

Caffe使用I

基础使用

1.Caffe的I/O模块

■ 1.1 如何对Layer做参数配置 (Data Layer参数配置为例)

```
layer {  
  name: "cifar"  
  type: "Data"  
  top: "data"  
  top: "label"  
  include {  
    phase: TRAIN  
  }  
  transform_param {  
    scale: 0.00390625  
  }  
  data_param {  
    source:  
"cifar10_train_lmdb"  
    batch_size: 100  
    backend: LMDB  
  }  
}
```

LayerParameter

DataParameter

name: 表示该层的名称，可随意取。

type: 层类型，如果是Data，表示数据来源于LevelDB或LMDB。根据数据的来源不同，数据层的类型也不同（后面会详细阐述）。一般在练习的时候，我们都是采用的LevelDB或LMDB数据，因此层类型设置为Data。

top或bottom: 每一层用bottom来输入数据，用top来输出数据。如果只有top没有bottom，则此层只有输出，没有输入。反之亦然。如果有多个top或多个bottom，表示有多个blobs数据的输入和输出。
data 与 label: 在数据层中，至少有一个命名为data的top。如果有第二个top，一般命名为label。这种(data,label)配对是分类模型所必需的。

include: 一般训练的时候和测试的时候，模型的层是不一样的。该层(layer)是属于训练阶段的层，还是属于测试阶段的层，需要用include来指定。如果没有include参数，则表示该层既在训练模型中，又在测试模型中。

Transformations: 数据的预处理，可以将数据变换到定义的范围內。如设置scale为0.00390625，实际上就是1/255，即将输入数据由0-255归一化到0-1之间。

- 所有数据预处理都在这里设置：

```
transform_param {  
  scale: 0.00390625  
  mean_file_size: "examples/cifar10/mean.binaryproto" # 用一个配置文件来  
  进行均值操作  
  mirror: 1 # 1表示开启镜像，0表示关闭，也可用ture和false来表示  
  crop_size: 227 # 剪裁一个 227*227的图块，在训练阶段随机剪裁，在测试阶段  
  从中间裁剪  
}
```

* 通常数据的预处理（如减去均值, 放大缩小, 裁剪和镜像等），Caffe使用OpenCV做处理。

```

message DataParameter {
  enum DB {
    LEVELDB = 0;
    LMDB = 1;
  }
  // Specify the data source.
  optional string source = 1;
  // Specify the batch size.
  optional uint32 batch_size = 4;
  // The rand_skip variable is for the data layer to skip a few data points
  // to avoid all asynchronous sgd clients to start at the same point. The skip
  // point would be set as rand_skip * rand(0,1). Note that rand_skip should not
  // be larger than the number of keys in the database.
  // DEPRECATED. Each solver accesses a different subset of the database.
  optional uint32 rand_skip = 7 [default = 0];
  optional DB backend = 8 [default = LEVELDB];
  // DEPRECATED. See TransformationParameter. For data pre-processing, we can do
  // simple scaling and subtracting the data mean, if provided. Note that the
  // mean subtraction is always carried out before scaling.
  optional float scale = 2 [default = 1];
  optional string mean_file = 3;
  // DEPRECATED. See TransformationParameter. Specify if we would like to randomly
  // crop an image.
  optional uint32 crop_size = 5 [default = 0];
  // DEPRECATED. See TransformationParameter. Specify if we want to randomly mirror
  // data.
  optional bool mirror = 6 [default = false];
  // Force the encoded image to have 3 color channels
  optional bool force_encoded_color = 9 [default = false];
  // Prefetch queue (Number of batches to prefetch to host memory, increase if
  // data access bandwidth varies).
  optional uint32 prefetch = 10 [default = 4];
}

```

1、数据来自于数据库（如LevelDB和LMDB）

层类型（layer type）:Data

必须设置的参数：

source: 包含数据库的目录名称，如examples/mnist/mnist_train_lmdb

batch_size: 每次处理的数据个数，如64

可选的参数：

rand_skip: 在开始的时候，路过某个数据的输入。通常对异步的SGD很有用。

backend: 选择是采用LevelDB还是LMDB, 默认是LevelDB.

2、数据来自于内存

层类型: MemoryData

必须设置的参数:

batch_size: 每一次处理的数据个数, 比如2

channels: 通道数

height: 高度

width: 宽度

示例:

```
layer {
  top: "data"
  top: "label"
  name: "memory_data"
  type: "MemoryData"
  memory_data_param{
    batch_size: 2
    height: 100
    width: 100
    channels: 1
  }
  transform_param {
    scale: 0.0078125
    mean_file: "mean.proto"
    mirror: false
  }
}
```

3、数据来自于图片

层类型: ImageData

必须设置的参数:

source: 一个文本文件的名称, 每一行给定一个图片文件的名称和标签 (label)

batch_size: 每一次处理的数据个数, 即图片数
可选参数:

rand_skip: 在开始的时候, 路过某个数据的输入。通常对异步的SGD很有用。

shuffle: 随机打乱顺序, 默认值为false

new_height, new_width: 如果设置, 则将图片进行resize

示例:

```
layer {
  name: "data"
  type: "ImageData"
  top: "data"
  top: "label"
  transform_param {
    mirror: false
    crop_size: 227
    mean_file: "data/ilsrvrc12/
imagenet_mean.binaryproto"
  }
  image_data_param {
    source: "examples/_temp/file_list.txt"
    batch_size: 50
    new_height: 256
    new_width: 256
  }
}
```

4、数据来自于HDF5

层类型: HDF5Data

必须设置的参数:

source: 读取的文件名称

batch_size: 每一次处理的数据个数

示例:

```
layer {
  name: "data"
  type: "HDF5Data"
  top: "data"
  top: "label"
  hdf5_data_param {
    source: "examples/
hdf5_classification/data/train.txt"
    batch_size: 10
  }
}
```

5、数据来源于Windows

层类型: WindowData

必须设置的参数:

source: 一个文本文件的名称

batch_size: 每一次处理的数据个数, 即图片数

示例:

```
layer{...
  window_data_param...
}
```

■ 1.2 将图片数据转化为LMDB数据

- 第一步：创建图片文件列表清单，一般为一个txt文件，一行一张图片
- 第二步：使用Caffe工具命令

convert_imageset [FLAGS] [ROOTFOLDER/] [LISTFILE] [DB_NAME]

