=0.0, stddev=0.05, seed=None))

RandomUniform(minval=-0.05, maxval=0.05, seed=None)

TruncatedNormal(mean=0.0, stddev=0.05, seed=None):截尾高斯分布初始化，该初始化方法与RandomNormal类似，但位于均值两个标准差以外的数据将会被丢弃并重新生成，形成截尾分布。该分布是神经网络权重和滤波器的推荐初始化方法。

VarianceScaling(scale=1.0, mode='fan\_in', distribution='normal', seed=None):自适应目标张量的shape。

Orthogonal(gain=1.0, seed=None):用随机正交矩阵初始化

Identity(gain=1.0)

lecun\_uniform(seed=None): LeCun均匀分布初始化方法

glorot\_normal(seed=None):Glorot正态分布初始化方法，也称作Xavier正态分布初始化

he\_normal(seed=None):He正态分布初始化方法，也称作Xavier正态分布初始化

he\_uniform(seed=None)

自定义初始化器

from keras import backend as K

def my\_init(shape, dtype=None):

return K.random\_normal(shape, dtype=dtype)

model.add(Dense(64, init=my\_init))

正则项，约束项

**正则项在优化过程中层的参数或层的激活值添加惩罚项，这些惩罚项将与损失函数一起作为网络的最终优化目标**

惩罚项基于层进行惩罚，目前惩罚项的接口与层有关，但Dense, Conv1D, Conv2D, Conv3D具有共同的接口。

kernel\_regularizer：施加在权重上的正则项，为keras.regularizer.Regularizer对象

bias\_regularizer：施加在偏置向量上的正则项，为keras.regularizer.Regularizer对象

activity\_regularizer：施加在输出上的正则项，为keras.regularizer.Regularizer对象

**from keras import regularizers**

model.add(Dense(64, input\_dim=64, kernel\_regularizer=regularizers.l2(0.01),

activity\_regularizer=regularizers.l1(0.01)))

开发新的正则项

任何以权重矩阵作为输入并返回单个数值的函数均可以作为正则项

from keras import backend as K

def l1\_reg(weight\_matrix):

return 0.01 \* K.sum(K.abs(weight\_matrix))

Dense(64, input\_dim=64, kernel\_regularizer=l1\_reg)

约束项

在优化过程中为网络的参数施加约束

惩罚项基于层进行惩罚，目前惩罚项的接口与层有关，但Dense, Conv1D, Conv2D, Conv3D具有共同的接口。

kernel\_constraint：对主权重矩阵进行约束

bias\_constraint：对偏置向量进行约束

from keras.constraints import maxnorm

Dense(64, kernel\_constraint=max\_norm(2.))

预定义约束项

max\_norm(m=2)：最大模约束

non\_neg()：非负性约束

unit\_norm()：单位范数约束, 强制矩阵沿最后一个轴拥有单位范数

#### 训练过程的网络状态--通过回调函数Callbacks

**回调函数是一组在训练的特定阶段被调用的函数集**，你可以使用回调函数来观察训练过程中网络内部的状态和统计信息。

回调函数以字典logs为参数，该字典包含了一系列与当前batch或epoch相关的信息。

在每个epoch的结尾处（on\_epoch\_end），logs将包含训练的正确率和误差，acc和loss，如果指定了验证集，还会包含验证集正确率和误差val\_acc和val\_loss，val\_acc还额外需要在.compile中启用metrics=['accuracy']。

在每个batch的开始处（on\_batch\_begin）：logs包含size，即当前batch的样本数

在每个batch的结尾处（on\_batch\_end）：logs包含loss，若启用accuracy则还包含acc

**from keras import callbacks**

callbacks.BaseLogger(): 对每个epoch累加metrics指定的监视指标的epoch平均值, 在Keras模型中会被自动调用

callbacks.History(): 在Keras模型上会被自动调用，History对象即为fit方法的返回值

callbacks.ProgbarLogger(): 将metrics指定的监视指标输出到标准输出上

**callbacks.ModelCheckpoint(filepath ...): 在每个epoch后保存模型到filepath**

**callbacks.EarlyStopping(...): 当监测值不再改善时，该回调函数将中止训练**

callbacks.RemoteMonitor(root='http://localhost:9000'):向服务器发送事件流，该回调函数需要requests库

callbacks.TensorBoard(log\_dir='./logs', ...): 可视化的展示器

**callbacks.ReduceLROnPlateau(monitor='val\_loss', factor=0.1,...):当评价指标不在提升时，减少学习率**

callbacks.CSVLogger(filename, separator=',', append=False):将epoch的训练结果保存在csv文件中

编写自己的回调函数

class LossHistory(keras.callbacks.Callback):

def on\_train\_begin(self, logs={}):

self.losses = []

def on\_batch\_end(self, batch, logs={}):

self.losses.append(logs.get('loss'))

history = LossHistory()

model.fit(X\_train, Y\_train, batch\_size=128, epochs=20, verbose=0, callbacks=[history])

print history.losses

恢复最新的模型进行训练

latest\_model, max\_iter = find\_latest\_model(snapshot\_path, backbone, dataset\_type)

if latest\_model is not None:

model = keras.models.load\_model(latest\_model, custom\_objects=custom\_objects)

model.fit\_generator(

generator=train\_generator,

steps\_per\_epoch=args.steps,

epochs=args.epochs,

verbose=1,

callbacks=callbacks,

initial\_epoch=max\_iter #epoch接着上次停下来的

)

# Find most recent snapshot.

def find\_latest\_model(snapshot\_dir, backbone, dataset\_type='csv'):

max\_iter = 0

for file in os.listdir(snapshot\_dir):

if file.endswith(".h5"):

basename = os.path.splitext(file)[0]

iter = int(basename.split("{}\_{}\_".format(backbone, dataset\_type))[1])

if iter > max\_iter:

max\_iter = iter

lastest\_model\_name = "{}\_{}\_{}.h5".format(backbone, dataset\_type, max\_iter)

latest\_model = os.path.join(snapshot\_dir, lastest\_model\_name) if max\_iter>0 else None

return latest\_model, max\_iter

keras.utils提供的使得方法

x\_data = keras.utils.io\_utils.HDF5Matrix('input/file.hdf5', 'data')

model.predict(x\_data)

keras.utils.to\_categorical(y, num\_classes=None)

keras.utils.normalize(x, axis=-1, order=2)

x：待规范化的数据

axis: 规范化的轴

order：规范化方法，如2为L2范数

**Key points**

#开发阶段，如何避免性能改变是由于随机改变，还是模型提升？

keras.io/getting-started/faq/

import os

os.environ['PYTHONHASHSEED'] = '0'

np.random.seed(42)

rn.seed(12345)

**模型 f(input\_shape, n\_class)**

#Model 需要指定输入维数（不需要包含batch\_size）, 由input tensor and output #tensor定义。可以将model视为layers，输入tensor, 输出tensor

input\_shape = (w, h, channel)

model = Model(Input(input\_shape), output)

Y\_predict = model(X\_train)

冻结层

def freeze(model):

#Set all layers in a model to non-trainable.

for layer in model.layers:

layer.trainable = False

return model

#注意，同样地，所有层，不考虑batch\_size，比如：

h1 = Dense(32, input\_shape=(16, )) 输入(\*, 16), 输出(\*, 32)

c1 = Conv2D(64, 3, 3)(pool1) 若pool1(\*, 32, 32, 3), 输出(\*, 32, 32, 64)

f1 = Flatten()(c1) 输出(\*, 32\*32\*64)

r1 = Reshape((32, 32, 64))(f1) 输出(\*, 32, 32, 64)

Permute((1, 3))(r1) 输出(\*, 64, 32, 32)

RepeatVector(3)(h1) 输出(\*, 3, 32)

**保存/导入模型**

#saving/loading whole models (architecture + weights + optimizer state)

from keras.models import load\_model

model.save('my\_model.h5') # creates a HDF5 file 'my\_model.h5'

del model # deletes the existing model

# returns a compiled model, identical to the previous one

model = load\_model('my\_model.h5')

**训练**

若指标没有随训练下降，停止学习

from keras.callbacks import EarlyStopping

early\_stopping = EarlyStopping(monitor='val\_loss', patience=2)

model.fit(x, y, validation\_split=0.2, callbacks=[early\_stopping])

