读书笔记 1 CIFAR-10 在 caffe 上进行训练与学习

2014.7.21 薛开宇

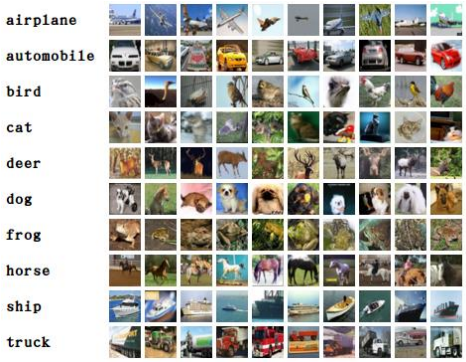
本次学习笔记作用，知道如何在 caffe 上训练与学习，如何看结果。

1.1 使用数据库：CIFAR-10

60000 张 32X32 彩色图像 10 类

50000 张训练

10000 张测试



1.2 准备

在终端运行以下指令：

cd $CAFFE_ROOT/data/cifar10

./get_cifar10.sh

cd $CAFFE_ROOT/examples/cifar10

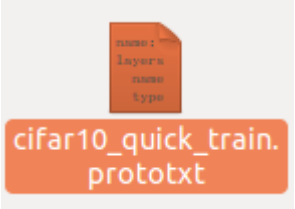
./create_cifar10.sh

其中 CAFFE_ROOT 是 caffe-master 在你机子的地址

运行之后，将会在 examples 中出现数据库文件./cifar10-leveldb 和数据库图像均值二进制文件./mean.binaryproto

1.3 模型

该 CNN 由卷积层，POOLing 层，非线性变换层，在顶端的局部对比归一化线性分类器组成。该模型的定义在 CAFFE_ROOT/examples/cifar10 directory’s cifar10_quick_train.prototxt 中，可以进行修改。其实后缀为 prototxt 很多都是用来修改配置的。



1.4 训练和测试

训练这个模型非常简单，当我们写好参数设置的文件 cifar10_quick_solver.prototxt 和定义的文件 cifar10_quick_train.prototxt 和 cifar10_quick_test.prototxt 后，运行 train_quick.sh 或者在终端输入下面的命令：

cd $CAFFE_ROOT/examples/cifar10

./train_quick.sh

即可，train_quick.sh 是一个简单的脚本，会把执行的信息显示出来，培训的工具是 train_net.bin,cifar10_quick_solver.prototxt 作为参数。

然后出现类似以下的信息：

I0317 21:52:48.945710 2008298256 net.cpp:74] Creating Layer conv1

I0317 21:52:48.945716 2008298256 net.cpp:84] conv1 <- data

I0317 21:52:48.945725 2008298256 net.cpp:110] conv1 -> conv1

I0317 21:52:49.298691 2008298256 net.cpp:125] Top shape: 100 32 32 32 (3276800)

I0317 21:52:49.298719 2008298256 net.cpp:151] conv1 needs backward computation.

这是搭建模型的相关信息

接着：

0317 21:52:49.309370 2008298256 net.cpp:166] Network initialization done.

I0317 21:52:49.309376 2008298256 net.cpp:167] Memory required for Data 23790808

I0317 21:52:49.309422 2008298256 solver.cpp:36] Solver scaffolding done.

I0317 21:52:49.309447 2008298256 solver.cpp:47] Solving CIFAR10_quick_train

之后，训练开始

I0317 21:53:12.179772 2008298256 solver.cpp:208] Iteration 100, lr = 0.001

I0317 21:53:12.185698 2008298256 solver.cpp:65] Iteration 100, loss = 1.73643

...

I0317 21:54:41.150030 2008298256 solver.cpp:87] Iteration 500, Testing net

I0317 21:54:47.129461 2008298256 solver.cpp:114] Test score #0: 0.5504

I0317 21:54:47.129500 2008298256 solver.cpp:114] Test score #1: 1.27805

其中每 100 次迭代次数显示一次训练时 lr(learning rate),和 loss（训练损失函数），每 500 次测试一次，输出 score 0（准确率）

和 score 1（测试损失函数）

当 5000 次迭代之后，正确率约为 75%，模型的参数存储在二进制 protobuf 格式在 cifar10_quick_iter_5000
然后，这个模型就可以用来运行在新数据上了。

1.5 其他
另外，更改 cifar*solver.prototxt 文件可以使用 CPU 训练，

solver mode: CPU or GPU
solver_mode: CPU

可以看看 CPU 和 GPU 训练的差别。

主要资料来源：caffe 官网教程

- CIFAR-10 tutorial
  Train and test Caffe on CIFAR-10 data.

读书笔记 2 用一个预训练模型提取特征

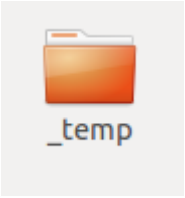
2014.7.21 薛开宇

本学习笔记的作用在于为后面打基础，没有什么实际的东西可以观测到，要可视化特征还要观看后面的教程。

2.1 制作一个数据库

先做一个临时文件夹存放东西。

mkdir examples/_temp



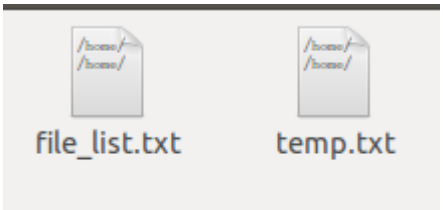
我们为两张在 images 文件夹的照片生成一个文件列表（默认为一张图片，cat1 是我随意加上去的）

find `pwd`/examples/images -type f -exec echo {} \; > examples/_temp/temp.txt

我们将使用 imagedatalayer 预计标签之后的每一个文件名，所以让我们添加一个 0 到每一行的末尾

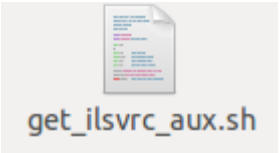
sed "s/$/ 0/" examples/_temp/temp.txt > examples/_temp/file_list.txt

这样，我们将得到两个文件 temp.txt 和 file_list.txt。



2.2.定义特征提取网络结构

在实践中，从一个数据集中减去均值图像对于提高分类准确性很重要，因此从 ILSVRC dataset 中下载均值图像数据库



data/ilsvrc12/get_ilsvrc_aux.sh

我们将使用其中的 data/ilsvrc212/imagenet_mean.binaryproto 去定义网络结构。

将定义结构的文件 cope 到我们的临时文件夹。

cp examples/feature_extraction/imagenet_val.prototxt examples/_temp
然后，我们进入 imagenet_val.prototxt 更改路径。更改其中$CAFFE_DIR 的地方
下图是我做的改动。

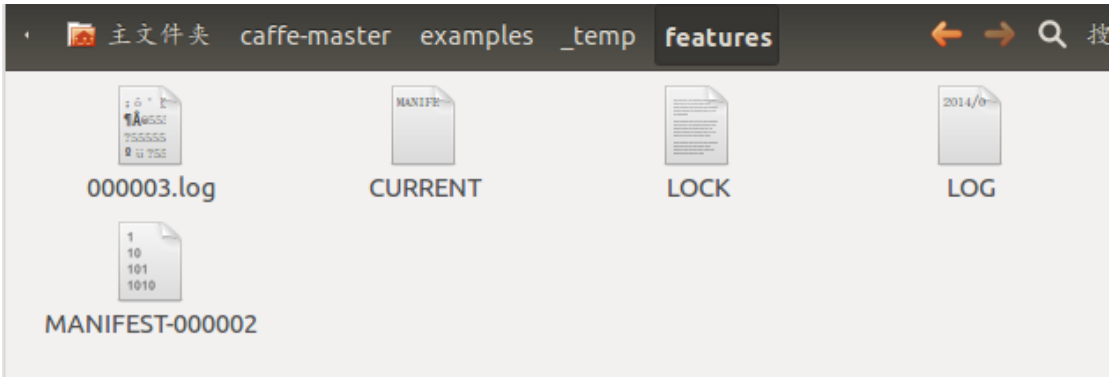
```
image_data_param {
  source: "/home/xuekaiyu/caffe-master/examples/_temp/file_list.txt"
  mean_file: "/home/xuekaiyu/caffe-master/data/ilsvrc12/imagenet_mean.binaryproto"
```

2.3.提取特征

执行指令

build/tools/extract_features.bin examples/imagenet/caffe_reference_imagenet_model examples/_temp/imagenet_val.prototxt fc7 examples/_temp/features 10

其中 fc7 是最高层的特征，我们也可以使用其他层提取，像 conv5 或 pool3
最后的参数是数据的批次
特征保存在 LevelDB examples/_temp/features ，可以运用到其他代码了。
这里可以发现，特征文件中 5 个文件和 cifar10 中的 leveldb 是相似的。
_temp 下 features



Cifar10 下特征数据库

因此，这是我们训练时组建模型时必要的文件。

2.4.注意

当存在 features 文件夹时将出现错误，这时需要移除该文件夹。

资料来源: caffe 官网教程

学习笔记 3　用自己的数据训练和测试“CaffeNet”