需要带四个参数：

FLAGS: 图片参数组

-gray: 是否以灰度图的方式打开图片。程序调用opencv库中的imread()函数来打开图片，默认为false

-shuffle: 是否随机打乱图片顺序。默认为false

-backend:需要转换成的db文件格式，可选为leveldb或lmdb,默认为lmdb

-resize_width/resize_height: 改变图片的大小。在运行中，要求所有图片的尺寸一致，因此需要改变图片大小。程序调用opencv库的resize（）函数来对图片放大缩小，默认为0，不改变

-check_size: 检查所有的数据是否有相同的尺寸。默认为false,不检查

-encoded: 是否将原图片编码放入最终的数据中，默认为false

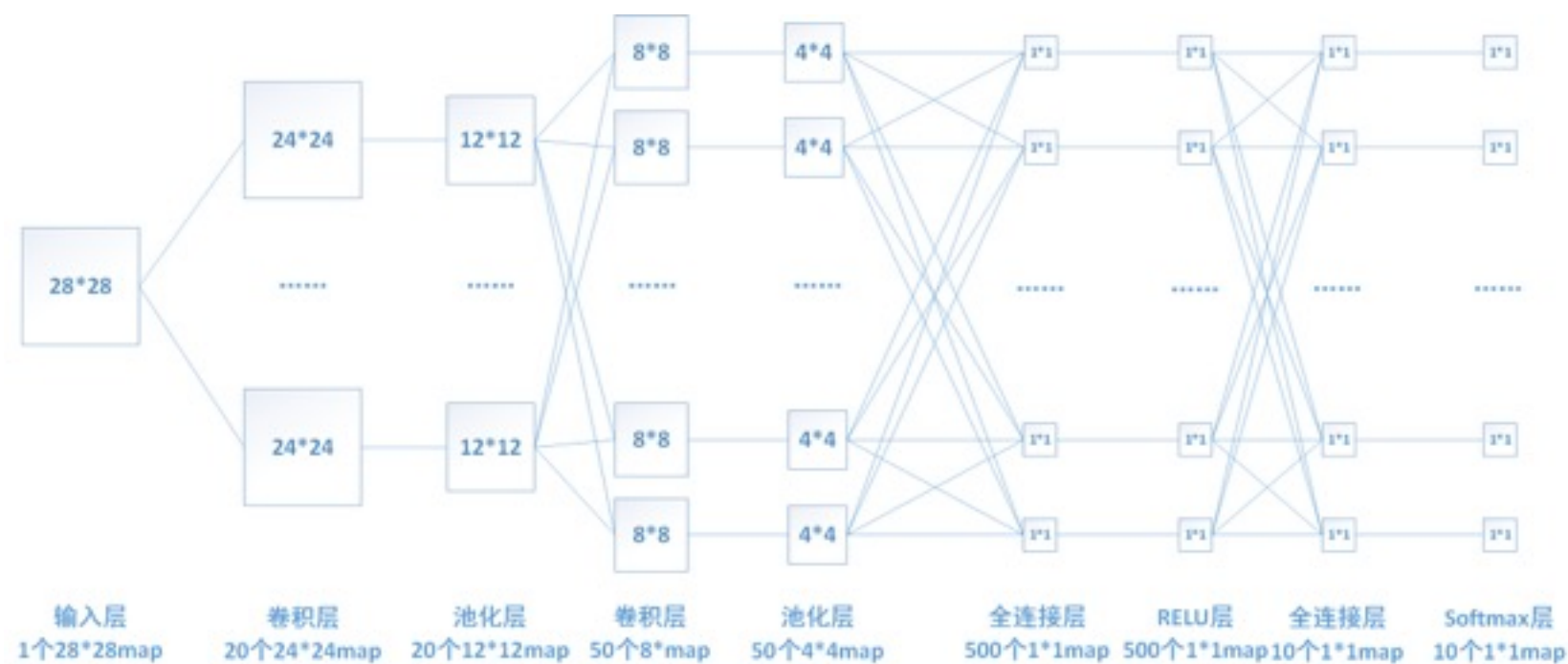
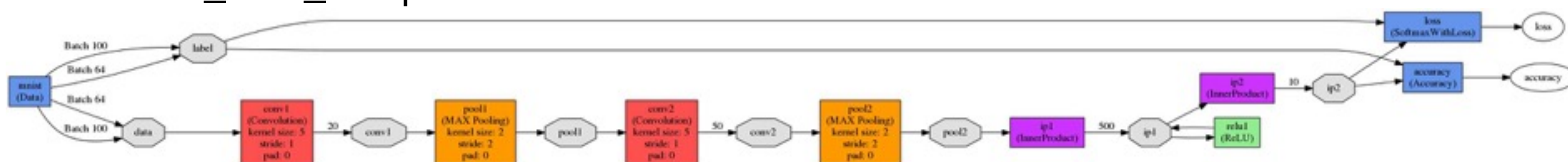
- 1.3 实操演示：将图片数据转化为LMDB数据

2.Caffe中五种类型的层的实现与参数配置

2.Caffe中五种层的实现和参数配置

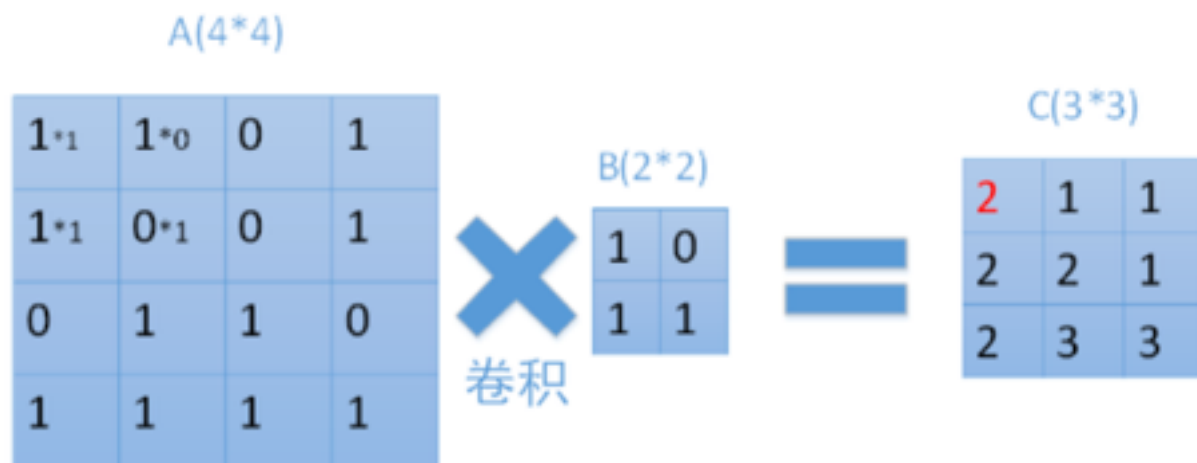
Data: mnist

Net : lenet_train_test.prototxt



- 卷积层 (Convolution)
- 池化层 (Pooling)
- 激活层 (ReLU、Sigmoid、TanH、AbsVal、Power)
- 全连接层 (InnerProduct)
- softmax层 (SoftmaxWithLoss、Softmax)