**正则化**

mnist 60000训练集，加了reg, acc从99.11 → 99.12

Conv2D(32, (5, 5), activation='relu',

kernel\_regularizer=l2(weight\_decay),

bias\_regularizer=l2(weight\_decay),

activity\_regularizer=l2(weight\_decay))

**不平衡类**

mnist 60000训练集，加了class\_weight, acc从99.12 → 99.19

class\_weight = {0 : 1., 1: 50., 2: 2.} 对应类权重

model.fit(X\_train, Y\_train， class\_weight = class\_weight)

**计算类权重，越少类权重越大**

from sklearn.utils.class\_weight import compute\_class\_weight

class\_weight = compute\_class\_weight('balanced', np.unique(y\_train), y\_train)

model.fit(X\_train, y\_train, class\_weight=class\_weight)

熵作为类权重

log(1 / yi) 其中yi ＝ count / (mu\*total), mu = 0.15

# labels\_dict : {ind\_label: count\_label}

# mu : parameter to tune

def create\_class\_weight(labels\_dict, mu=0.15):

total = np.sum(labels\_dict.values())

class\_weight = {}

for key in labels\_dict.keys():

score = math.log(mu\*total/float(labels\_dict[key]))

class\_weight[key] = score if score > 1.0 else 1.0

return class\_weight

**损失函数 f(target, output)**

#input have shape [batch\_size, w, h, 1]

#output have shape [batch\_size, 1]

target = K.reshape(target, [-1, w\*h\*1])

output = K.reshape(output, [-1, w\*h\*1])

dice\_coef

intersection = K.sum(y\_true \* y\_pred, axis=-1)

return (2. \* intersection + smooth) /

(K.sum(y\_true, axis=-1) + K.sum(y\_pred, axis=-1) + smooth)

binary\_cross\_entropy

pos\_weight = 2

loss = pos\_weight \* y\_true \* K.log(y\_pred+K.epsilon()) + (1-y\_true)\*K.log(1-y\_pred+K.epsilon())

return -K.mean(loss, axis=-1)

# keras的损失函数binary\_crossentropy支持[batch\_size, w, h, 1], 但最后一层需为sigmoid激活函数

model.compile(loss=binary\_crossentropy, metrics=[dice\_coef])

但是class\_weight不支持3D以上, 自定义可以实现此功能

输出为 [batch\_size, 1], 每个元素表示case的loss, keras会求平均

若输出为标量，则表示batch\_size case的平均loss

dice\_coef

intersection = K.sum(y\_true \* y\_pred)

return 2. \* intersection / (K.sum(y\_true) + K.sum(y\_pred) )

categorical\_crossentropy

若logits and y have same shape (32, 512, 512, 10)

则class\_weights have shape (1, 10)

flat\_logits = tf.reshape(logits, [-1, n\_class]) (32\*512\*512, 10)

flat\_labels = tf.reshape(y, [-1, n\_class])

class\_weights = tf.constant(np.array(class\_weights, dtype=np.float32))

weight\_map = tf.multiply(flat\_labels, class\_weights) (32\*512\*512, 10)

weight\_map = tf.reduce\_sum(weight\_map, axis=1) (32\*512\*512, 1)

(32\*512\*512, 1) ← (32, 512, 512, 10) , (32, 512, 512, 10)

loss\_map = tf.nn.softmax\_cross\_entropy\_with\_logits(flat\_logits, flat\_labels)

weighted\_loss = tf.multiply(loss\_map, weight\_map) (32\*512\*512, 1)

loss = tf.reduce\_mean(weighted\_loss) scalar

**Metrics f(y\_true, y\_pred)**

#(y\_true, y\_pred) as arguments and return a single tensor value.

import keras.backend as K

def mean\_pred(y\_true, y\_pred):

return K.mean(y\_pred)

model.compile(optimizer='rmsprop',

loss='binary\_crossentropy',

metrics=['accuracy', mean\_pred])

**多输入和多输出模型**



main\_input = Input(shape=(100,), dtype='int32', name='main\_input')

x = Embedding(output\_dim=512, ...)(main\_input)

lstm\_out = LSTM(32)(x)

auxiliary\_output = Dense(1, 'sigmoid', name='aux\_output')(lstm\_out)

auxiliary\_input = Input(shape=(5,), name='aux\_input')

x = keras.layers.concatenate([lstm\_out, auxiliary\_input])

for i in range(3)

x = Dense(64, activation='relu')(x)

x = Dense(64, activation='relu')(x)

x = Dense(64, activation='relu')(x)

main\_output = Dense(1, activation='sigmoid', name='main\_output')(x)

model = Model(inputs=[main\_input, auxiliary\_input],

**outputs=[main\_output, auxiliary\_output]**)

model.compile(optimizer='rmsprop',

**loss={'main\_output':** 'binary\_crossentropy',

**'aux\_output':** 'binary\_crossentropy'},

loss\_weights={'main\_output': 1.,

'aux\_output': 0.2})

model.fit({'main\_input': headline\_data, 'aux\_input': additional\_data},

**{'main\_output': labels, 'aux\_output': labels}**,

epochs=50, batch\_size=32)

### Keras后端

Keras并不处理如张量乘法、卷积等底层操作。这些操作依赖于某种特定的、优化良好的张量操作库。Keras依赖于处理张量的库就称为“后端引擎”。默认后端引擎是tensorflow

**使用抽象的Keras后端来编写代码**

from keras import backend as K

#tensor, 等价于tf.placeholder() ,T.matrix(),T.tensor3()

input = K.placeholder(shape=(2, 4, 5))

#获取tensor信息

print(K.dtype(input))

print(K.ndim(input))

print(K.int\_shape(input))

K.is\_keras\_tensor(input) == False or True

#实例化tensor，等价于tf.variable()或 theano.shared()

val = np.random.random((3, 4, 5))

var = K.variable(value=val)

var = K.zeros(shape=(3, 4, 5)) # all-zeros variable

var = K.ones(shape=(3, 4, 5)) # all-ones

K.eye(size, dtype='float32', name=None)

#获取tensor信息

print(K.dtype(var))

print(K.ndim(var))

print(K.int\_shape(var))

print(K.eval(var))

K.set\_value(var, np.zeros(dim)) #从numpy array将值载入张量中

K.get\_value(var) #以Numpy array的形式返回张量的值

var1 = K.zeros(dim)

var2 = K.ones(dim)

vars = [var1, var2]

result = K.batch\_get\_value(vars) #Numpy array list的形式返回多个张量的值

K.batch\_set\_value(tuples)

**大多数你需要的张量操作都可以通过统一的Keras后端接口完成**

a = b + c \* K.abs(d)

c = K.dot(a, K.transpose(b))

a = K.sum(b, axis=2)

a = K.softmax(b)

eps = K.epsilon()

K.set\_epsilon(1e-05)

K.floatx() #返回默认的浮点数数据类型，为字符串，如 'float16', 'float32', 'float64'

K.set\_floatx('float16')

K.cast(input, dtype='float16') #修改张量类型

data\_format = K.image\_data\_format() #‘channels\_last’或‘channels\_first’

K.set\_image\_data\_format(data\_format)

zeros(shape, dtype='float32', name=None)

ones(shape, dtype='float32', name=None)

eye(size, dtype='float32', name=None)

zeros\_like(x, name=None) #生成与另一个张量x的shape相同的全0张量

ones\_like(x, name=None)

arange(start, stop=None, step=1, dtype='int32') #生成1D的整数序列张量

#初始化一个Keras变量，其数值为从一个均匀分布中采样的样本

random\_uniform\_variable(shape, low, high, dtype=None, name=None, seed=None)

#返回具有正态分布值的张量，mean和stddev为均值和标准差

random\_normal(shape, mean=0.0, stddev=1.0, dtype=None, seed=None)

#返回具有二项分布值的张量，p是二项分布参数

random\_binomial(shape, p=0.0, dtype=None, seed=None)