2014.7.22 薛开宇

本次学习笔记作用比较大，也是重点，知道如何在 caffe 上搭建自己的数据库。

3.1 数据准备

本学习笔记有点脱离了原文，原文是用 ImageNet1000 类的数据库，而因为电脑内存不足，只能自己模仿做一个小的数据库进行下去。

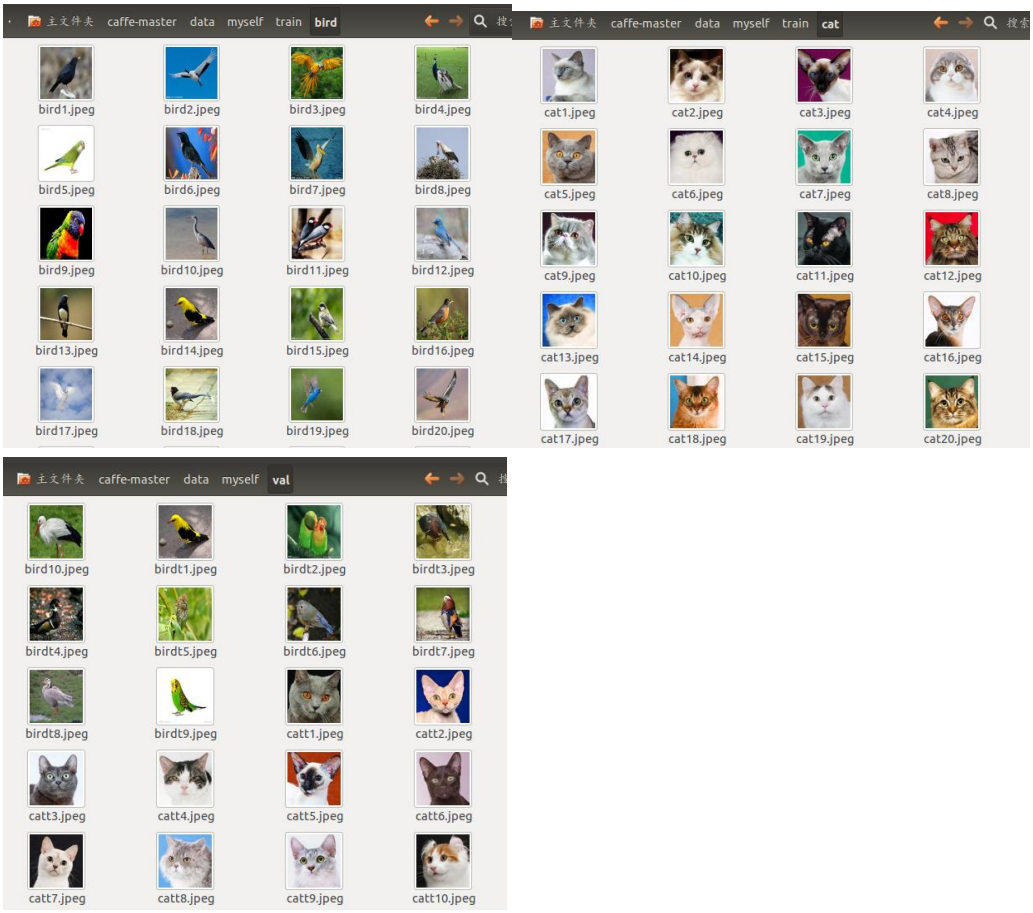
本来教程是假设已经下载了 ImageNet 训练数据和验证数据(非常大)，并以下面的格式存储在磁盘：

/path/to/imagenet/train/n01440764/n01440764_10026.JPEG
/path/to/imagenet/val/ILSVRC2012_val_00000001.JPEG

里面是各种的分类图。

因为实在太大，所以我们改为模仿搭建自己的数据库。

在 data 中新建文件夹 myself，本人在网上下载了训练猫的图片 50 张，测试猫 10 张，训练鸟的图片 50 张，测试鸟 10 张。如图所示：



如果坚持用 Imagenet 的话，我们还需要一些标签数据进行训练，用以下指令可以下载，如果不用，就可以不执行下面指令。

cd $CAFFE_ROOT/data/ilsvrc12/
./get_ilsvrc_aux.sh

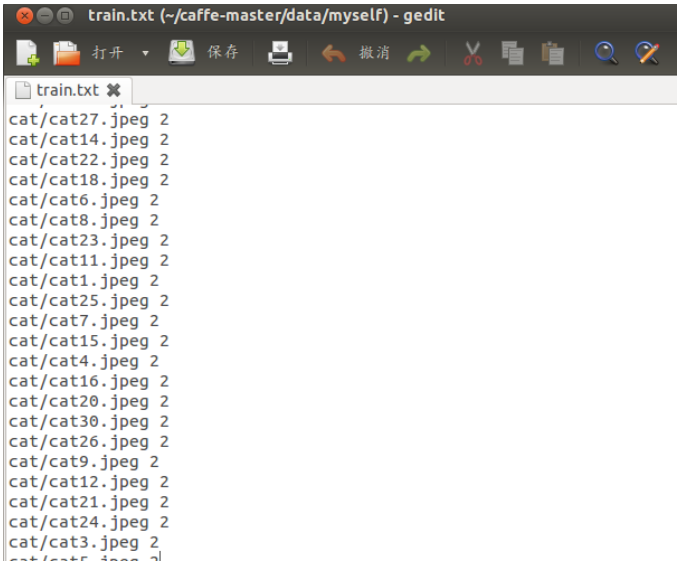
培训和测试的输入是用 train.txt 和 val.txt 描述的，这些文档列出所有文件和他们的标签。注意，我们分类的名字是 ASCII 码的顺序，即 0-999，对应的分类名和数字的映射在 synset_words.txt（自己写）中。

运行以下指令：

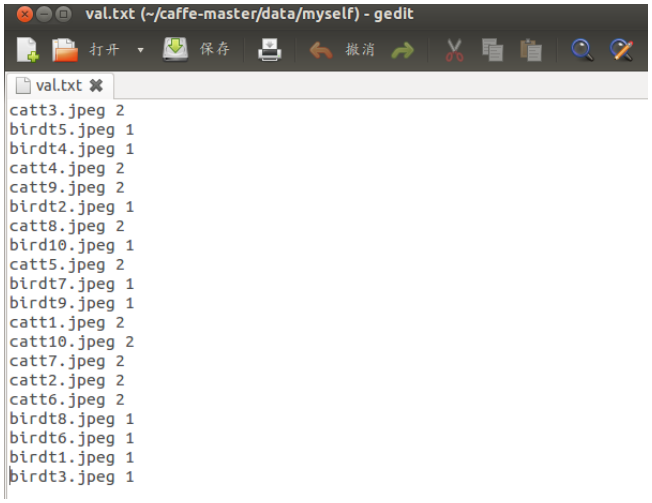
find -name *.jpeg |cut -d '/' -f2-3 > train.txt

注意路径

然后，因为样本数比较少，可以自行手动做分类标签。在 train.txt 的每个照片后用 0-999 分类。当样本过多，就自己编写指令批量处理。



同理，获得 val.txt



Test.txt 和 val.txt 一样，不过后面不能标签，全部设置成 0。

我们还需要把图片的大小变成 256X256，但在一个集群环境，可以不明确的进行，使用 Mapreduce 就可以了，像杨青就是这样做的。如果我们希望更简单，用下面的命令：

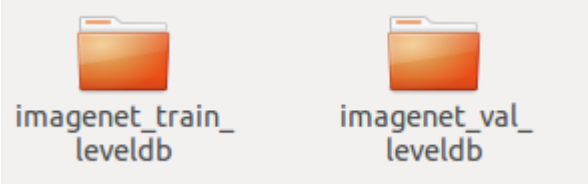
```
for name in /path/to/imagenet/val/*.JPEG; do
    convert -resize 256x256\! $name $name
done
```

我做成了 sh，方便以后使用。

然后在 caffe-master 创建 myself 文件夹，然后将 imagenet 的 create_imagenet.sh. 复制到该文件夹下进行修改，进行训练和测试路径的设置，运行该 sh.

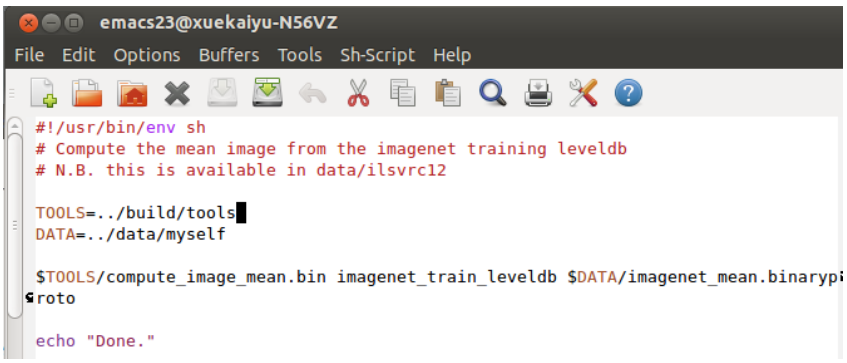
```
DATA=../../data/myself

echo "Creating leveldb..."

GLOG_logtostderr=1 $TOOLS/convert_imageset.bin \
    /home/xuekaiyu/caffe-master/data/myself/train/ \
    $DATA/train.txt \
    imagenet_train_leveldb 1

GLOG_logtostderr=1 $TOOLS/convert_imageset.bin \
    /home/xuekaiyu/caffe-master/data/myself/val/ \
    $DATA/val.txt \
    imagenet_val_leveldb 1
```

最后得到



3.2 计算图像均值

模型需要我们从每张图片减去均值，所以我们必须获得训练的均值，用 tools/compute_image_mean.cpp 实现，这个 cpp 是一个很好的例子去熟悉如何操作多个组建，例如协议的缓冲区，leveldbs，登录等。我们可以直接复制 imagenet 的 ./make_imagenet_mean.加以修改应用就行,注意路径。



3.3 网络的定义

从 imagenet 中复制修改 imagenet_train.prototxt，注意修改数据层的路径。

source: "ilvsrc12_train_leveldb"
mean_file: "../../data/ilsvrc12/imagenet_mean.binaryproto"

我改成

```
data_param {
  source: "imagenet_train_leveldb"
  mean_file: "../data/myself/imagenet_mean.binaryproto"
  batch_size: 256
  crop_size: 227
  mirror: true
}
```

同理，复制修改 imagenet_val.prototxt.
改成

```
data_param {
  source: "imagenet_val_leveldb"
  mean_file: "../data/myself/imagenet_mean.binaryproto"
  batch_size: 50
  crop_size: 227
  mirror: false
}
```

如果你细心观察 imagenet_train.prototxt and imagenet_val.prototxt，你会发现他们除了数据来源不同和最后一层不同，其他基本相同。在训练中，我们用一个 softmax——loss 层计算损失函数和初始化反向传播，而在验证，我们使用精度层检测我们的精度。