2.1 卷积层



例子:

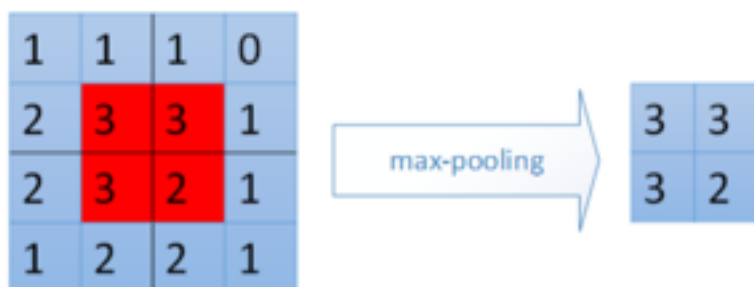
输入为 28×28 的图像，经过 5×5 的卷积之后，得到一个 $(28-5+1) \times (28-5+1) = 24 \times 24$ 的map。

* 每个map是不同卷积核在前一层每个map上进行卷积，并将每个对应位置上的值相加然后再加上一个偏置项。

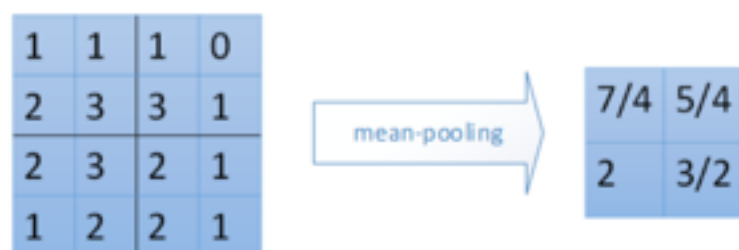
```
layer {
  name: "conv2"
  type: "Convolution"
  bottom: "pool1"
  top: "conv2"
  param {
    lr_mult: 1          #学习率1，和权值更新相关
  }
  param {
    lr_mult: 2          #学习率2，和权值更新相关
  }
  convolution_param {
    num_output: 50      # 50个输出的map
    kernel_size: 5       #卷积核大小为5*5
    stride: 1            #卷积步长为1
    weight_filler {      #权值初始化方式
      type: "xavier"     #默认为"constant",值全为0，很多时候我们也可以用"xavier"或者"gaussian"来进行初始化
    }
    bias_filler {        #偏置值的初始化方式
      type: "constant"   #该参数的值和weight_filler类似，一般设置为"constant"，值全为0
    }
  }
}
```

2.2 池化层

max-pooling:



mean-pooling:



例子:

输入为卷积层1的输出，大小为 24×24 ，对每个不重叠的 2×2 的区域进行降采样。对于max-pooling，选出每个区域中的最大值作为输出。而对于mean-pooling，需计算每个区域的平均值作为输出。最终，该层输出一个 $(24/2) \times (24/2)$ 的map

```
layer {  
  name: "pool1"  
  type: "Pooling"  
  bottom: "conv1"  
  top: "pool1"  
  pooling_param {  
    pool: MAX #Pool为池化方式，默  
              认为MAX，可以选择的参数有  
              MAX、AVE、STOCHASTIC  
    kernel_size: 2 #池化区域的大小，  
                  也可以用kernel_h和kernel_w分别设  
                  置长和宽  
    stride: 2 #步长，即每次池化区域  
              左右或上下移动的距离，一般和  
              kernel_size相同，即为不重叠池  
              化。也可以也可以小于  
              kernel_size，即为重叠池化，  
              Alexnet中就用到了重叠池化的方法  
  }  
}
```

2.3 全连接层



50*4*4=800个输入结点和500个输出结点

#参数和卷积层表达一样

```
layer {
  name: "ip1"
  type: "InnerProduct"
  bottom: "pool2"
  top: "ip1"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  inner_product_param {
    num_output: 500
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
```

2.4 激活函数层

- 激活函数作用：激活函数是用来引入非线性因素的。
- 激活函数一般具有以下性质：
 - 非线性：线性模型的不足我们前边已经提到。
 - 处处可导：反向传播时需要计算激活函数的偏导数，所以要求激活函数除个别点外，处处可导。
 - 单调性：当激活函数是单调的时候，单层网络能够保证是凸函数。
 - 输出值的范围：当激活函数输出值是有限的时候，基于梯度的优化方法会更加稳定，因为特征的代表受有限权值的影响更显著

```
layer {  
  name: "relu1"  
  type: "ReLU"  
  bottom: "ip1"  
  top: "ip1"  
}
```

```
layer {  
  name: "layer"  
  bottom: "in"  
  top: "out"  
  type: "Power"  
  power_param {  
    power: 2  
    scale: 1  
    shift: 0  
  }  
}
```