#返回具有截尾正态分布值的张量，在距离均值两个标准差之外的数据将会被截断并重新生成

truncated\_normal(shape, mean=0.0, stddev=1.0, dtype=None, seed=None)

round, sign, pow, sin, cos, log, square, abs, sqrt, exp,

logsumexp(x, axis=None, keepdims=False) #在给定轴上计算log(sum(exp()))

clip(x, min\_value, max\_value) 逐元素clip（将超出指定范围的数强制变为边界值）

sum(x, axis=None, keepdims=False) #在给定轴上计算张量中元素之和

var(x, axis=None, keepdims=False) #在给定轴上计算张量方差

std(x, axis=None, keepdims=False) #在给定轴上求张量元素之标准差

mean(x, axis=None, keepdims=False) #在给定轴上求张量元素之均值

cumprod(x, axis=0) #在给定轴上求张量的累积和

prod(x, axis=None, keepdims=False) #在给定轴上计算张量中元素之积

min(x, axis=None, keepdims=False) #求张量中的最小值

max(x, axis=None, keepdims=False) #求张量中的最大值

argmax(x, axis=-1) #在给定轴上求张量之最大元素下标

argmin(x, axis=-1) #在给定轴上求张量之最小元素下标

maximum(x, y) #逐元素取两个张量的最大值

minimum(x, y) #逐元素取两个张量的最小值

gather(reference, indices) #在给定的张量中检索给定下标的向量

dot(x, y) #两个张量的乘积

batch\_dot(x, y, axes=None) #按批进行张量乘法

transpose(x)

resize\_images(X, height\_factor, width\_factor, dim\_ordering)

#(batch, depth, height, width, channels)

resize\_volumes(X, depth\_factor, height\_factor, width\_factor, dim\_ordering)

concatenate(tensors, axis=-1) #在给定轴上将一个列表中的张量串联为一个张量 specified axis

reshape(x, shape) #将张量的shape变换为指定shape

permute\_dimensions(x, pattern) #按照给定的模式重排一个张量的轴

repeat\_elements(x, rep, axis) #在给定轴上重复张量元素rep次

若xshape(s1, s2, s3)并且给定轴为axis=1`，输出张量的shape为`(s1, s2 \* rep, s3)

repeat(x, n) #重复2D张量

若xshape是(samples, dim)且n为2，则输出张量的shape是(samples, 2, dim)

tile(x, n) #将x在各个维度上重复n次

batch\_flatten(x) #将一个n阶张量转变为2阶张量，其第一维度保留不变

expand\_dims(x, dim=-1) #在下标为dim的轴上增加一维

squeeze(x, axis) #将下标为axis的一维从张量中移除

temporal\_padding(x, padding=1) #向3D张量中间的那个维度的左右两端填充padding个0值

asymmetric\_temporal\_padding(x, left\_pad=1, right\_pad=1)

spatial\_2d\_padding(x, padding=(1, 1), dim\_ordering='th')

spatial\_3d\_padding(x, padding=(1, 1, 1), dim\_ordering='th')

reverse(x, axes) #将一个张量在给定轴上反转

stack(x, axis=0) #将一个列表中维度数目为R的张量堆积起来形成维度为R+1的新张量

one\_hot(indices, nb\_classes) #输出为(n+1)维的one-hot编码

normalize\_batch\_in\_training() #对batch数据计算均值和方差，然后再进行batch\_normalization

batch\_normalization() #output = (x-mean)/(sqrt(var)+epsilon)\*gamma+beta

gradients(loss, variables) #返回loss函数关于variables的梯度，variables为张量变量的列表

stop\_gradient(variables)

in\_train\_phase(x, alt) #如果处于训练模式，则选择x，否则选择alt

in\_test\_phase(x, alt) #如果处于测试模式，则选择x，否则选择alt

relu(x, alpha=0.0, max\_value=None)

elu(x, alpha=1.0)

softmax(x)

softplus(x)

softsign(x)

sigmoid(x)

hard\_sigmoid(x)

tanh(x)

categorical\_crossentropy(output, target, from\_logits=False) #类别交叉熵

sparse\_categorical\_crossentropy(output, target, from\_logits=False)#类别交叉熵目标张量必须是整型张量

binary\_crossentropy(output, target, from\_logits=False)

l2\_normalize(x, axis) #在给定轴上对张量进行L2范数规范化

dropout(x, level, seed=None) #随机将x中一定比例的值设置为0，并放缩整个tensor

in\_top\_k(predictions, targets, k) #判断目标是否在predictions的前k大值位置

conv1d(...)

conv2d(...)

deconv2d(...) #2D反卷积（转置卷积）

conv3d(...)

pool2d(...)

pool3d(...)

bias\_add(x, bias, data\_format=None)

equal(x, y) #逐元素判相等关系，返回布尔张量

greater(x,y) #逐元素判断x>y关系，返回布尔张量

greater\_equal(x,y) #逐元素判断x>=y关系，返回布尔张量

lesser(x,y) #逐元素判断x<y关系，返回布尔张量

lesser\_equal(x,y) #逐元素判断x<=y关系，返回布尔张量

switch(condition, then\_expression, else\_expression) #依据给定的条件‘condition’（整数或布尔值）在两个表达式之间切换，注意两个表达式都应该是具有同样shape的符号化张量表达式

map\_fn(fn, elems, name=None) #元素elems在函数fn上的映射，并返回结果

foldl(fn, elems, initializer=None, name=None) #用fn从左到右连接它们

foldr(fn, elems, initializer=None, name=None) #用fn从右到左连接它们

**自定义层**

方法一：若无状态操作，采用layers.core.Lambda

*Lambda(function, output\_shape=None, mask=None, arguments=None)*

本函数用以对上一层的输出施以任何Theano/TensorFlow表达式

若输出维数不变：model.add(Lambda x: x\*\*2) # add a x -> x^2 layer

若输出维数改变：

def antirectifier(x):

pos = K.relu(x)

neg = K.relu(-x)

return K.concatenate([pos, neg], axis=1)

def output\_shape(input\_shape):

shape = list(input\_shape)

shape[-1] \*= 2

return tuple(shape)

model.add(Lambda(antirectifier, output\_shape=output\_shape))

方法二：若有状态操作，如含权重，需要继承Layer

**from keras import backend as K**

**from keras.engine.topology import Layer**

import numpy as np

#阅读源代码

class Conv2D(**Layer**):

def \_\_init\_\_(self, kernel\_size=(3,3), filters=32, \*\*kwargs):

self.kernel\_size = kernel\_size

self.filters = filters

super(MyLayer, self).\_\_init\_\_(\*\*kwargs)

#**定义权重**的方法 # Create a trainable weight variable for this layer.

**def build(self, input\_shape):**

channels\_axis = -1 if self.data\_format==’channels\_last’ else 1

input\_dim = input\_shape[channels\_axis]

kernel\_shape = self.kernel\_size + (input\_dim, self.filters)

self.kernel = self.add\_weight(shape=kernel\_shape, initializer='uniform')

super(MyLayer, self).build(input\_shape)

# 实现层的业务逻辑

**def call(self, inputs):**

return K.con2d(inputs, self.kernel)

# 若层改变输出层shape,需要指明

**def compute\_output\_shape(self, input\_shape):**

return (input\_shape[0], input\_shape[1], self.filters )

Keras visualization Toolkit

## tensorflow

TensorFlow + TensorBoard + TensorFlow Serving

TensorFlow =

TensorFlow Core (low level API) +

tf.contrib.learn (high-level API)

设计目标：在**多台计算机**以及单机多CPU,单机多GPU环境中具有良好的可伸缩性

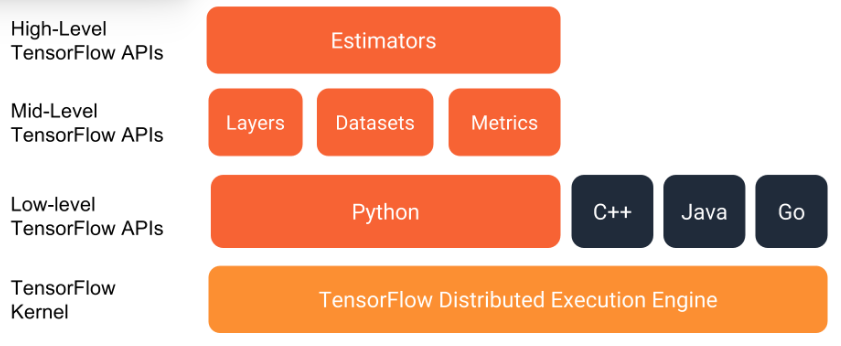
Install

cuda9 + cuDNN v7 (见keras install)