我们还有一个运行的协议 imagenet_train.prototxt，复制过来，从里面可以观察到，我们将运行 256 批次，迭代 4500000 次（90 期），每 1000 次迭代，我们测试学习网络验证数据，我们设置初始的学习率为 0.01，每 100000（20 期）次迭代减少学习率，显示一次信息，训练的 weight_decay 为 0.0005，每 10000 次迭代，我们显示一下当前状态。

以上是教程的，实际上，以上需要耗费很长时间，因此，我们稍微改一下

test_iter: 1000 是指测试的批次，我们就 10 张照片，设置 10 就可以了。

test_interval: 1000 是指每 1000 次迭代测试一次，我们改成 500 次测试一次。

base_lr: 0.01 是基础学习率，因为数据量小，0.01 就会下降太快了，因此改成 0.001

lr_policy: "step"学习率变化

gamma: 0.1 学习率变化的比率

stepsize: 100000 每 100000 次迭代减少学习率

display: 20 每 20 层显示一次

max_iter: 450000 最大迭代次数，

momentum: 0.9 学习的参数，不用变

weight_decay: 0.0005 学习的参数，不用变

snapshot: 10000 每迭代 10000 次显示状态，这里改为 2000 次

solver_mode: GPU 末尾加一行，代表用 GPU 进行

3.4 训练

把./train_imagenet.sh 复制过来运行，然后就像学习笔记 1 一样训练了，当然，只有两类，正确率还是相当的高。

```
xuekaiyu@xuekaiyu-N56VZ: ~/caffe-master/myself
I0716 20:36:26.403406 22176 solver.cpp:142] Test score #1: 0.295262
I0716 20:38:35.474187 22176 solver.cpp:237] Iteration 100, lr = 0.001
I0716 20:38:35.501572 22176 solver.cpp:87] Iteration 100, loss = 0.116571
I0716 20:38:35.501607 22176 solver.cpp:106] Iteration 100, Testing net
I0716 20:38:39.976903 22176 solver.cpp:142] Test score #0: 0.9
I0716 20:38:39.976945 22176 solver.cpp:142] Test score #1: 0.276086
I0716 20:40:49.117224 22176 solver.cpp:106] Iteration 120, Testing net
I0716 20:40:53.598922 22176 solver.cpp:142] Test score #0: 0.85
I0716 20:40:53.598964 22176 solver.cpp:142] Test score #1: 0.225632
I0716 20:43:02.717257 22176 solver.cpp:106] Iteration 140, Testing net
I0716 20:43:07.201568 22176 solver.cpp:142] Test score #0: 0.85
I0716 20:43:07.201611 22176 solver.cpp:142] Test score #1: 0.25258
I0716 20:45:16.278803 22176 solver.cpp:106] Iteration 160, Testing net
I0716 20:45:20.745877 22176 solver.cpp:142] Test score #0: 0.9
I0716 20:45:20.745918 22176 solver.cpp:142] Test score #1: 0.506309
I0716 20:47:29.819383 22176 solver.cpp:106] Iteration 180, Testing net
I0716 20:47:34.291712 22176 solver.cpp:142] Test score #0: 0.9
I0716 20:47:34.291764 22176 solver.cpp:142] Test score #1: 0.465867
I0716 20:49:43.363847 22176 solver.cpp:237] Iteration 200, lr = 0.001
I0716 20:49:43.390959 22176 solver.cpp:87] Iteration 200, loss = 0.0109258
I0716 20:49:43.391000 22176 solver.cpp:106] Iteration 200, Testing net
I0716 20:49:47.858572 22176 solver.cpp:142] Test score #0: 0.9
I0716 20:49:47.858611 22176 solver.cpp:142] Test score #1: 0.291695
```

3.5 恢复数据

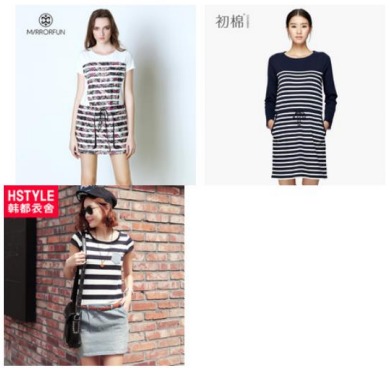
我们用指令./resume_training.sh 即可。

3.6 网上一些有趣的做法：

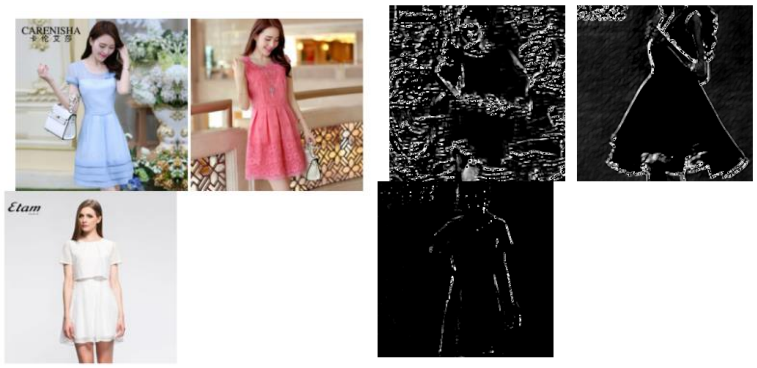
http://www.tuicool.com/articles/MZN3IvU

用 caffe 自己做衣服识别，两类，条纹衣服和纯色衣服。

条纹衣服的原图和特征图



纯色衣服和特征图：



看条纹衣服的特征图比较有意思，把“条纹”特征给抽取出来了。也许这就是神经网络神奇的地方，在没有人的干扰的情况下，竟然能学习出来“条纹”特征。

3.7 该文章总结的做样本库的注意事项：

1 数据集要保证质量。曾经玩过一字领和 polo 领的分类，刚开始效果很差，后来发现有一些“错误”的标签，于是把那些样本给去掉。效果好了很多。

2 learning rate 要调整。有一次训练了很久，准确率几乎不变，于是我减少了 lr，发现好了很多。这点深有同感，可以尝试把学习率调回 0.01，保证小数据不能超过 50 的识别。

3 均值化图片。实践证明，均值化后再训练收敛速度更快，准确率更高。

3.8 我们能做什么？

1 对于哪些数据集，深度学习比较适合？

2 对于效果差的数据集，如何能提高准确率？

主要资料来源：caffe 官网的教程

- ImageNet tutorial

  Train and test "CaffeNet" on ImageNet challenge data.

读书笔记 4 学习搭建自己的网络 MNIST 在 caffe 上进行训练与学习

2014.7.22 薛开宇

本次学习笔记作用也是比较重要，知道如何在 caffe 上搭建自己的训练网络。

1.1 准备数据库：MNIST 手写字体库

运行以下指令下载：

cd $CAFFE_ROOT/data/mnist

./get_mnist.sh

cd $CAFFE_ROOT/examples/mnist

./create_mnist.sh

运行之后，会有 mnist-train-leveldb 和 mnist-test-leveldb.文件夹

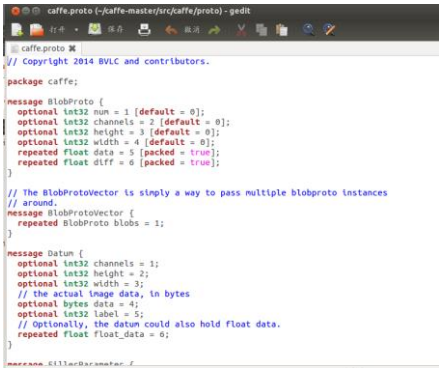
1.2 训练模型的解释

在我们训练之前，我们解释一下将会发生什么，我们将使用 LeNet 的训练网络，这是一个被认为在数字分类任务中运行很好的网络，我们会运用一个和原始版本稍微不同的版本，这次用 ReLU（线性纠正函数）取代 sigmoid 函数去激活神经元

这次设计包含 CNNs 的精髓，即像训练 imageNet 那样也是运用较大的训练模型。一般来说，由一层卷基层后跟着池层，然后再是卷基层，最后两个全连接层开诶死于传统的多层感知器，我们在 CAFFE_ROOT/data/lenet.prototxt.中已经定义了层

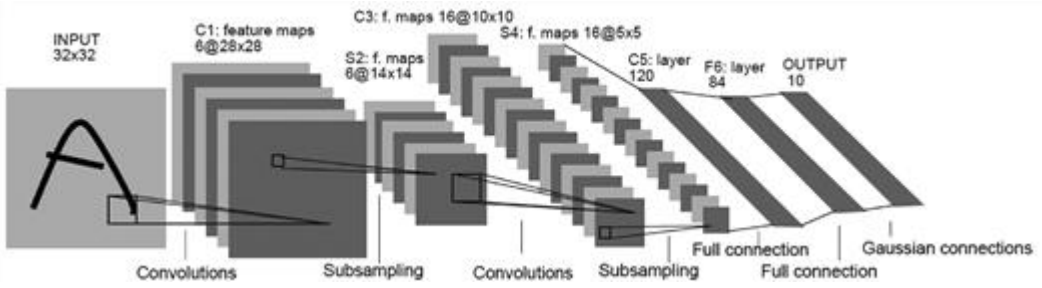
1.3 定义 MNIST 训练网络

这部分介绍如何使用 lenet_train.prototxt，我们假设您已经熟悉 Google Protobuf（主要作用是把某种数据结构的信息，以某种格式保存起来。主要用于数据存储、传输协议格式等场合。），同时假设已经阅读了 caffe 中的 protobuf 定义（可以在 src/caffe/proto/caffe.proto.找到）