Type为该层类型，可取值分别为：

(1)ReLU：表示我们使用relu激活函数，relu层支持in-place计算，这意味着该层的输入和输出共享一块内存，以避免内存的消耗。

(2)Sigmoid：代表使用sigmoid函数；

(3) TanH：代表使用tanh函数；

(4) AbsVal：计算每个输入的绝对值 $f(x)=\text{Abs}(x)$

(5)power对每个输入数据进行幂运算

$f(x) = (\text{shift} + \text{scale} * x) ^ \text{power}$


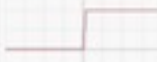



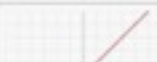






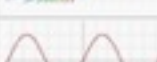

层类型：Power

可选参数：

power: 默认为1

scale: 默认为1

激活函数列表：

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
Tanh		$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2 + 1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1 + e^{-x}}$
Bent Identity		$f(x) = \frac{\sqrt{x^2 + 1} - 1}{2} + x$	$f'(x) = \frac{x}{2\sqrt{x^2 + 1}} + 1$
SoftExponential		$f(\alpha, x) = \begin{cases} -\frac{\log_e(1 - \alpha(x + \alpha))}{\alpha} & \text{for } \alpha < 0 \\ x & \text{for } \alpha = 0 \\ \frac{e^{\alpha x} - 1}{\alpha} + \alpha & \text{for } \alpha > 0 \end{cases}$	$f'(\alpha, x) = \begin{cases} \frac{1}{1 - \alpha(x + \alpha)} & \text{for } \alpha < 0 \\ 1 & \text{for } \alpha = 0 \\ e^{\alpha x} & \text{for } \alpha > 0 \end{cases}$
Sine		$f(x) = \sin(x)$	$f'(x) = \cos(x)$
Sinc		$f(x) = \begin{cases} 1 & \text{for } x = 0 \\ \frac{\sin(x)}{x} & \text{for } x \neq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x = 0 \\ \frac{\cos(x)}{x} - \frac{\sin(x)}{x^2} & \text{for } x \neq 0 \end{cases}$
Gaussian		$f(x) = e^{-x^2}$	$f'(x) = -2xe^{-x^2}$

- 一般比较常见的激活函数有sigmoid、tanh和Relu，其中Relu由于效果最好，现在使用的比较广泛。

relu函数的表达式为 $f(x) = \begin{cases} 0, & x < 0 \\ x, & x \geq 0 \end{cases}$ ，所以前向传播时，大于0的输入不变，小于0的置零即可。

2.5 softmax层

Softmax回归模型是logistic回归模型在多分类问题上的推广，在多分类问题中，待分类的类别数量大于2，且类别之间互斥。Softmax公式：

$$f(z_j) = \frac{e^{z_j}}{\sum_{i=1}^n e^{z_i}}$$

通常情况下softmax会被用在网络中的最后一层，用来进行最后的分类和归一化。

#可以计算给出每个样本对应的损失函数值

```
layer {  
  name: "loss"  
  type:  
  "SoftmaxWithLoss"  
  bottom: "ip2"  
  bottom: "label"  
  top: "loss"  
}
```

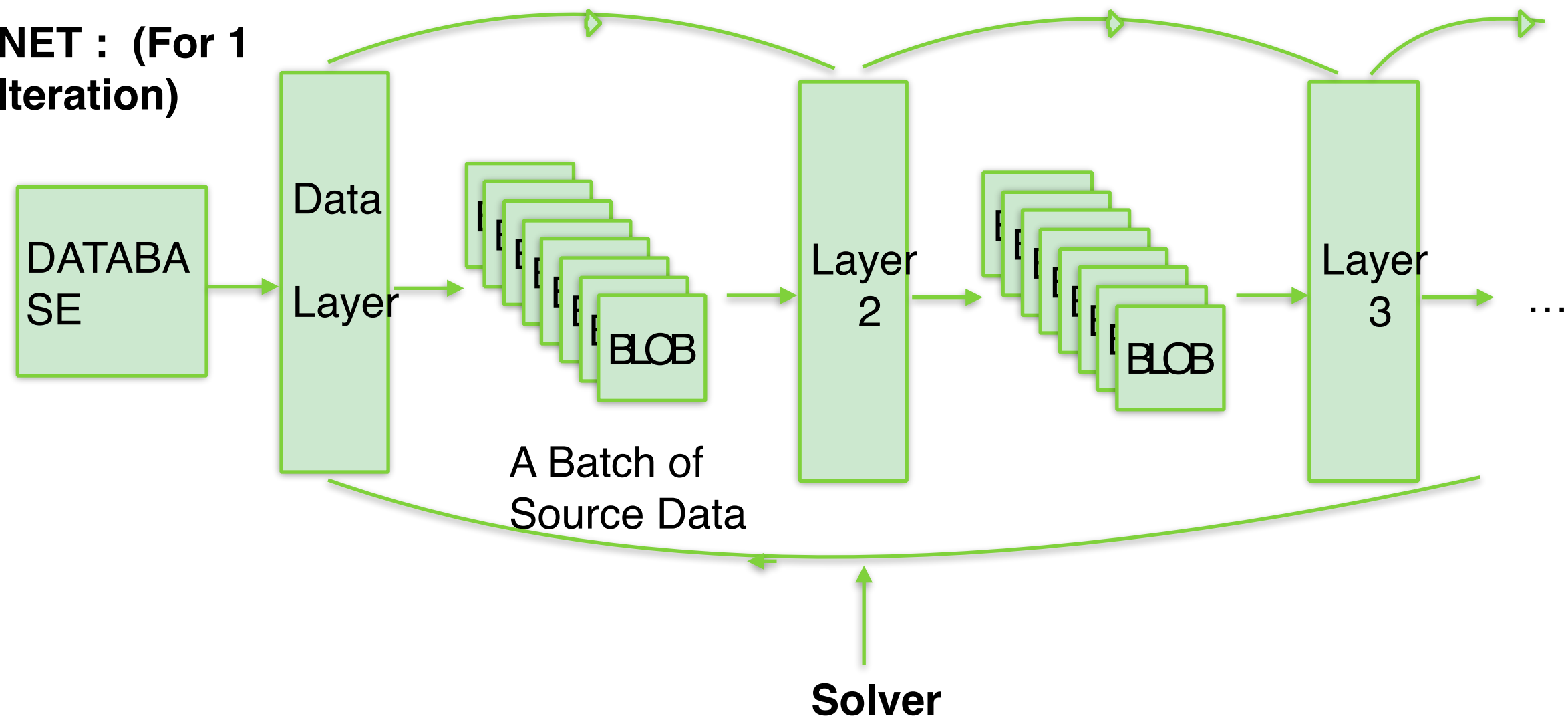
#输出为每个类别的概率值

```
layers {  
  name: "prob"  
  type: "Softmax"  
  bottom: "ip2"  
  top: "prob"  
}
```

3.Caffe的最优求解过程Solver

3. Caffe最优求解过程——Solver

NET : (For 1
Iteration)