$sudo pip uninstall tensorflow-gpu

$sudo pip install tensorflow-gpu

Tensorflow programming stack



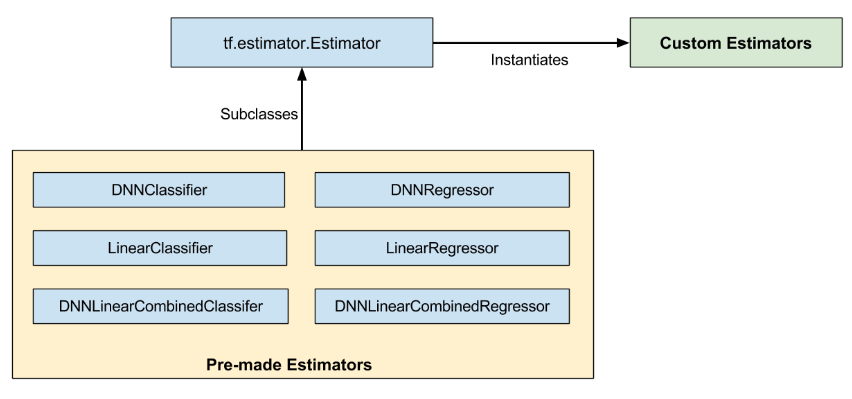
High Level APIs

tf.contrib.slim

tf.estimator.Estimator

tf.Keras

使用Estimators(模型)



特点：

* built on tf.layers
* build the graph and manage session
* initialize variables
* start queues
* handle exceptions
* create checkpoint files and recover from failures
* save summaries for TensorBoard

代码结构：

1. **数据导入函数**

features = {'SepalLength': np.array([6.4, 5.0, ...]),

'SepalWidth': np.array([2.8, 2.3, ...]),

'PetalLength': np.array([5.6, 3.3, ...]),

'PetalWidth': np.array([2.2, 1.0, ...])}

labels = np.array([N, 1])

# Convert the inputs to a Dataset.

def input\_fn(features, labels, batch\_size):

features=dict(features)

inputs = features if labels is None else (features, labels)

dataset = **tf.data.Dataset.from\_tensor\_slices**((inputs)

dataset = dataset.shuffle(1000).repeat().batch(batch\_size)

return dataset

1. **define Feature columns which describe how to use the input.**

my\_feature\_columns = []

for key in train\_x.keys():

my\_feature\_columns.append(tf.feature\_column.numeric\_column(key=key))

1. **实例化estimator**

# Build DNN with two hidden layer of 10 nodes each, output 3 nodes

# The model must choose between 3 classes.

classifier = **tf.estimator.DNNClassifier**(

feature\_columns=my\_feature\_columns,

hidden\_units=[10, 10],

n\_classes=3)

1. **调用train, evaluate, predict**

# Train the Model with 200 steps of batch

**classifier.train**(input\_fn=lambda: input\_fn(features, labels, 32), steps=200)

# Evaluate the model.获取eval\_result['accuracy']

eval\_result = **classifier.evaluate**(input\_fn=lambda: input\_fn(test\_x, test\_y, 32))

# model predict

predict\_x = {

'SepalLength': [5.1, 5.9, 6.9],

'SepalWidth': [3.3, 3.0, 3.1],

'PetalLength': [1.7, 4.2, 5.4],

'PetalWidth': [0.5, 1.5, 2.1],

}

expected = ['Setosa', 'Versicolor', 'Virginica']

predictions = **classifier.predict**(input\_fn=lambda: input\_fn(predict\_x, labels=None,32))

for pred\_dict in predictions: #3个样本

class\_id = pred\_dict['class\_ids'][0] 预测类别

probability = pred\_dict['probabilities'][class\_id] 该类别概率

Estimators <- Keras models

keras\_inception\_v3 = **tf.keras.applications.inception\_v3.InceptionV3**(weights=None)

keras\_inception\_v3.compile(optimizer=tf.keras.optimizers.SGD(lr=0.0001, momentum=0.9),

loss='categorical\_crossentropy',

metric='accuracy')

est\_inception\_v3 = **tf.keras.estimator.model\_to\_estimator**(keras\_model=keras\_inception\_v3)

keras\_inception\_v3.input\_names # print out: ['input\_1']

train\_input\_fn = **tf.estimator.inputs.numpy\_input\_fn**(

x={"input\_1": train\_data},

y=train\_labels,

num\_epochs=1,

shuffle=False)

est\_inception\_v3.train(input\_fn=train\_input\_fn, steps=2000)

自定义Estimators

def my\_model(features, labels, mode, params):

#创建net

net = tf.feature\_column.input\_layer(features, params['feature\_columns'])

for units in params['hidden\_units']:

net = tf.layers.dense(net, units=units, activation=tf.nn.relu)

logits = tf.layers.dense(net, params['n\_classes'], activation=None)

predicted\_classes = tf.argmax(logits, 1) # Compute predictions.

loss = tf.losses.sparse\_softmax\_cross\_entropy(labels=labels, logits=logits) # Compute loss

accuracy = tf.metrics.accuracy(labels=labels, predictions=predicted\_classes,name='acc\_op')

tf.summary.scalar('accuracy', accuracy[1]) #记录信息

# predict

if mode == tf.estimator.ModeKeys.PREDICT:

predictions = {

'class\_ids': predicted\_classes[:, tf.newaxis],

'probabilities': tf.nn.softmax(logits),

'logits': logits,

}

return tf.estimator.EstimatorSpec(mode, predictions=predictions)

# evaluate

if mode == tf.estimator.ModeKeys.EVAL:

metrics = {'accuracy': accuracy}

return tf.estimator.EstimatorSpec(mode, loss=loss, eval\_metric\_ops=metrics)

# train

optimizer = tf.train.AdagradOptimizer(learning\_rate=0.1)

train\_op = optimizer.minimize(loss, global\_step=tf.train.get\_global\_step())

return tf.estimator.EstimatorSpec(mode, loss=loss, train\_op=train\_op)

#使用同tf.estimator.DNNClassifier

classifier = **tf.estimator.Estimator**(

model\_fn=my\_model,

params={

'feature\_columns': my\_feature\_columns,

'hidden\_units': [10, 10],

'n\_classes': 3

}

)

保存和恢复模型 Estimators

Checkpoints #format dependent on the code that created the model.

SavedModel #format independent of the code that created the model.

只要定义模型文件夹路径，模型保存和恢复自动化处理

By default, the Estimator saves checkpoints in the model\_dir according to the following schedule:

* Writes a checkpoint every 10 minutes (600 seconds).
* Writes a checkpoint when the train method starts (first iteration) and completes (final iteration).
* Retains only the 5 most recent checkpoints in the directory.

#修改保存模型频率

my\_checkpointing\_config = **tf.estimator.RunConfig**(

save\_checkpoints\_secs = 20\*60, # Save checkpoints every 20 minutes.

keep\_checkpoint\_max = 10, # Retain the 10 most recent checkpoints.

)

classifier = tf.estimator.DNNClassifier(

feature\_columns=my\_feature\_columns,

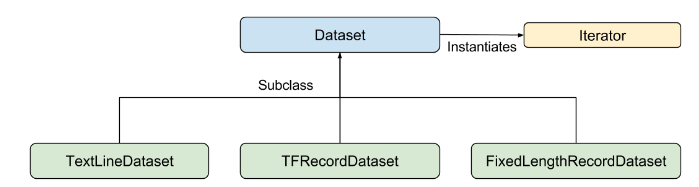
hidden\_units=[10, 10],

n\_classes=3,

**model\_dir='models/iris',**

**config=my\_checkpointing\_config**)

数据



Dataset sequence, 元素含一个或多个 tf.Tensor objects

Base class containing methods to create and transform datasets. Also allows you to initialize a dataset from data in memory, or from a Python generator.

from tf.data import Dataset, TextLineDataset, TFRecordDataset, Iterator

**创建dataset (和Spark，pandas数据框有点像）**

方式1： 基于tensor, numpy 或提供的数据函数

# 元素含单tensor

dataset1 = **Dataset.from\_tensor\_slices**(tf.random\_uniform([4, 10]))

print(dataset1.output\_types) # ==> "tf.float32"

print(dataset1.output\_shapes) # ==> "(10,)"

# 元素含多tensor，建议用{key:value}方式

dataset = Dataset.from\_tensor\_slices(

{"a": tf.random\_uniform([4]),

"b": tf.random\_uniform([4, 100], maxval=100, dtype=tf.int32)})

#直接从numpy构成dataset

dataset = tf.data.Dataset.from\_tensor\_slices({

"features": features,

"labels": labels})

data\_np = [[0, 1,], [2, 3,], [4, 5,], [6, 7,],]

slices = tf.data.Dataset.from\_tensor\_slices(data\_np)

dataset = tf.data.Dataset.**range**(100)

dataset3 = tf.data.Dataset.zip((dataset1, dataset2))

方式2： TextLineDataset - Reads lines from text files.

filenames = ["/var/data/file1.txt", "/var/data/file2.txt"]

dataset = TextLineDataset(filenames).skip(1) # dataset containing the text lines.