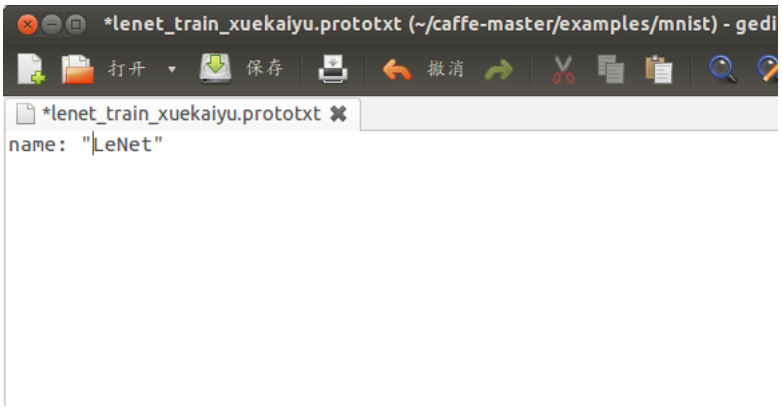
这个文档可以在我们建立自己的网络时，方便我们查到我们需要的格式。

我们将尝试写一个 caffe 网络参数 protubuf，先观察一下传统的网络,但实际上 caffe 上的对这个网络有点改变，例如 C1 层是 20 个 feature maps，第 C3 层是 50 个，C5 层是 500 个，没有 F6 层，直接是 OUTPUT 层。



首先是命名：

name: "LeNet"



写入数据层，首先，我们需要先观察一下数据，之后，将定义一个数据层：

注释如下：

```
layers {
# 输入层的名字为 mnist
  name: "mnist"
# 输入的类型为 DATA
  type: DATA
# 数据的参数
  data_param {
# 从 mnist-train-leveldb 中读入数据
    source: "mnist-train-leveldb"
# 我们的批次大小为 64，即为了提高性能，一次处理 64 条数据
    batch_size: 64
# 我们需要把输入像素灰度归一到【0，1），将 1 处以 256，得到 0.00390625。
    scale: 0.00390625
  }
# 然后这层后面连接 data 和 label Blob 空间
  top: "data"
  top: "label"
}
```

然后是卷积层：

```
layers {
# 卷积层名字为 conv1
  name: "conv1"
# 类型为卷积
  type: CONVOLUTION
# 这层前面使用 data，后面生成 conv1  的 Blob 空间
  bottom: "data"
  top: "conv1"
# 学习率调整的参数，我们设置权重学习率和运行中求解器给出的学习率一样，同时是偏置
#学习率的两倍，
```

```
  blobs_lr: 1
  blobs_lr: 2
# 卷积层的参数
  convolution_param {
# 输出单元数 20
    num_output: 20
# 卷积核的大小为 5*5
    kernel_size: 5
# 步长为 1
    stride: 1
# 网络允许我们用随机值初始化权重和偏置值。
    weight_filler {
# 使用 xavier 算法自动确定基于输入和输出神经元数量的初始规模
      type: "xavier"
    }
    bias_filler {
#  偏置值初始化为常数，默认为 0
      type: "constant"
    }
  }
}
```

然后是 pooling 层：

```
layers {
#pooling 层名字叫 pool1
  name: "pool1"
#类型是 pooling
  type: POOLING
#这层前面使用 conv1,后面生层 pool1 的 Blob 空间
  bottom: "conv1"
  top: "pool1"
#pooling 层的参数
  pooling_param {
#pooling 的方式是 MAX
    pool: MAX
#pooling 的核是 2X2
    kernel_size: 2
#pooling 的步长是 2
    stride: 2
  }
}
```

然后是第二个卷积层，和前面没什么不同，不过这里用了 50 个卷积核，前面是 20 个。

```
layers {
# 卷积层的名字是 conv2
  name: "conv2"
# 类型是卷积
  type: CONVOLUTION
# 这层前面使用 pool1,后面生层 conv2 的 Blob 空间
  bottom: "pool1"
  top: "conv2"
  blobs_lr: 1
  blobs_lr: 2
  convolution_param {
# 输出频道数 50
    num_output: 50
    kernel_size: 5
    stride: 1
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
```

然后是第二个 pooling 层，和前面的没什么不同。

```
layers {
  name: "pool2"
  type: POOLING
  bottom: "conv2"
  top: "pool2"
  pooling_param {
    pool: MAX
    kernel_size: 2
    stride: 2
  }
}
```

然后是全连接层，在某些特殊原因下看成卷积层，因此结构差不多。

```
layers {
# 全连接层的名字
  name: "ip1"
# 类型是全连接层
  type: INNER_PRODUCT
  blobs_lr: 1.
  blobs_lr: 2.

# 全连接层的参数
 inner_product_param {
#输出 500 个节点，据说在一定范围内这里节点越多，正确率越高。
      num_output: 500
      weight_filler {
        type: "xavier"
      }
      bias_filler {
        type: "constant"
      }
  }
  bottom: "pool2"
  top: "ip1"
}
```

然后是 ReLU 层，由于是元素级的操作，我们可以现场激活来节省内存，

```
layers {
  name: "relu1"
#类型是 RELU
  type: RELU
  bottom: "ip1"
  top: "ip1"
}
```

该层后，我们编写另外一个全连接层：

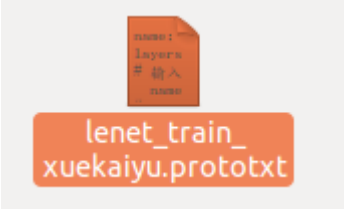
```
layers {
  name: "ip2"
  type: INNER_PRODUCT
  blobs_lr: 1.
  blobs_lr: 2.
  inner_product_param {
# 输出十个单元
      num_output: 10
      weight_filler {
        type: "xavier"
```

```
    }
    bias_filler {
      type: "constant"
    }
  }
  bottom: "ip1"
  top: "ip2"
}
```

然后是 LOSS 层，该 softmax_loss 层同时实现了 SOFTMAX 和多项 Logistic 损失，即节省了时间，同时提高了数据稳定性。它需要两块，第一块是预测，第二块是数据层提供的标签。它不产生任何输出，它做的是去计算损失函数值，在 BP 算法运行的时候使用，启动相对于 ip2 的梯度。

```
layers {
  name: "loss"
  type: SOFTMAX_LOSS
  bottom: "ip2"
  bottom: "label"
}
```

然后就完成了自己编写的网络了



同时定义 MNIST Solver

```
# 定义训练数据来源
train_net: "lenet_train.prototxt"
# 定义检测数据来源
test_net: "lenet_test.prototxt"
# 这里是训练的批次，设为 100，迭代次数 100 次，这样，就覆盖了 10000 张（100*100）
# 个测试图片。
test_iter: 100
# 每迭代次数 500 次测试一次
test_interval: 500
# 学习率，动量，权重的递减
base_lr: 0.01
momentum: 0.9
weight_decay: 0.0005
# 学习政策 inv，注意的是，cifar10 类用固定学习率，imagenet 用每步递减学习率。
lr_policy: "inv"
gamma: 0.0001
power: 0.75
```

每迭代 100 次显示一次
display: 100
最大迭代次数 10000 次
max_iter: 10000
每 5000 次迭代存储一次数据到电脑，名字是 lenet。
snapshot: 5000
snapshot_prefix: "lenet"
0 是用 CPU 训练，1 是用 GPU 训练。
solver_mode: 1

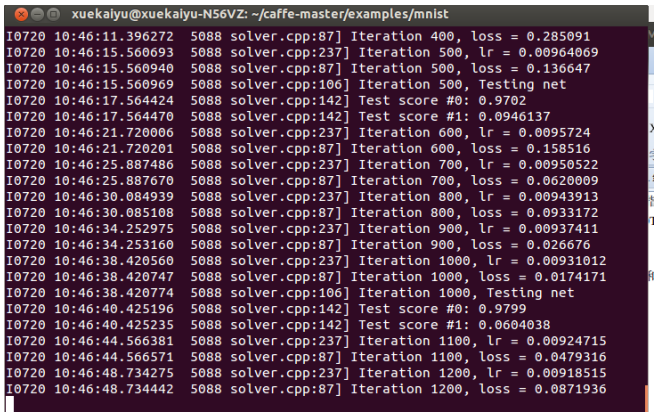
1.4 训练和测试该模型

注意更改好路径，这里可以尝试用自己写的网络训练。

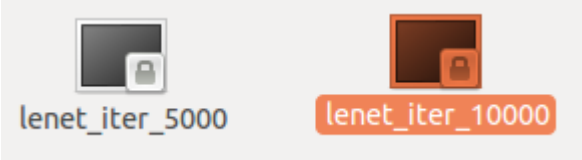
```
# The training protocol buffer definition
train_net: "lenet_train_xuekaiyu.prototxt"
# The testing protocol buffer definition
test_net: "lenet_test.prototxt"
```

然后在终端执行指令：
cd $CAFFE_ROOT/examples/mnist
./train_lenet.sh

之后的事就和其他与学习笔记 1 差不多了，可以参考学习笔记 1。



下图就是迭代到 5000 和迭代到 10000 的模型，至于怎么用，在后面的学习笔记将会提及。



主要资料来源：http://caffe.berkeleyvision.org/gathered/examples/mnist.html

读书笔记 5 逐层可视化图像特征

2014.7.23 薛开宇

本篇比较重要，讲述了如何可视化每层的特征。其中有些涉及分类的细节，该部分具体看读书笔记 6。

注意，本笔记的代码由原文复制过来，再复制过去 ipython 可能会有些格式上的错误，建议使用原文（本文最后）的代码进行复制。

1.1 开始 Ipython notebook

配置 python

文件夹 python 的 requirements.txt 列举了 Caffe 依赖的 python 库，可以看一下教程具体配置网上有教程 http://www.cnblogs.com/alfredtofu/p/3577241.html

我也是手动安装的

Cython>=0.19.2
h5py>=2.2.0
ipython>=1.1.0
leveldb>=0.191
matplotlib>=1.3.1
networkx>=1.8.1
nose>=1.3.0
numpy>=1.7.1
pandas>=0.12.0
protobuf>=2.5.0
python-gflags>=2.0
scikit-image>=0.9.3
scikit-learn>=0.14.1
scipy>=0.13.2