3.1 Solver介绍

- 求解器**Solver**是什么？
 - Caffe的重中之重（核心）——Solver
 - 负责对模型优化，让损失函数(loss function)达到全局最小。
 - 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，实际上就是一种迭代的优化算法。
- 在每一次的迭代过程中，**solver**做了这几步工作：
 - 1、调用forward算法来计算最终的输出值，以及对应的loss
 - 2、调用backward算法来计算每层的梯度
 - 3、根据选用的solver方法，利用梯度进行参数更新

Solver的重点是最小化损失函数的全局最优问题，对于数据集 $D(\text{epoch})$ ，优化目标是在全数据集 D 上损失函数平均值：

$$L(W) = \frac{1}{|D|} \sum_i^{|D|} f_w(X^{(i)}) + \lambda r(W)$$

其中， $f_w(X^{(i)})$ 是在数据实例 $X^{(i)}$ 上的损失函数， $r(W)$ 为规整项， λ 为规整项的权重。数据集 D 一般都很大，工程上在每次迭代中使用这个目标函数的随机逼近，即小批量数据 $N \ll |D|$ 个数据实例：

$$L(W) \approx \frac{1}{|N|} \sum_i^{|N|} f_w(X^{(i)}) + \lambda r(W)$$

模型向前传播计算损失函数 f_w ，反向传播计算梯度 ∇f_w 。权值增量 ΔW 由求解器通过误差梯度 ∇f_w 、规整项梯度 $\nabla r(W)$ 以及其他与方法相关的项求解得到。

3.2 Solver参数配置

- Solver参数配置
 - 查看可配置参数: <https://github.com/BVLC/caffe/blob/master/src/caffe/proto/caffe.proto>
message SolverParameter {
...
}

```
# The train/test net protocol buffer definition
net: "examples/mnist/lenet_train_test.prototxt" //网络协议具体定义

# test_iter specifies how many forward passes the test should carry out.
# In the case of MNIST, we have test batch size 100 and 100 test iterations,
# covering the full 10,000 testing images.
test_iter: 100 //test迭代次数 如果batch_size =100,则100张图一批,训练100次,则可以覆盖10000张图的需求

# Carry out testing every 500 training iterations.
test_interval: 500 //训练迭代500次,测试一次

# The base learning rate, momentum and the weight decay of the network. //网络参数: 学习率, 动量, 权重的衰减
base_lr: 0.01
momentum: 0.9
weight_decay: 0.0005

# The learning rate policy //学习策略: 有固定学习率和每步递减学习率
lr_policy: "inv"
gamma: 0.0001
power: 0.75

# Display every 100 iterations //每迭代100次显示一次
display: 100

# The maximum number of iterations //最大迭代次数
max_iter: 10000

# snapshot intermediate results // 每5000次迭代存储一次数据, 路径前缀是<</span>span style="font-family: Arial, Helvetica, sans-
serif;">examples/mnist/lenet</>span>
snapshot: 5000
snapshot_prefix: "examples/mnist/lenet"

# solver mode: CPU or GPU //使用GPU或者CPU
solver_mode: GPU
```


- **net: "examples/mnist/lenet_train_test.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：

train_net:"examples/mnist/lenet_train_test.prototxt"

test_net: "examples/mnist/lenet_test_test.prototxt"

- **test_iter: 100**

mnist数据中测试样本总数为10000，一次性执行全部数据效率很低，因此我们将测试数据分成几个批次来执行，每个批次的数量就是batch_size。假设我们设置batch_size为100，则需要迭代100次才能将10000个数据全部执行完。因此test_iter设置为100。执行完一次全部数据，称之为一个epoch。

- **test_interval: 500**

在训练集中每迭代500次，在测试集进行一次测试。

- **base_lr: 0.01**
- **lr_policy: "inv"**
- **gamma: 0.0001**
- **power: 0.75**

这四个参数用于学习率的设置。只要是梯度下降法来求解优化，都会有一个学习率，也叫步长。base_lr用于设置基础学习率，在迭代的过程中，可以对基础学习率进行调整。怎么样进行调整，就是调整的策略，由lr_policy来设置。

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: 如果设置为step,则还需要设置一个stepsize, 返回 $\text{base_lr} * \gamma^{\lfloor \text{iter} / \text{stepsize} \rfloor}$, 其中iter表示当前的迭代次数

- exp: 返回 $\text{base_lr} * \gamma^{\text{iter}}$, iter为当前迭代次数

- inv: 如果设置为inv,还需要设置一个power, 返回 $\text{base_lr} * (1 + \gamma * \text{iter})^{-\text{power}}$

- multistep: 如果设置为multistep,则还需要设置一个stepvalue。这个参数和step很相似，step是均匀等间隔

- **weight_decay: 0.0005**
- **momentum : 0.9**
- **type: SGD**
 - Stochastic Gradient Descent (type: "SGD")
 - AdaDelta (type: "AdaDelta")
 - Adaptive Gradient (type: "AdaGrad")
 - Adam (type: "Adam")
 - Nesterov's Accelerated Gradient (type: "Nesterov")
 - RMSprop (type: "RMSProp")

3.3 Solver优化方法

- **Solver优化方法**
- Solver优化方法有六种，每一种的相关论文见“相关论文”里面有对每一种优化方法的论文。
- 具体介绍参考“Readme.txt”介绍。
- 重点讲解SGD随机梯度下降法

SGD随机梯度下降法

- **Stochastic gradient descent (SGD) 随机梯度下降法**

- SGD在通过负梯度 $\nabla L(W)$ 和上一次的权重更新值 V_t 的线性组合来更新 W ，迭代公式如下：

$$V_{t+1} = \mu V_t - \alpha \nabla L(W_t)$$

$$W_{t+1} = W_t + V_{t+1}$$

- 其中， α 是负梯度的学习率(base_lr)， μ 是上一次梯度值的权重（momentum），用来加权之前梯度方向对现在梯度下降方向的影响。这两个参数需要通过tuning来得到最好的结果，一般是根据经验设定的。如果你不知道如何设定这些参数，可以参考相关的论文。
- 在深度学习中使用SGD，比较好的初始化参数的策略是把学习率设为0.01左右（base_lr: 0.01），在训练的过程中，如果loss开始出现稳定水平时，对学习率乘以一个常数因子（gamma），这样的过程重复多次。
- 对于momentum，一般取值在0.5--0.99之间。通常设为0.9，momentum可以让使用SGD的深度学习方法更加稳定以及快速。