方式3： TFRecordDataset - Reads records from TFRecord files.

filenames = ["/var/data/file1.tfrecord", "/var/data/file2.tfrecord"]

dataset = TFRecordDataset(filenames)

**数据预处理 函数式编程**

dataset = dataset.**map**(lambda x: ...)

dataset = dataset.**flat\_map**(lambda x, y: ...)

dataset = dataset.**filter**(lambda x, (y, z): ...)

dataset = dataset.shuffle(buffer\_size=10000) # randomly shuffles the input dataset

dataset = dataset.batch(32) #mini-batch

dataset = dataset.repeat() # Repeat the input indefinitely.

dataset = dataset.repeat(epochs) #iterate over a dataset in multiple epochs

#图像预处理

#tensorflow函数

def \_parse\_function(filename, label):

image\_string = tf.read\_file(filename)

image\_decoded = tf.image.decode\_image(image\_string)

image\_resized = tf.image.resize\_images(image\_decoded, [28, 28])

return image\_resized, label

filenames = tf.constant(["/var/data/image1.jpg", "/var/data/image2.jpg", ...])

labels = tf.constant([0, 2, ...])

dataset = tf.data.Dataset.from\_tensor\_slices((filenames, labels))

dataset = dataset.map(\_parse\_function)

tf.image提供的图像数据操作

tf.image.rag\_to\_grayscale

tf.image.convert\_image\_dtype tf.uint8 → tf.float，且归一化

tf.image.central\_crop

tf.image.crop\_to\_bounding\_box

tf.image.pad\_to\_bounding\_box

tf.image.resize\_image\_with\_crop\_or\_pad

tf.image.flip\_left\_right

tf.image.flip\_up\_down

tf.image.adjust\_brightness

tf.image.adjust\_contrast

tf.image.adjust\_hue

tf.image.adjust\_saturation

tf.image.rgb\_to\_hsv

#集成python函数进行预处理

import cv2

#python函数

def \_read\_py\_function(filename, label):

image\_decoded = cv2.imread(filename.decode(), cv2.IMREAD\_GRAYSCALE)

return image\_decoded, label

# tensorflow函数

def \_resize\_function(image\_decoded, label):

image\_decoded.set\_shape([None, None, None])

image\_resized = tf.image.resize\_images(image\_decoded, [28, 28])

return image\_resized, label

dataset = dataset.map(lambda filename, label:

tuple(**tf.py\_func**(\_read\_py\_function, [filename, label], [tf.uint8, label.dtype])))

dataset = dataset.map(\_resize\_function)

**迭代元素**

#一次遍历，直到tf.errors.OutOfRangeError

iterator = **dataset.make\_one\_shot\_iterator**()

next\_element = **iterator.get\_next**()

sess = tf.Session()

while True:

try:

sess.run(next\_element)

except tf.errors.OutOfRangeError:

print("End of dataset") # ==> "End of dataset"

#feedable iterator + tf.placeholder 动态改变Iterator

#实现交替training and validation

train\_ds = Dataset.range(100).repeat()

validation\_ds = Dataset.range(50)

handle = tf.placeholder(tf.string, shape=[])

iterator = **Iterator.from\_string\_handle**(handle,train\_ds.output\_types, train\_ds.output\_shapes)

next\_element = iterator.get\_next()

train\_iterator = train\_ds.make\_one\_shot\_iterator()

validation\_iterator = validation\_ds.**make\_initializable\_iterator**()

training\_handle = sess.run(train\_**iterator.string\_handle**())

validation\_handle = sess.run(validation\_iterator.string\_handle())

while True:

for \_ in range(200): #训练

sess.run(next\_element, feed\_dict={handle: training\_handle})

sess.run(validation\_iterator.initializer)

for \_ in range(50): #测试

sess.run(next\_element, feed\_dict={handle: validation\_handle})

Eager execution

TensorFlow's eager execution is an imperative programming environment that **evaluates operations immediately, without building graphs**

With graph execution, program state (such as the variables) is stored in global collections and their lifetime is managed by the tf.Session object.

during eager execution the lifetime of state objects is determined by the lifetime of their corresponding Python object.

While eager execution makes development and debugging more interactive, TensorFlow graph execution has advantages for distributed training, performance optimizations, and production deployment

通过关掉eager execution the same code written for eager execution will also build a graph during graph execution. So write compatible code

import tensorflow as tf

tf.enable\_eager\_execution()

a = tf.constant([[1, 2],[3, 4]])

1. numpy() #获取值

a = None #回收a的GPU内存

#求微分

import tensorflow.contrib.eager as tfe

w = tfe.Variable([[1.0]])

with tf.GradientTape() as tape:

loss = w \* w

grad = tape.gradient(loss, [w])

grad.numpy()

#求函数微分

def square(x):

return tf.multiply(x, x)

grad = tfe.gradients\_function(square) #一阶微分

gradgrad = tfe.gradients\_function(lambda x: grad(x)[0]) #二阶微分

gradgradgrad = tfe.gradients\_function(lambda x: gradgrad(x)[0]) #三阶微分

square(3.) # => 9.0

grad(3.) # => [6.0]

gradgrad(3.) # => [2.0]

gradgradgrad(3.) # => [None]

def abs(x):

return x if x > 0. else -x

grad = tfe.gradients\_function(abs)

grad(3.) # => [1.0]

grad(-3.) # => [-1.0]

#对象保存和恢复

checkpoint = tfe.Checkpoint(a=a) # save as "a"

save\_path = checkpoint.save('./ckpt/')

checkpoint.restore(save\_path)

#模型保存和恢复

model = MyModel()

optimizer = tf.train.AdamOptimizer(learning\_rate=0.001)

checkpoint\_dir = ‘/path/to/model\_dir’

checkpoint\_prefix = os.path.join(checkpoint\_dir, "ckpt")

root = tfe.Checkpoint(optimizer=optimizer, model=model,

optimizer\_step=tf.train.get\_or\_create\_global\_step())

root.save(file\_prefix=checkpoint\_prefix)

root.restore(tf.train.latest\_checkpoint(checkpoint\_dir))

Build a model

from tf.keras.layers import Dense

方法1：Sequential

model = tf.keras.Sequential([

Dense(10, input\_shape=(784,)), # must declare input shape

Dense(10)

])

方法2： 函数式

class MNISTModel(tf.keras.Model):

def \_\_init\_\_(self):

super(MNISTModel, self).\_\_init\_\_()

self.dense1 = Dense(units=10)

self.dense2 = Dense(units=10)

def call(self, input):

"""Run the model."""

result = self.dense1(input)

result = self.dense2(result)

result = self.dense2(result) # reuse variables from dense2 layer

return result

model = MNISTModel()

自定义层 （同Keras自定义层）

class MySimpleLayer(tf.keras.layers.Layer):

def \_\_init\_\_(self, output\_units):

self.output\_units = output\_units

def build(self, input):

self.kernel = self.add\_variable("kernel", [input.shape[-1], self.output\_units])

def call(self, input):

return tf.matmul(input, self.kernel)