以 Cython 为例，手动的话用 sudo pip install Cython 即可，然后在 pip 的过程中有什么问题就相应解决。

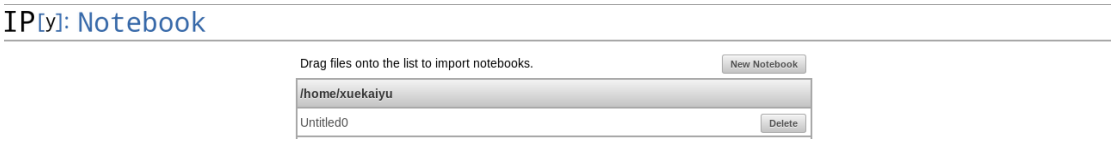
同时注意的是，我们下面的语句都是用 ipython，ipython 中用 ！表示执行 shell 命令，用 $ 将 python 的变量转化成 shell 变量。通过这种两个符号，我们就可以做到和 shell 命令之间的交互，可以非常方便地做许多复杂的工作。

Ipython 可以在终端执行，但为了方便，我们可以用 Ipython notebook。notebook 是 web based IPython 封装，但是可以展现富文本，使得整个工作可以以笔记的形式展现、存储，对于交互编程、学习非常方便。Ipython notebook 的安装配置可以在网上找到教程。

http://blog.csdn.net/awayyao/article/details/17785321

配置好 python，ipython 和 python notebook，直接在终端输入：

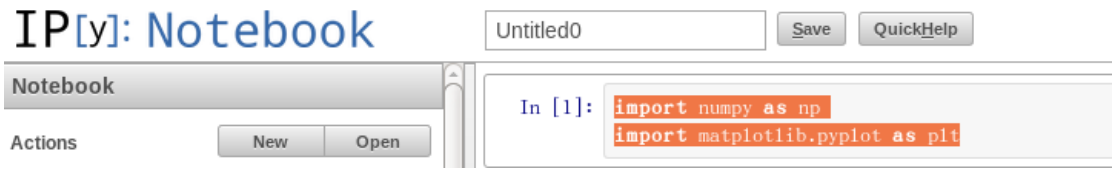
ipython notebook



1.2 开始进行

先输入一段代码：

```
In [1]:  import numpy as np
         import matplotlib.pyplot as plt
```



按 shift+enter 执行。

代码的意思是调入 numpy 子程序，调入后名字为 np，调入 matplotlib.pyplot 子程序（用作画图），调入后名字为 plt。

然后是 （以下是我的路径）：

```
caffe_root = '/home/xuekaiyu/caffe-master/'
```

这里注意路径一定要设置正确，记得前后可能都有“/”，路径的使用是{caffe_root}/examples，记得 caffe-master 中的 python 文件夹需要包括 caffe 文件夹。

接着是

```
import sys
sys.path.insert(0, caffe_root + 'python')
import caffe
```

把 ipython 的路径改到我们之前指定的地方，以便可以调入 caffe 模块，如果不改路径，import 这个指令只会在当前目录查找，是找不到 caffe 的。

接着是：

```
plt.rcParams['figure.figsize'] = (10, 10)
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'
```

代码的意思是，显示的图表大小为 10，图形的插值是以最近为原则，图像颜色是灰色

```
net = caffe.Classifier(caffe_root + 'examples/imagenet/imagenet_deploy.prototxt',
        caffe_root + 'examples/imagenet/caffe_reference_imagenet_model')
```

预先下载好模型，模型的大小 232.57MB,可以用指令：examples/imagenet/get_caffe_reference_imagenet_model.sh. 下载。调用分类网络进行分类，记得设置好路径，启用后，终端将用该模型进行分类。

```
I0722 15:41:23.833319  6558 net.cpp:99] relu7 -> fc7 (in-place)
I0722 15:41:23.833329  6558 net.cpp:126] Top shape: 10 4096 1 1 (40960)
I0722 15:41:23.833338  6558 net.cpp:152] relu7 needs backward computation.
I0722 15:41:23.833364  6558 net.cpp:75] Creating Layer drop7
I0722 15:41:23.833375  6558 net.cpp:85] drop7 <- fc7
I0722 15:41:23.833386  6558 net.cpp:99] drop7 -> fc7 (in-place)
I0722 15:41:23.833398  6558 net.cpp:126] Top shape: 10 4096 1 1 (40960)
I0722 15:41:23.833410  6558 net.cpp:152] drop7 needs backward computation.
I0722 15:41:23.833423  6558 net.cpp:75] Creating Layer fc8
I0722 15:41:23.833433  6558 net.cpp:85] fc8 <- fc7
I0722 15:41:23.833444  6558 net.cpp:111] fc8 -> fc8
I0722 15:41:23.841811  6558 net.cpp:126] Top shape: 10 1000 1 1 (10000)
I0722 15:41:23.841853  6558 net.cpp:152] fc8 needs backward computation.
I0722 15:41:23.841869  6558 net.cpp:75] Creating Layer prob
I0722 15:41:23.841897  6558 net.cpp:85] prob <- fc8
I0722 15:41:23.841910  6558 net.cpp:111] prob -> prob
I0722 15:41:23.841934  6558 net.cpp:126] Top shape: 10 1000 1 1 (10000)
I0722 15:41:23.841945  6558 net.cpp:152] prob needs backward computation.
I0722 15:41:23.841958  6558 net.cpp:163] This network produces output prob
I0722 15:41:23.841984  6558 net.cpp:181] Collecting Learning Rate and Weight Dec
ay.
I0722 15:41:23.842002  6558 net.cpp:174] Network initialization done.
I0722 15:41:23.842013  6558 net.cpp:175] Memory required for Data 42022840
```

接着：输入以下指令：

```
net.set_phase_test()
net.set_mode_cpu()
```

再接着，是以下指令：

```
net.set_mean('data', caffe_root + 'python/caffe/imagenet/ilsvrc_2012_mean.npy')  # ImageNet 的均值
net.set_channel_swap('data', (2,1,0))  # 因为参考模型本来频道是 BGR，所以要将 RGB 转换
net.set_input_scale('data', 255)  #参考模型运行在【0，255】的灰度，而不是【0，1】。
```

接着是：

```
scores = net.predict([caffe.io.load_image(caffe_root + 'examples/images/cat.jpg')])
```

出现 in classifier.py line,TpyeError: squeeze takes no keyword arguments. 纠结了半天之后，发现是 numpy 版本过低导致的，ubuntu 自带的 numpy 版本是 1.6.1，更新到 1.7 以上。

用指令：sudo pip install numpy --upgrade

然后输入：

```
[(k, v.data.shape) for k, v in net.blobs.items()]
```

显示出各层的参数和形状，第一个是批次，第二个 feature map 数目，第三和第四是每个神经元中图片的长和宽，可以看出，输入是 227*227 的图片，三个频道，卷积是 32 个卷积核卷三个频道，因此有 96 个 feature map

Out[4]:

```
[('data', (10, 3, 227, 227)),
 ('conv1', (10, 96, 55, 55)),
 ('pool1', (10, 96, 27, 27)),
 ('norm1', (10, 96, 27, 27)),
 ('conv2', (10, 256, 27, 27)),
 ('pool2', (10, 256, 13, 13)),
 ('norm2', (10, 256, 13, 13)),
 ('conv3', (10, 384, 13, 13)),
 ('conv4', (10, 384, 13, 13)),
 ('conv5', (10, 256, 13, 13)),
 ('pool5', (10, 256, 6, 6)),
 ('fc6', (10, 4096, 1, 1)),
 ('fc7', (10, 4096, 1, 1)),
 ('fc8', (10, 1000, 1, 1)),
 ('prob', (10, 1000, 1, 1))]
```

再输入：

```
[(k, v[0].data.shape) for k, v in net.params.items()]
predictions = out[self.outputs[0]].squeeze(axis=(2,3))
```

输出：一些网络的参数

```
[('conv1', (96, 3, 11, 11)),
 ('conv2', (256, 48, 5, 5)),
 ('conv3', (384, 256, 3, 3)),
 ('conv4', (384, 192, 3, 3)),
 ('conv5', (256, 192, 3, 3)),
 ('fc6', (1, 1, 4096, 9216)),
 ('fc7', (1, 1, 4096, 4096)),
 ('fc8', (1, 1, 1000, 4096))]
```