其他优化方法设置

- 对于RMSProp, AdaGrad, AdaDelta and Adam , 还可以设置delta参数。
- 对于Adam solver, 设置momentum2
- 对于RMSProp, 设置rms_decay

```
// numerical stability for RMSProp, AdaGrad and AdaDelta and Adam
optional float delta = 31 [default = 1e-8];
// parameters for the Adam solver
optional float momentum2 = 39 [default = 0.999];

// RMSProp decay value
// MeanSquare(t) = rms_decay*MeanSquare(t-1) + (1-rms_decay)*SquareGradient(t)
optional float rms_decay = 38 [default = 0.99];
```

4.Caffe的可视化工具

4.0 准备pycaffe环境

已经执行过的caffe安装命令：

```
# cmake -D CPU_ONLY=on -D CMAKE_INSTALL_PREFIX=/usr/local ..
```

```
# make all
```

```
# make install
```

首先执行完下面命令，进入\${CAFFE_ROOT}/python 才可以执行其中的python脚本：

```
sudo apt-get update
```

```
sudo apt-get install python-pip python-dev python-numpy
```

```
sudo apt-get install gfortran graphviz
```

```
sudo pip install --upgrade pip
```

```
sudo pip install -r ${CAFFE_ROOT}/python/requirements.txt
```

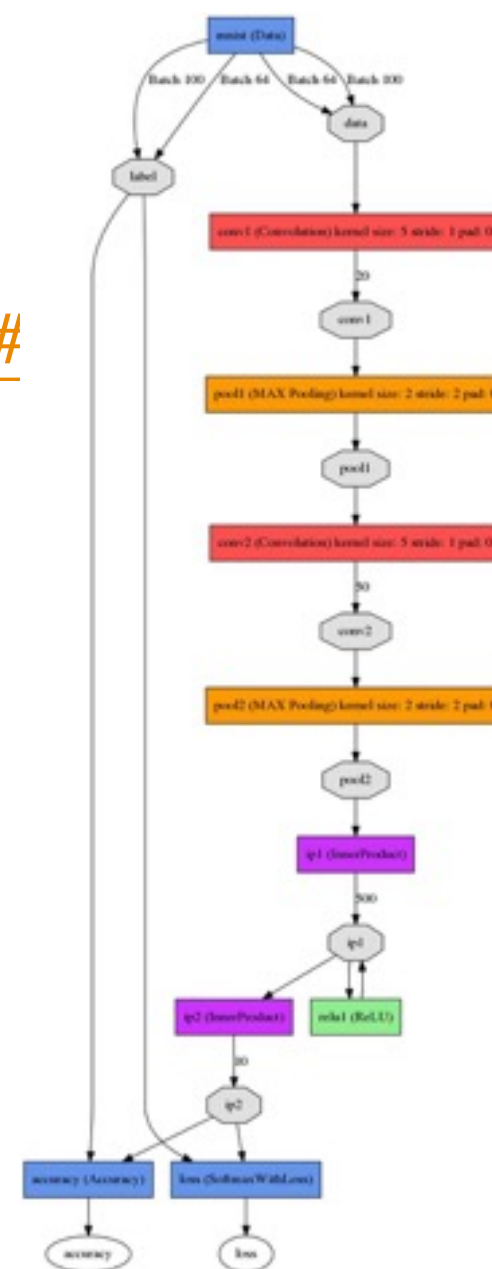
```
sudo pip install pydot
```

导入全局：

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

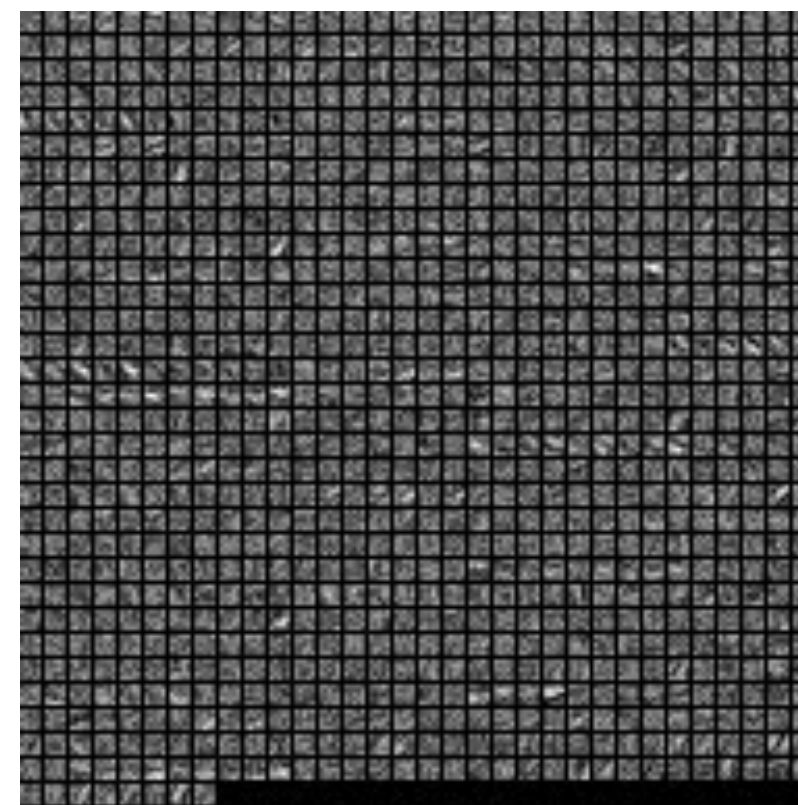
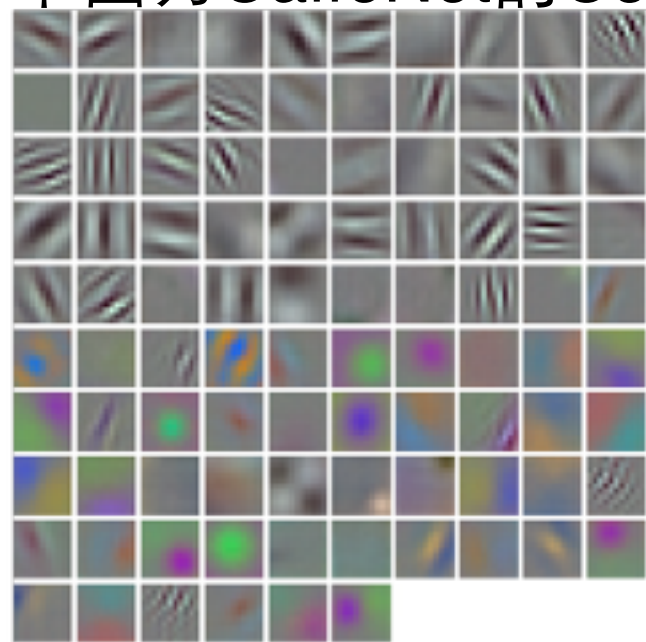
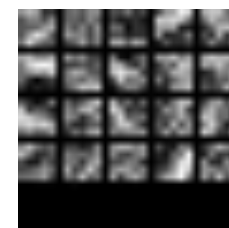