例子：手写体

from **tf.layers** import conv2d, max\_pooling2d, dense, dropout

from **tf.nn** import relu, softmax

from **tf.losses** import sparse\_softmax\_cross\_entropy

from **tf.train** import GradientDescentOptimizer

from **tf.metrics** import accuracy

def cnn\_model\_fn(features, labels, mode):

input\_layer = tf.reshape(features["x"], [-1, 28, 28, 1])

conv1 = conv2d(inputs=input\_layer, filters=32, kernel\_size=[5, 5], padding="same", activation=relu)

pool1 = max\_pooling2d(inputs=conv1, pool\_size=[2, 2], strides=2)

conv2 = conv2d(inputs=pool1,filters=64,kernel\_size=[5, 5],padding="same",activation=relu)

pool2 = max\_pooling2d(inputs=conv2, pool\_size=[2, 2], strides=2)

pool2\_flat = tf.reshape(pool2, [-1, 7 \* 7 \* 64])

dense = dense(inputs=pool2\_flat, units=1024, activation=relu)

dropout = dropout(inputs=dense, rate=0.4, training=mode == tf.estimator.ModeKeys.TRAIN)

logits = dense(inputs=dropout, units=10) # Logits layer

predictions = {

"classes": tf.argmax(input=logits, axis=1),

"probabilities": softmax(logits, name="softmax\_tensor")

}

if mode == tf.estimator.ModeKeys.PREDICT:

return tf.estimator.EstimatorSpec(mode=mode, predictions=predictions)

loss = sparse\_softmax\_cross\_entropy(labels=labels, logits=logits)

if mode == tf.estimator.ModeKeys.TRAIN:

optimizer = GradientDescentOptimizer(learning\_rate=0.001)

train\_op = optimizer.minimize(loss=loss,global\_step=tf.train.get\_global\_step())

return tf.estimator.EstimatorSpec(mode=mode, loss=loss, train\_op=train\_op)

eval\_metric\_ops = {"accuracy": accuracy(labels=labels, predictions=predictions["classes"])}

return tf.estimator.EstimatorSpec(mode=mode, loss=loss, eval\_metric\_ops=eval\_metric\_ops)

def main(unused\_argv):

(train\_data, train\_labels) # np.array

(eval\_data, eval\_labels)

mnist\_classifier = tf.estimator.**Estimator**(model\_fn=cnn\_model\_fn, model\_dir="model")

# Set up logging for predictions,log the values in the "Softmax" tensor with label "probabilities"

tensors\_to\_log = {"probabilities": "softmax\_tensor"}

logging\_hook = tf.train.LoggingTensorHook(tensors=tensors\_to\_log, every\_n\_iter=50)

# Train the model

train\_input\_fn = tf.estimator.inputs.numpy\_input\_fn(

x={"x": train\_data}, y=train\_labels, batch\_size=100, num\_epochs=None, shuffle=True)

mnist\_classifier.**train**(input\_fn=train\_input\_fn, steps=20000,hooks=[logging\_hook])

# Evaluate the model

eval\_input\_fn = tf.estimator.inputs.numpy\_input\_fn(

x={"x": eval\_data}, y=eval\_labels, num\_epochs=1, shuffle=False)

eval\_results = mnist\_classifier.**evaluate**(input\_fn=eval\_input\_fn)

**Low Level APIs --TensorFlow Core**

定义计算图(tf.Graph) ＋ 运行计算图（tf.Session）

Tensor (含属性：data type and shape)

data type: tf.string, tf.int16, tf.float64, tf.complex64, ...

除了能够将张量的每一维指定为固定长度，也可以将None作为某一维的值，使该张量具有可变长度。

shape = [None, 2] #行数任意，列数为2的矩阵形状

* Tensor <- numpy

operator输入标准python数据类型,自动将它们转化为tensor，输出tensor 是 numpy ndarray 对象；tensorflow的数据类型是基于numpy的数据类型的。实际上，语句np.int32==tf.int32

np.array(50, dtype=np.int32) #0阶张量即标量

np.array([1., 2.], dtype=np.float32) #1阶张量即向量

np.array([[1, 2, 3],[4, 5, 6]], dtype=np.int32) #2阶张量即矩阵

* Tensor <- tf.placeholder

利用占位节点添加输入

a = tf.placeholder(tf.int32, shape=[2])

b = tf.reduce\_sum(a)

sess.run(b, feed\_dict={a: np.array([5, 3], dtype=np.int32)})

* Tensor <- tf.Variable

tensor对象和Op对象都是不可变的，variable包含了在对Session.run()多次调用中可持久化的可变张量值, 通常会将模型参数表示为一组变量.

#默认tf.float32, 初始值基于tf.glorot\_uniform\_initializer随机化

# 变量名是w, shape(3, 3, 10)

w = tf.get\_variable("w", [3, 3, 10])

w = tf.get\_variable("w", [1, 2, 3], dtype=tf.int32, initializer=tf.zeros\_initializer)

w = tf.get\_variable("w", dtype=tf.int32, initializer=tf.constant([23, 42]))

#通过numpy赋初值

myxor = tf.Variable([[False, True],[True, False]], tf.bool)

linear\_squares = tf.Variable([[4], [9], [16], [25]], tf.int32)

#通过tf.op赋初值

weights = tf.Varaible(tf.zeros([2, 2]))

# 由权重w1初始化w2

w2 = tf.Variable(w1.initialized\_value(), name="w2")

#weights变量一般作为trainable参数，Optimizer会自动修改Variable对象的值。#若该权重不参与训练，可以设置参数trainable=False

#也可以通过将权重添加到LOCAL\_VARIABLES中

weight\_no = tf.Variable(0, trainable=False)

By default every tf.Variable gets placed in the following two collections:

tf.GraphKeys.GLOBAL\_VARIABLES --- variables that can be shared across multiple devices,

tf.GraphKeys.TRAINABLE\_VARIABLES --- variables for which TensorFlow will calculate gradients.

tf.GraphKeys.LOCAL\_VARIABLES --- variables are not trainable

初始化变量

#为使用Variable对象，必须在一个Session对象内对Variable对象进行初始化。这样#会使Session对象开始追踪这个Variable对象的值的变化

#由于不同Session对象会各自独立地维护Variable对象的值，因此每个Session对象#都拥有自己的，在Graph对象中定义的Variable对象的当前值

# initializing all variables in the tf.GraphKeys.GLOBAL\_VARIABLES collection

默认情况下，tf.global\_variables\_initializer并不指定变量初始化顺序，所以变量初始化若存在相关性，可能会出错，最好用如下方式：

v = tf.get\_variable("v", shape=(), initializer=tf.zeros\_initializer())

w = tf.get\_variable("w", initializer=v.initialized\_value() + 1)

session.run(tf.global\_variables\_initializer())

images = tf.zeros([batch, h, w, 3])

tf.ones([...])

tf.range(-1, 3)

tf.random\_uniform([3, 3, 3], minval=0, maxval=10)

tf.random\_normal([3, 3, 3], mean=0.0, stddev=2.0)

tf.truncated\_normal([2, 2], mean, stddev) #创建[mean-2std, mean+2std]

#张量切片

image = images[0, :, :, :]

print(sess.run(image))

#获取张量信息

images.dtype

sess.run(tf.shape(images)) #求张量images的shape

sess.run(tf.rank(images)) #求张量images的rank

matrix = tf.reshape(images, [batch, -1])

#张量数量类型

tf.to\_float(weights)

weights\_float = tf.cast(weights, dtype=tf.float32)

节点Op

a = np.array([2, 3], dtype=np.int32) tensor

b = tf.constant([5, 3], name=”input\_b”) 节点，指向常量操作输出Tensor句柄

c = tf.add(a, b) #变量c为指向该Op输出的Tensor对象的句柄

c = a + b #运算符重载

张量逐元素运算+-\*/, //, %, \*\*, < <= > >= & | ^

tf.matmul(X, W) 矩阵乘

tf.transpose(W)

回归: 总平方差(回归损失函数)

loss = tf.reduce\_sum(tf.squared\_difference(Y\_predicted, Y))

计算图tf.Graph

**计算图本质是一组链接在一起的函数**，每个函数都会将其输出传递给0,1,多个位于这个级联链上的其他函数

计算图（数据流图） ＝ 节点（对数据所做的运算或输入操作）＋边（tensor)

一个 TensorFlow 图描述了计算的过程. 为了进行计算, 图必须在会话里被启动. 会话将图的op分发到诸如CPU或GPU之类的设备上, 同时提供执行 op 的方法. 这些方法执行后, 将产生的 tensor 返回

当TensorFlow库被加载时，它会自动创建一个Graph对象，并将其作为默认的数据流图。因此在Graph.as\_default()上下文管理器之外定义的任何Op, Tensor对象都会自动放置在默认的数据流图中。