再接着是一些显示图片的指令：

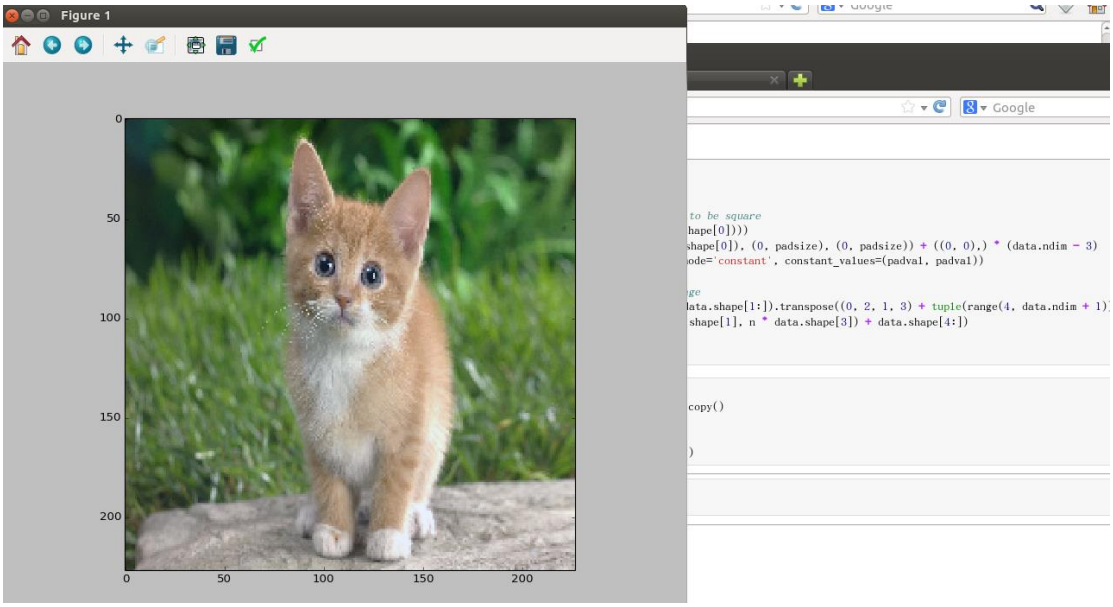
```
    # our network takes BGR images, so we need to switch color channels
    #  网络需要的是 BGR 格式图片，所以转换颜色频道
def showimage(im):
    if im.ndim == 3:
    m = im[:, :, ::-1]
  plt.imshow(im)

    # take an array of shape (n, height, width) or (n, height, width, channels)
    #  用一个格式是（数量，高，宽）或（数量，高，宽，频道）的阵列
    #    and visualize each (height, width) thing in a grid of size approx. sqrt(n) by sqrt(n)
    #  每个可视化的都是在一个由一个个网格组成
```

```
def vis_square(data, padsize=1, padval=0):
        data -= data.min()
        data /= data.max()
            # force the number of filters to be square
n = int(np.ceil(np.sqrt(data.shape[0])))
padding = ((0, n ** 2 - data.shape[0]), (0, padsize), (0, padsize)) + ((0, 0),) * (data.ndim - 3)
data = np.pad(data, padding, mode='constant', constant_values=(padval, padval))

            # 对图像使用滤波器
data = data.reshape((n, n) + data.shape[1:]).transpose((0, 2, 1, 3) + tuple(range(4, data.ndim + 1)))
data = data.reshape((n * data.shape[1], n * data.shape[3]) + data.shape[4:])

        showimage(data)
```

接着输入：

```
# index four is the center crop
# 输出输入的图像
image = net.blobs['data'].data[4].copy()
image -= image.min()
image /= image.max()
showimage(image.transpose(1, 2, 0))
```

使用 ipt.show（）观看图像：



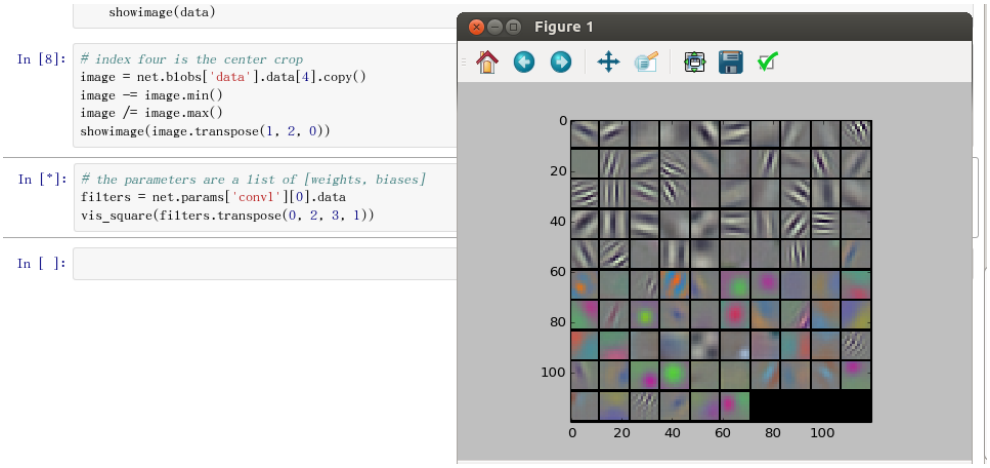
1.3 各层的特征：

第一个卷积层：

共 96 个过滤器

```
# the parameters are a list of [weights, biases]
filters = net.params['conv1'][0].data
vis_square(filters.transpose(0, 2, 3, 1))
```

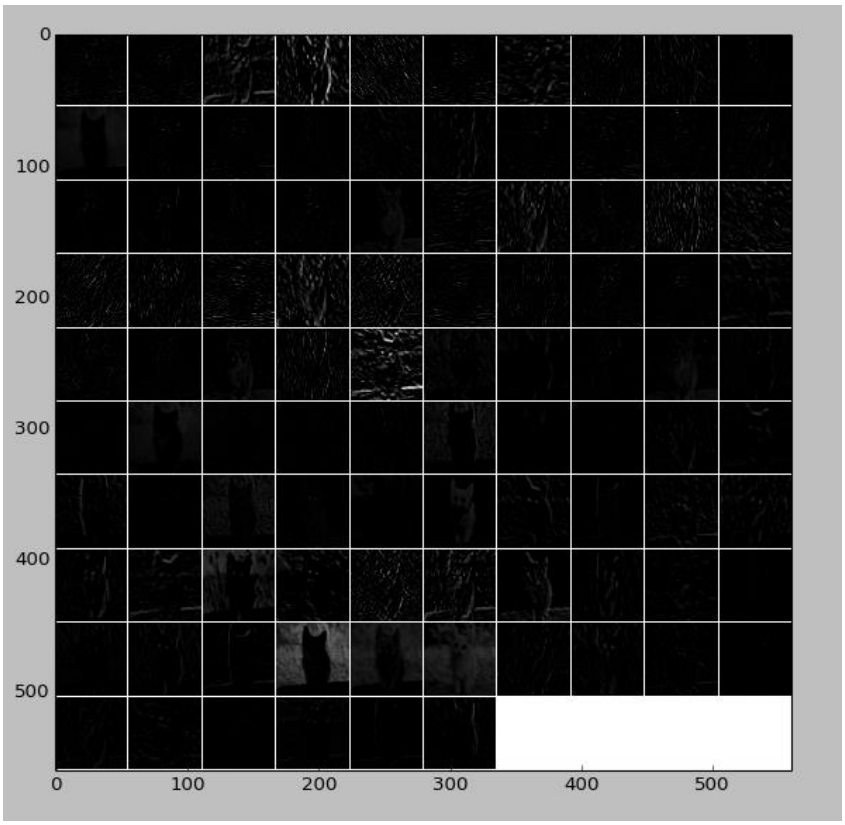
使用 ipt.show（）观看图像：



过滤后的输出，96 张 featuremap

```
feat = net.blobs['conv1'].data[4, :96]
vis_square(feat, padval=1)
```

使用 ipt.show（）观看图像：



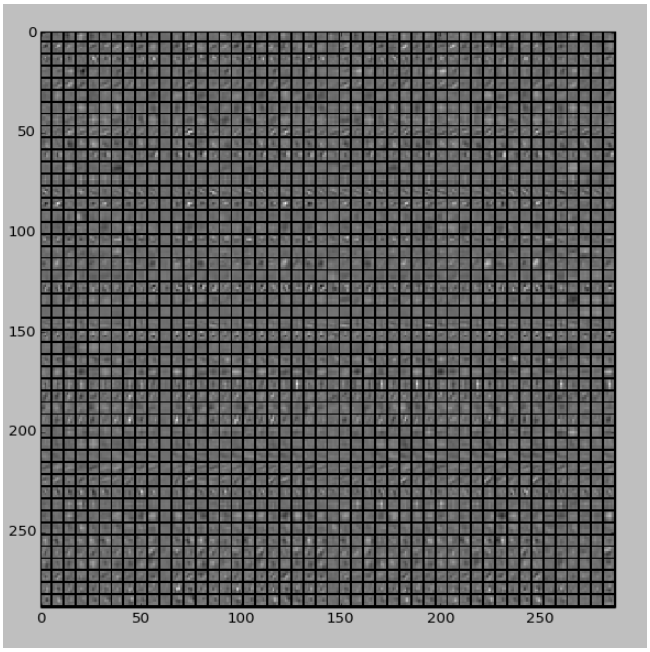
第二个卷积层：

有 128 个滤波器，每个尺寸为 5X5X48。我们只显示前面 48 个滤波器，每一个滤波器为一行。

输入：

```
filters = net.params['conv2'][0].data
vis_square(filters[:48].reshape(48**2, 5, 5))
```