4.1 网络结构可视化工具

- 1. 代码: {caffe_root}/python/draw_net.py
- 2. 在线可视化工具 <http://ethereon.github.io/netscope/#>



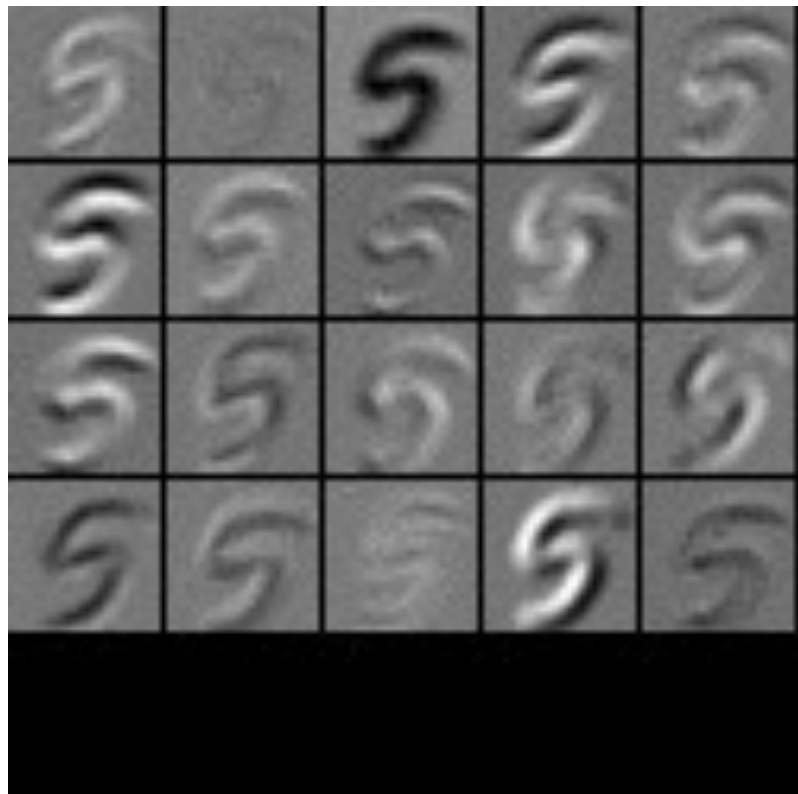
4.2 caffeemodel的可视化

- 课程代码: `test_extract_weights.py`
- 右图为训练mnist生成的LeNet Conv1和Conv2的权重值的可视化:
训练良好: 美观、光滑的滤波器
训练时间不够或者过拟合: 出现噪声图样
- 下图为CaffeNet的Conv1图:



4.3 特征图可视化

- 课程代码: `test_extract_data.py`
- * 代码注释讲解



4.4 可视化loss和accuracy曲线

- 课程代码: {caffe_root}/tools/extra/plot_training_log.py.example

Usage:

```
./plot_training_log.py chart_type[0-7] /where/to/save.png /path/to/first.log
```

...

Notes:

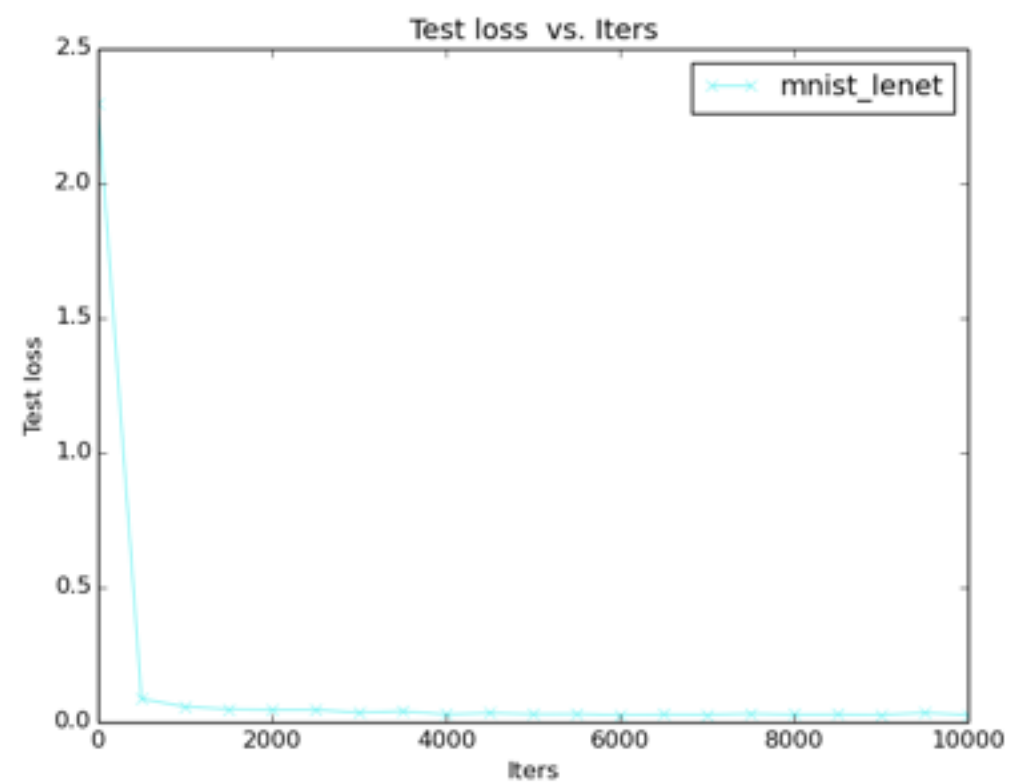
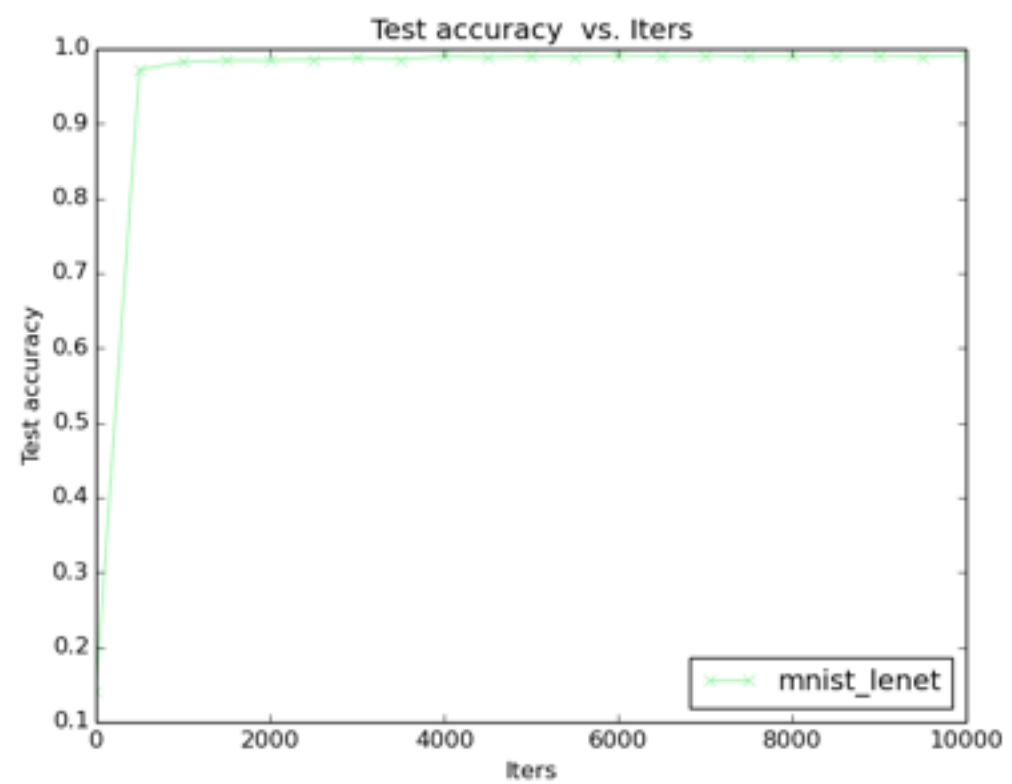
1. Supporting multiple logs.
2. Log file name must end with the lower-cased ".log".

Supported chart types:

- 0: Test accuracy vs. Iters
- 1: Test accuracy vs. Seconds
- 2: Test loss vs. Iters
- 3: Test loss vs. Seconds
- 4: Train learning rate vs. Iters
- 5: Train learning rate vs. Seconds
- 6: Train loss vs. Iters
- 7: Train loss vs. Seconds

aa &> log.log

caffe train -solver="" &> log.log



5. 使用训练好的模型

- 5.1 均值文件mean file
- 5.2 改写deploy文件
- 5.3 实操演示：
实现在新的数据上调用训练好的模型
- 5.4 使用fine turning微调网络

5.1 均值文件

- 将所有训练样本的均值保存为文件
- 图片减去均值后，再进行训练和测试，会提高速度和精度
- 运行方法：（使用Caffe工具）

```
compute_image_mean [train_lmdb] [mean.binaryproto]
```


5.2 改写deploy文件（以mnist为例）

- 1. 把数据层（Data Layer）和连接数据层的Layers去掉(即top:data的层)

```
layer {
  name: "mnist"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
  transform_param {
    scale: 0.00390625
  }
  data_param {
    source: "examples/mnist/mnist_train_lmdb"
    batch_size: 64
    backend: LMDB
  }
}
layer {
  name: "mnist"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TEST
  }
  transform_param {
    scale: 0.00390625
  }
  data_param {
    source: "examples/mnist/mnist_test_lmdb"
    batch_size: 100
    backend: LMDB
  }
}
```

- 2. 去掉输出层和连接输出层的Layers（即bottom:label）

```
...  
layer {  
  name: "accuracy"  
  type: "Accuracy"  
  bottom: "ip2"  
  bottom: "label"  
  top: "accuracy"  
  include {  
    phase: TEST  
  }  
}  
layer {  
  name: "loss"  
  type: "SoftmaxWithLoss"  
  bottom: "ip2"  
  bottom: "label"  
  top: "loss"  
}
```

- 3. 重新建立输入

```
input: "data"
```

```
input_shape {
```

```
  dim: 1 # batchsize,每次forward的时候输入的图片个数
```

```
  dim: 3 # number of colour channels - rgb
```

```
  dim: 28 # width
```

```
  dim: 28 # height
```

```
}
```

- 4.重新建立输出

```
layer {  
  name: "prob"  
  type: "Softmax"  
  bottom: "ip2"  
  top: "prob"  
}
```

* 修改后的mnist的deploy文件可以参考caffe/example/mnist/

lenet_train.prototxt

6.fine-turn微调网络

- 1. 准备新数据的数据库（如果需要用mean file,还要准备对应的新的mean file），具体方法和图片转换lmdb方式一样。
- 2. 调整网络层参数：
 - 将来训练的网络配置prototxt中的数据层source换成新数据的数据库。
 - 调整学习率，因为最后一层是重新学习，因此需要有更快的学习速率相比较其他层，因此我们将，weight和bias的学习速率加快。
- 3. 修改solver参数
 - 原来的数据是从原始数据开始训练的，因此一般来说学习速率、步长、迭代次数都比较大，fine turning微调时，因为数据量可能减少了，所以一般来说，test_iter,base_lr,stepsize都要变小一点，其他的策略可以保持不变。
- 4. 重新训练时，要指定之前的权值文件：
 - # caffe train --solver [新的solver文件] --weights [旧的caffemodel]