在大多数TensorFlow程序中，只使用默认数据流图就足够了。然而，如果需要定义多个相互之间不存在依赖关系的模型，则创建多个Graph对象十分有用。

g1 = tf.get\_default\_graph()

g2 = tf.Graph()

with g1.as\_default():

定义g1的Op 和张量等

with g2.as\_default():

定义g2的Op和张量等

调用tf.matmul(x, y)，创建并添加tf.matmul于默认的graph中，返回tf.Tensor指向tf.matmul操作的结果

调用tf.train.Optimizer.minimize，创建并添加operations and tensors于默认的graph中，返回tf.Operation，当sess.run时，对变量更新梯度

tf会自动给graph里的每个tf.operation赋名字，但建议自己写程序时显式命名tf.Operation

tf.constant(42.0, name="answer")

解析：tf.constant这个操作赋名为"answer", 若graph里已有answer，那么这个操作会赋名为answer\_1 or answer\_2 ...直至独一无二；返回的tf.Tensor赋名为"answer:0"

with tf.name\_scope("outer"):

c\_2 = tf.constant(2, name="c") # => operation named "outer/c"

# Name scopes nest like paths in a hierarchical file system.

with tf.name\_scope("inner"):

c\_3 = tf.constant(3, name="c") # => operation named "outer/inner/c"

运行计算图（tf.Session）

A session encapsulates the state of the TensorFlow runtime, and runs TensorFlow operations. If a tf.Graph is like a .py file, a tf.Session is like the python executable.

tf.Session owns physical resources (such as GPUs and network connections), it is typically used as a context manager (in a with block)

sess利用节点之间的依赖关系只对必要的**节点执行运算**

#运行计算图，获取计算图中的某节点输出

a = tf.placeholder(tf.float32)

b = a \* 3

with tf.Session() as sess: 等价于tf.Session(graph=tf.get\_default\_graph())

sess.run(tf.initialize\_all\_variables()) #执行初始化Variable对象所需的计算

b\_val = sess.run(fetches=[b], feed\_dict={a: 15})

fetches：接收任意的数据流图元素(Op or Tensor对象），后者指定了用户希望执行的对象。若请求的是Tensor对象，则run()输出的是Numpy数组； 若请求的是Op，则输出为None

feed\_dict: 用于覆盖数据流图中的Tensor对象值，**字典键为Tensor对象的句柄**

#遍历数据

my\_data = [[0, 1,], [2, 3,], [4, 5,], [6, 7,]]

slices = tf.data.Dataset.from\_tensor\_slices(my\_data)

next\_item = slices.make\_one\_shot\_iterator().get\_next()

while True:

try:

print(sess.run(next\_item))

except tf.errors.OutOfRangeError:

break

#获取计算图内部执行信息

y = tf.matmul([[37.0, -23.0], [1.0, 4.0]], tf.random\_uniform([2, 2]))

with tf.Session() as sess:

# Define options for the `sess.run()` call.

options = tf.RunOptions()

options.output\_partition\_graphs = True

options.trace\_level = tf.RunOptions.FULL\_TRACE

metadata = tf.RunMetadata() # Define a container for the returned metadata.

sess.run(y, options=options, run\_metadata=metadata)

print(metadata.partition\_graphs) # Print the subgraphs that executed on each device.

print(metadata.step\_stats) # Print the timings of each operation that executed.

# 查看分布在GPU and CPU设备上运算的情况

sess = tf.Session(config=tf.ConfigProto(log\_device\_placement=True))

#选择设备进行运算

/job:<JOB\_NAME>/task:<TASK\_INDEX>/device:<DEVICE\_TYPE>:<DEVICE\_INDEX>

# 没有指定，依赖于内部实现，若是GPU编程，选用GPU

weights = tf.random\_normal(...)

with tf.device("/device:CPU:0"):

img = tf.decode\_jpeg(tf.read\_file("img.jpg"))

with tf.device("/device:GPU:0"):

result = tf.matmul(weights, img)

c = tf.matmul(a, b)

sess = tf.Session(config=tf.ConfigProto(log\_device\_placement=True))

TensorBoard: 可视化学习

TensorBoard 通过读取 TensorFlow 的事件文件来运行

#在定义计算图时，添加想summary的节点

**tf.summary.scalar**('learning rate', lr, name='lr\_summary')

tf.summary.scalar('loss function', loss, name='loss\_summary')

#在变量初始化后，将所有summary合并到一个Op中

merged\_summary\_op = **tf.merge\_all\_summaries()**

#summary.FileWriter 对象保存来自数据流图的数据和概括流计量

#第一个参数是数据流图的描述在磁盘中的存放路径

#第二个参数是要追踪的数据流图

在指定目录下生成event file,命名如下：

events.out.tfevents.{timestamp}.{hostname}

writer = tf.summary.FileWriter(‘./my\_graph’, **sess.graph**)

for step in total\_step:

session.run(...)

if step % 100 == 0: #每执行100个batch，执行一次汇总

summary\_str = session.run(merged\_summary\_op)

**writer.add\_summary**(summary\_str, step)

writer.close()

#一量执行完以上代码，便可启动TensorBoard

$ tensorboard –-logdir=”.my\_graph”

localhost:6006会显示利用Writer对象要求TensorFlow所保存的信息。

模型

方法一：原始组合Ops tf.nn 不含参数，只计算

#X: (N, H, W, C) RGB空间

#权重W：5\*5表示卷积核大小,1表示输入通道数目, 32表示输出通道数目

#偏置b: 32表示输出通道数目

#strides:(batch\_size\_stride, height\_stride, width\_stride, channels\_stride)

with tf.name\_scope("conv1"):

conv1\_w = tf.Variable(tf.truncated\_normal([5, 5, 1, 32], stddev=0.1))

conv1\_b = tf.Variable(tf.constant(0.1, shape=[32]), name="bias")

c1 = tf.nn.conv2d(X, conv1\_w, strides=[1, 1, 1, 1], padding='SAME')

conv1 = c1 + conv1\_b

conv1 = tf.nn.relu( conv1)

pool1 = tf.nn.max\_pool(conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

tf.nn.relu #激活函数

tf.sigmoid

tf.tanh

tf.nn.local\_response\_normalization #归一化层,一般对tf.nn.relu的输出进行归一化

tf.nn.max\_pool

tf.nn.avg\_pool

tf.nn.dropout(h\_fc1, keep\_prob) #自动处理神经元输出值的scale

tf.nn.softmax(tf.matmul(h\_fc1\_drop, W\_fc2) + b\_fc2) #输出层

获取层张量的shape

sess.run(h\_pool1.get\_shape())

方法二：高级层 tf.layers 同keras, 含权值初始化，偏置初始化，激活函数

from **tf.layers** import Conv2D, MaxPooling2D, Dense, ... 类的使用方式

from **tf.layers** import conv2d, dense, ... 函数使用方式

具体查看API <https://www.tensorflow.org/api_docs/python/tf/layers>

#线性模型

x = tf.placeholder(tf.float32, shape=[None, 3])

linear\_model = Dense(units=1) #类的使用方式

y = linear\_model(x)

y = dense(x, units=1) #函数使用方式

#执行模型

#The layer contains variables that must be initialized before they can be used

init = tf.global\_variables\_initializer()

sess.run(init)

print(sess.run(y, {x: [[1, 2, 3], [4, 5, 6]]}))

保存和加载模型

保存时会生成4个文件

checkpoints

checkpoint #保存一个目录下所有的模型文件列表

\*.ckpt.meta #保存TensorFlow计算图

\*.ckpt #保存计算图变量值

.ckpt.data-00000-of-00001

#和matlab, caffe2一样

tf.train.Saver().save(sess, "\*.ckpt") #保存模型中的所有变量

tf.train.Saver().restore(sess, "\*.ckpt") #恢复模型中的所有变量

#仅保存需要的变量

#my\_v2自己取的变量名，v2:sess中需要保存的变量

tf.train.Saver({"my\_v2": v2}).save(...)