使用 ipt.show（）观看图像：

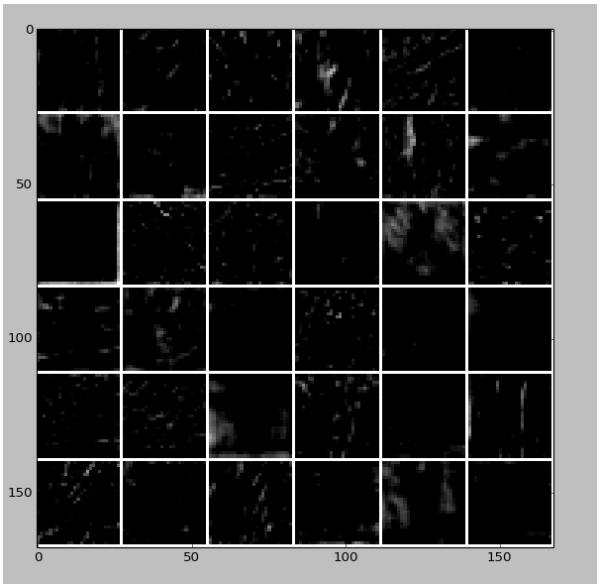


第二层输出 256 张 feature，这里显示 36 张。

输入：

```
feat = net.blobs['conv2'].data[4, :36]
vis_square(feat, padval=1)
```

使用 ipt.show（）观看图像：



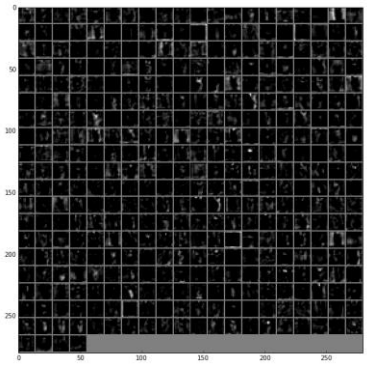
第三个卷积层：

全部 384 个 feature map

输入：

```
feat = net.blobs['conv3'].data[4]
vis_square(feat, padval=0.5)
```

使用 ipt.show（）观看图像：



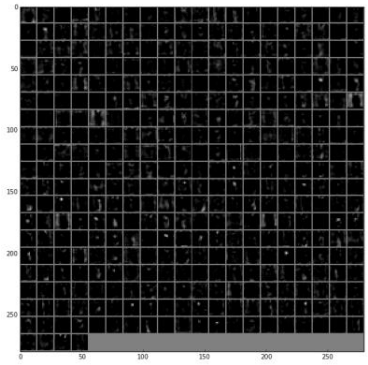
第四个卷积层：

全部 384 个 feature map

输入：

```
feat = net.blobs['conv4'].data[4]
vis_square(feat, padval=0.5)
```

使用 ipt.show（）观看图像：



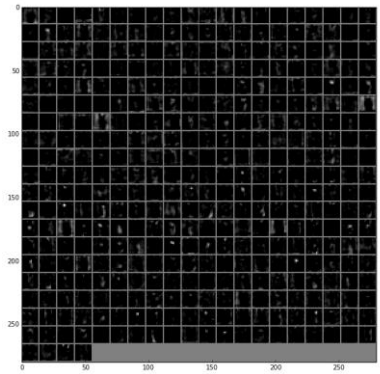
第五个卷积层：

全部 256 个 feature map

输入：

```
feat = net.blobs['conv5'].data[4]
vis_square(feat, padval=0.5)
```

使用 ipt.show（）观看图像：



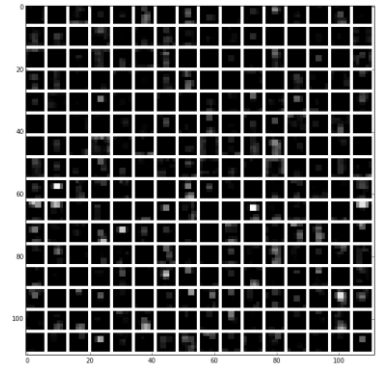
第五个 pooling 层：

我们也可以观察 pooling 层

输入：

```
feat = net.blobs['pool5'].data[4]
vis_square(feat, padval=1)
```

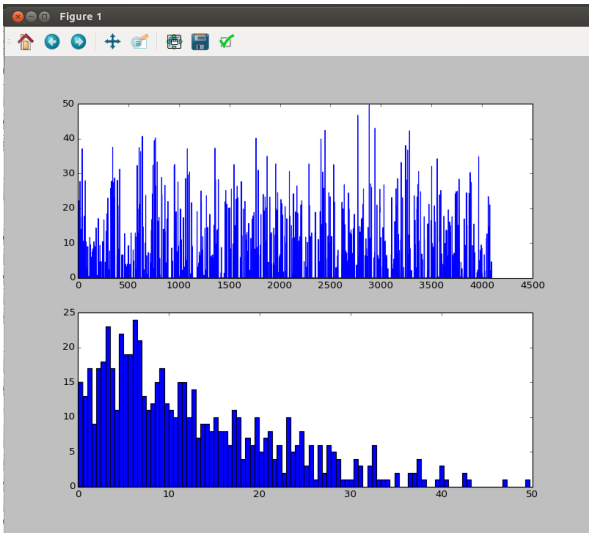
使用 ipt.show（）观看图像：



然后我们看看第六层输出后的直方分布：

```
feat = net.blobs['fc6'].data[4]
plt.subplot(2, 1, 1)
plt.plot(feat.flat)
plt.subplot(2, 1, 2)
_ = plt.hist(feat.flat[feat.flat > 0], bins=100)
```

使用 ipt.show（）观看图像：

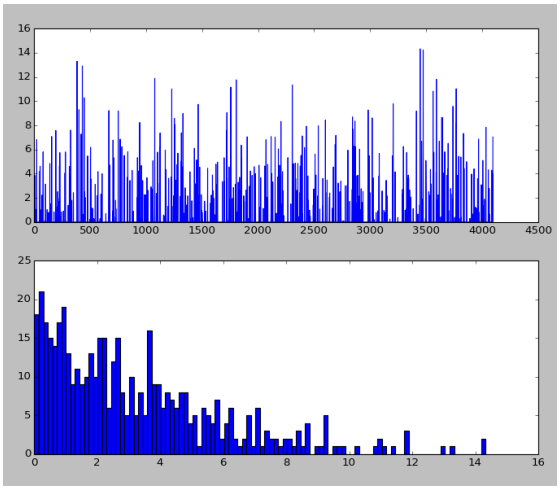


第七层输出后的直方分布：

可以看出值的分布没有这么平均了。

```
feat = net.blobs['fc7'].data[4]
plt.subplot(2, 1, 1)
plt.plot(feat.flat)
plt.subplot(2, 1, 2)
_ = plt.hist(feat.flat[feat.flat > 0], bins=100)
```

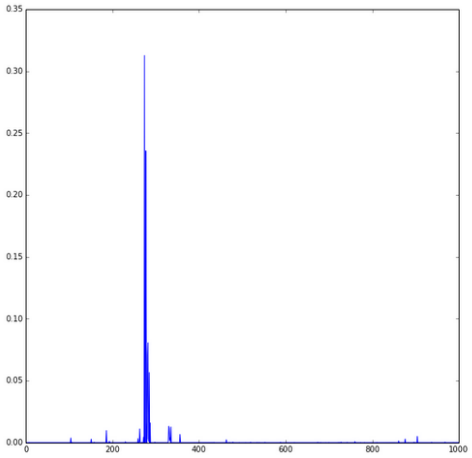
使用 ipt.show（）观看图像：



对后输出后的直方分布：

```
feat = net.blobs['prob'].data[4]
plt.subplot(2, 1, 1)
plt.plot(feat.flat)
```

使用 ipt.show（）观看图像：



最后看看标签：

```
imagenet_labels_filename = caffe_root + 'data/ilsvrc12/synset_words.txt'
try:
    labels = np.loadtxt(imagenet_labels_filename, str, delimiter='\t')
except:
    !../data/ilsvrc12/get_ilsvrc_aux.sh
    labels = np.loadtxt(imagenet_labels_filename, str, delimiter='\t')
top_k = net.blobs['prob'].data[4].flatten().argsort()[-1:-6:-1]
print labels[top_k]
```

```
In [41]: top_k = net.blobs['prob'].data[4].flatten().argsort()[-1:-6:-1]
         print labels[top_k]

['n02115913 dhole, Cuon alpinus' 'n02119022 red fox, Vulpes vulpes'
 'n02119789 kit fox, Vulpes macrotis' 'n02123159 tiger cat'
 'n02123045 tabby, tabby cat']
```

可以看出正确率还是挺高的。

资料来源：caffe 官网。

http://nbviewer.ipython.org/github/BVLC/caffe/blob/master/examples/filter_visualization.ipynb

读书笔记 6 在 python 界面上用训练好的 Imagenet 模型去分类图形

2014.7.24 薛开宇

本篇是对前面第 5 个读书笔记的一个补充。讲一下在 python 界面上分类的一些细节。

注意，本笔记的代码由原文复制过来，再复制过去 ipython 可能会有些格式上的错误，建议使用原文（本文最后）的代码进行复制使用。

1.1 开始

在终端输入：

ipython notebook

启动编辑平台。

Caffe 提供一个普遍的 python 接口用来接 caffe 的模型。该网络在 python/caffe/pycaffe.py 中。

我们可以用 python 和 matlab 进行分类，然而，python 拥有更多功能，因此，我们在这里使用它，对于 matlab，可以参考 matlab/caffe/matcaffe_demo.m.

然后，准备工作和读书笔记 5 一样，这里不描述了。

首先，以下一段指令：

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

caffe_root = '/home/xuekaiyu/caffe-master/'# this file is expected to be in {caffe_root}/examples
import sys
sys.path.insert(0, caffe_root + 'python')
import caffe
# Set the right path to your model definition file, pretrained model weights,
#设置好模型路径，和想分类的图像
# and the image you would like to classify.
MODEL_FILE = caffe_root+'examples/imagenet/imagenet_deploy.prototxt'
PRETRAINED = caffe_root+'examples/imagenet/caffe_reference_imagenet_model'
IMAGE_FILE = caffe_root+'examples/images/bird11.jpeg'
```

1.2 加载网络与输入图片

加载一个网络很简单，caffe.Classifier 已经帮你设置好一切，注意输入预处理的参数配置，减去均值的文件的设置，输入的 RGB 频道的交换（ImageNet model's 是 BGR），还有输入时候乘以一定的特征比例以达到从【0，1】到【256】的目的。

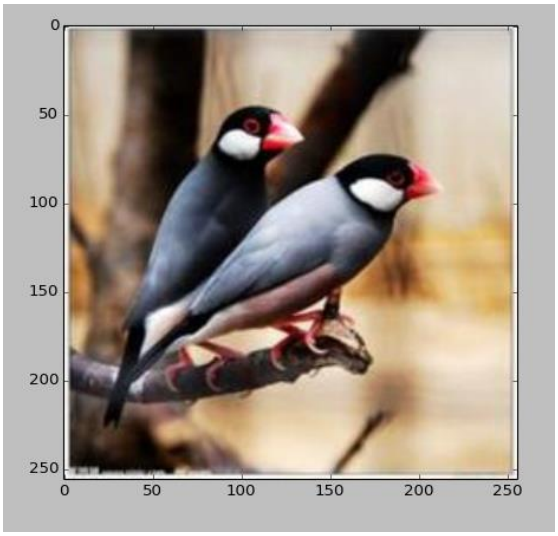
```
net = caffe.Classifier(MODEL_FILE, PRETRAINED,
                       mean_file=caffe_root+'python/caffe/imagenet/ilsvrc_2012_mean.npy',
                       channel_swap=(2,1,0),
                       input_scale=255)
```

因为是在测试阶段，而且我们先使用 CPU 计算，所以输入：