例子：

v1 = tf.get\_variable("v1", [3], initializer=tf.zeros\_initializer)

v2 = tf.get\_variable("v2", [5], initializer=tf.ones\_initializer)

init\_op = tf.global\_variables\_initializer()

saver = tf.train.Saver()

with tf.Session() as sess:

sess.run(init\_op)

sess.run([v1, v2])

saver.save(sess, "./model.ckpt") #保存graph里所有的变量v1, v2

tf.reset\_default\_graph()

# 打印保存模型里所有的变量

chkp.print\_tensors\_in\_checkpoint\_file("./model.ckpt", tensor\_name='', all\_tensors=True)

#指定要导入的变量，注意不需要初始化

v1 = tf.get\_variable("v1", shape=[3])

v2 = tf.get\_variable("v2", shape=[5])

saver = tf.train.Saver()

with tf.Session() as sess:

saver.restore(sess, "./model.ckpt") #恢复指定的变量

print("v1 : %s" % v1.eval())

print("v2 : %s" % v2.eval())

保存和恢复模型--variable, graph and graph metadata

#直接加载已经持久化的图,从而默认加载了TensorFlow计算图上定义的全部变量

import tensorflow as tf

saver = tf.train.import\_meta\_graph("save/model.ckpt.meta")

with tf.Session() as sess:

saver.restore(sess, "save/model.ckpt")

# 通过张量的名称来获取张量

print(sess.run(tf.get\_default\_graph().get\_tensor\_by\_name("v1:0")))

tf.saved\_model APIs

tf.estimator.Estimator

a command-line interface.

保存训练检查点：周期性地保存所有变量，创建检查点checkpoint文件，并在必要时从最近的检查点恢复训练。默认情况下，Saver对象 只会保留最近的5个文件

在TensorFlow中，真正的循环依赖关系是无法表示的，如何解决？

在实际使用中，完全可通过对数据流图进行有限次的复制，然后将它们并排放置，并将代表相邻迭代轮次的副本的输出与输入串接。该过程通常被称为数据流图的“展开”unrolling

分类

字符串特征转数值型特征？

为什么不能为每个可能的取值分配一个数值？如用1代表一等船票，2 and 3分别代表二，三等船票，因为这种方式会为这些取值强加一种实际并不存在的线性关系。我们不能说“三等票是一等票的3倍“

正确的做法是将每个属性特征扩展为N维的布尔型特征，每个可能的取值对应一维。若具备该属性，则相应的维度上取值为1.这样就可以使模型独立地学习到每个可能的取值的重要性。

对于只可能取两种值的属性，如果性别，用单个变量来表示已经足够，这是因为可表达这些值之间的线性关系。如令female = 1, male = 0, 则male = 1 – female, 因此单个权值便可学习同时表示两种状态

将类名称转换为从0开始计的类别索引

label = tf.to\_int32(tf.argmax(tf.to\_int32(tf.pack([

tf.equal(label\_name, [“Iris-setosa”]),

tf.equal(label\_name, [“Iris-versicolor”]),

tf.equal(label\_name, [“Iris-virginica”]) ])), 0)

两类问题：交叉熵

tf.sigmoid(f(X, W, b) sigmoid （概率值）

loss = sum(yi\*log(y\_predictedi )+ (1-yi)log(1-y\_predictedi)) i为case

loss = tf.reduce\_mean(tf.nn.sigmoid\_cross\_entropy\_with\_logits(Y\_predicted, Y))

交叉熵与香农熵区别：

香农熵 H = -sum(yi\*log(yi ))

物理意义：已知字符串中每个字符出现的概率 yi，采用最优编码方案，则字符串编码每个字符的平均位数

交叉熵 H = -sum(yi\*log(y\_predictedi ))

实际中，对符号编码时，得到的概率是y\_predictedi 而非真实概率 yi，则每个符号所需的编码长度会更大，交叉熵允许用户以次优编码方案对字符串编码。

当建模得到的概率是y\_predictedi ＝＝真实概率 yi时，交叉熵取得最小值。交叉熵度量的是模型分布与真实分布的差异信息。若交叉熵越接近于熵，则得到的概率是y\_predictedi 越逼近真实概率yi

多类问题：

tf.nn.softmax(f(X, W, b)) 概率值

#单样本的所有类别概率和为1,

#若<1, 则意味着存在一些隐藏的类别；

#若>1,则说明每个样本可能同时属于多个类别

loss = -sum(sum(yic\*log(y\_predictedic ))) i为case, c为类别

#若每个样本只对应单个类别

#注意：类别可以为0, 1, 2, 3, ...

loss = tf.reduce\_mean(tf.nn.spare\_softmax\_cross\_entropy\_with\_logits(Y\_predicted, Y))

#若每个样本含多个类别，即类别0的概率，类别1的概率,...

#注意：类别只能为one-hot编码

loss = tf.reduce\_mean(tf.nn.softmax\_cross\_entropy\_with\_logits(Y\_predicted, Y))

例子：手写体

输出：y = softmax(xW + b)

损失函数：交叉熵cross-entropy <=> Hy’(y) = - sum (yi’log(yi))

class CNN(object):

def \_\_init\_\_(self):

self.keep\_prob = 0.5

self.y\_class = 10

self.define\_graph() #定义计算图

self.sess = tf.Session()

self.sess.run(tf.global\_variables\_initializer()) #初始化参数

self.restore()

# 恢复上次的训练

def restore(self):

self.saver = tf.train.Saver()

self.initial\_epoches = 0

ckpt = tf.train.get\_checkpoint\_state(os.path.dirname(\_\_file\_\_))

if ckpt and ckpt.model\_checkpoint\_path:

**self.saver**.restore(self.sess, ckpt.model\_checkpoint\_path)

**self.initial\_epoches** = int(ckpt.model\_checkpoint\_path.rsplit('-', 1)[1])

def define\_graph(self, ):

self.X = tf.placeholder(tf.float32, [None, 28, 28, 1]) #输入占位符，待实际填充

self.Y = tf.placeholder(tf.float32, [None, 10])

Y\_predicted = self.inference(self.X)

**self.total\_loss** = self.loss(self.Y, Y\_predicted)

**self.optimizer** = self.optimize(self.total\_loss)

**self.accuracy** = self.acc(self.Y, Y\_predicted)

def inference(self, X):

...

return Y\_predicted

@staticmethod

def loss(Y, Y\_predicted):

return -tf.reduce\_sum(Y \* tf.log(Y\_predicted))

@staticmethod

def acc(Y, Y\_predicted):

#tf.argmax 给出某个tensor对象在某一维上的其数据最大值所在的索引值

#tf.argmax(Y\_predicted, 1)返回的是预测输出的标签值

correct\_prediction = tf.equal(tf.argmax(Y\_predicted, 1), tf.argmax(Y, 1))

return tf.reduce\_mean(tf.cast(correct\_prediction, "float"))

#自动计算梯度和更新参数

@staticmethod

def optimize(total\_loss):

return tf.train.AdamOptimizer(1e-4).minimize(total\_loss)

#return tf.train.GradientDescentOptimizer(0.01).minimize(loss)

def train(self, ds\_train, batch\_size=128, epoches=100):

#计算遍历一次数据集需要的epoch

total\_num = ds\_train.num\_examples

epoch = int(math.ceil(total\_num/float(batch\_size)))

for i in range(**self.initial\_epoches, epoches**):

for j in range(epoch):

batch = ds\_train.next\_batch(batch\_size)

#注意self.optimizer是个op, not a tensor, 返回None, 执行op是为了内部状态运算

\_, loss\_value, acc\_value = self.sess.run(

[**self.optimizer, self.total\_loss, self.accuracy**], #计算输出

feed\_dict={ #输入数据

**self.X**: batch[0].reshape([-1, 28, 28, 1]),

**self.Y**: batch[1].reshape([-1, 10])

})

#保存训练模型

if epoches % 10 == 0:

self.saver(self.sess, 'mnist-model', global\_step=epoches)

self.saver.save(self.sess, 'mnist-model', global\_step=epoches)

return self

def test(self, ds\_test, batch\_size=32):

loop = int(math.ceil(ds\_test.images.shape[0] / float(batch\_size)))

accs = []

for i in range(loop):

batch = ds\_test.next\_batch(batch\_size)

accuracy = self.sess.run(

**self.accuracy,**

feed\_dict={

self.X: batch[0].reshape([-1, 28, 28, 1]),

self.Y: batch[1].reshape([-1, 10])

})

accs.append(accuracy)

accuracy\_mean = sum(accs) / len(accs)