```
net.set_phase_test()
net.set_mode_cpu()
```

然后输入图片

```
input_image = caffe.io.load_image(IMAGE_FILE)
plt.imshow(input_image)
Plt.show()
```

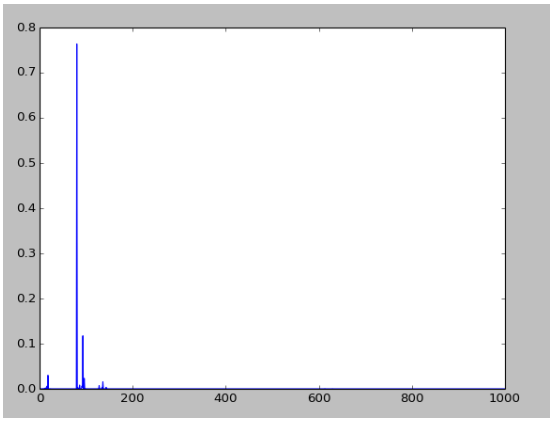


1.3 开始分类

默认是 10 种预测，裁剪照片的中心和边角，以及去掉他们的镜像版本，还有他们预测的均值。

（1）第一种分类政策：默认。

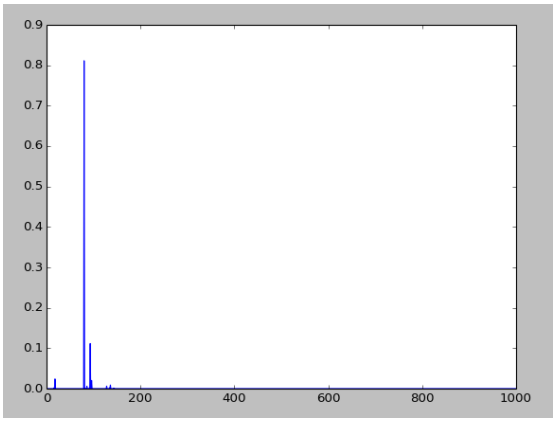
```
prediction = net.predict([input_image])  #预测可以输入任何大小的图像，caffe net 会自动调整。
print 'prediction shape:', prediction[0].shape
plt.plot(prediction[0])
```



（2）第二种分类政策：

现在我们只是用中心结果分类，关闭掉过采样，注意，对一个单一的输入，尽管我们检查模型定义 prototxt 我们可以看到网络的批次大小是 10，python 会自动处理和填充。

```
prediction=net.predict([input_image],oversample=False)
print 'prediction shape:', prediction[0].shape
plt.plot(prediction[0])
```



我们可以看到，预测是 1000 维，这样做更加分散。

我们的预训练模型使用同义词集 ID 类排序，在 ../data/ilsvrc12/synset_words.txt 可以看到对应，我们看指标，可以看到分数最大是【'n01795545 黑琴鸡”n01829413 犀鸟”n01582220 鹊 “n01843383 巨嘴鸟” n02017213 欧洲水鸡，紫水鸡’]，也算相当准确。

现在，我们可以看看执行分类的时间，这一结果是用 Intel I7 CPU，你可以看到一些性能上的差异。

1 loops, best of 3: 2.88 s per loop

这看起来有点慢，这是因为这是花费在接收获，python 的接口，并且运行 10 张照片。在性能上，可以选择编码在 C++和流水线上，实验的话当前速度是可以的。

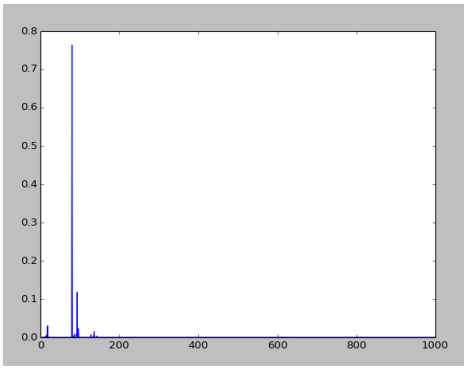
（3）第三种分类政策：

我们分类一张照片时，加入输入预处理。

```
# Resize the image to the standard (256, 256) and oversample net input sized crops.
#将照片的形状变成（256，256）同时进行过采样。
input_oversampled = caffe.io.oversample([caffe.io.resize_image(input_image, net.image_dims)], net.crop_dims)
# 'data' is the input blob name in the model definition, so we preprocess for that input.
# 数据输入的是 model 定义中 blob 的名字，所以我们预处理输入。
caffe_input = np.asarray([net.preprocess('data', in_) for in_ in input_oversampled])
# forward() takes keyword args for the input blobs with preprocessed input arrays.
# forward()为经过预处理输入的 blobs 获取关键字参数
%timeit net.forward(data=caffe_input)
```

使用的时间为 1 loops, best of 3: 2.73 s per loop，明显少了些。

让我们看看如果对我们的相同结果使用 GPU，从图可以看出，图没什么变化，那么时间来说：




```
# Full pipeline timing.
%timeit net.predict([input_image])
```

使用的时间，1 loops, best of 3: 289 ms per loop，少了非常多。之前是 S，现在是 ms 级别

```
# Forward pass timing.
%timeit net.forward(data=caffe_input)
```

使用的时间，10 loops, best of 3: 121 ms per loop，进一步减少。

事实上，使用 GPU 代码非常快，你可以看到几乎快了 10 倍，和数据加载，转换，接口都比本身的 ConvNet 计算的时间长。

若想发挥 GPU 的力量，你可以：

使用更大的批次，并且减少 Python 的调用和数据传输的开销。

管道数据加载操作，如使用 subprocess 模块来管理。

在 C++里编码。

1.4 后话

所以这是 python，我们希望接口非常简单。Python 的 boost 库和它源代码可以在

python/caffe 中发现。如果你想自己制作，从哪里开始，但是希望能给 caffe 团队知道你做出了改进并且提交。

资料来源：

http://nbviewer.ipython.org/github/BVLC/caffe/blob/master/examples/imagenet_classification.ipynb

读书笔记 7 如何改变模型参数将提取出的大量特征

用全卷积的分类器表示

2014.7.24 薛开宇

本篇主要是介绍一种方法，使输出不是整个图分类的概率，而是显示输入图各个部分的特征属于什么分类。

注意，本笔记的代码由原文复制过来，再复制过去 ipython 可能会有些格式上的错误，建议使用原文（本文最后）的代码进行复制使用。

1.1 前言

Caffe 可以通过编辑网络优化参数转化成我们所需要的模型。在这个例子，我们转化产品的内在分类器为卷积层。这会产生一个生成给定输入大小的分类图而不是单一的分类的全卷积模型。**本例子的一个分类会由每 6X6 区域的 pool5 层组成，用例子的 454x454 的输入尺寸产生一个 8X8 的分类图，如：**

```
array([[282, 282, 282, 282, 282, 278, 278, 278],
       [282, 283, 281, 281, 281, 281, 278, 282],
       [283, 283, 283, 283, 283, 287, 259, 278],
       [283, 283, 283, 283, 283, 259, 259, 259],
       [283, 283, 283, 283, 283, 283, 259, 259],
       [283, 283, 283, 283, 283, 283, 259, 259],
       [283, 283, 283, 283, 283, 259, 259, 852],
       [283, 283, 283, 283, 283, 263, 263, 331]])
```

其中，282 表示斑猫，281 表示波斯猫，有各种分类。

而原来的模型，只是给定某一个分类的概率。如：

```
['n02115913 dhole, Cuon alpinus' 'n02119022 red fox, Vulpes vulpes'
 'n02119789 kit fox, Vulpes macrotis' 'n02123159 tiger cat'
 'n02123045 tabby, tabby cat']
```

这里是列出了概率排名前 5 的分类。

需要注意的是，这个模型不适合 sliding-window 测试因为它是整个图像的分类训练。如果想进行 Sliding-window 的训练和调整，可以通过在真值和 loss 的基础上定义一个滑动窗口，这样一个 loss 图就会由每一个位置组成并且像往常一样做。（这是计划中的想法，由读者完成中）

1.2 比较

```
!diff /home/xuekaiyu/caffe-master/examples/imagenet/imagenet_full_conv.prototxt
/home/xuekaiyu/caffe-master/examples/imagenet/imagenet_deploy.prototxt
```

截取一段输入

```
151,152c171,172                    #这里的意思是第几行有差别。
<     name: "fc6-conv"             #对比第一个和第二个文件发现名字改变了
<     type: CONVOLUTION            #把内积层变成卷积层
---
>     name: "fc6"
>     type: INNER_PRODUCT
154,155c174,179
<     top: "fc6-conv"
<     convolution_param {          #因为变成卷积层，多了卷积的参数设置。
---
>     top: "fc6"
>     blobs_lr: 1
>     blobs_lr: 2
>     weight_decay: 1
>     weight_decay: 0
>     inner_product_param {
157d180
<       kernel_size: 6             #因为 pool5 的输出是 36，因此全连接卷积层也是
                                   6X6
```

从上面可以看出，唯一需要改变的就是去改变全连接的分类器内积层变成卷积层，并且使用正确的滤波器大小 6X6（由于参考模型的分类以 Pool5 的 36 个元素作为输入的），同时步长为 1 保证密集。请注意，层重新命名了以避免当图层命名为 pretrained model 时 caffe 去载入旧的参数。

1.3 比较

```
import caffe
# Load the original network and extract the fully-connected layers' parameters.
#载入传统的网络来提取全连接层的参数
net=caffe.Net(caffe_root+'examples/imagenet/imagenet_deploy.prototxt',
caffe_root+'examples/imagenet/caffe_reference_imagenet_model')
params = ['fc6', 'fc7', 'fc8']
# fc_params = {name: (weights, biases)}
#fc_params 调用的格式写法{name: (weights, biases)}
fc_params = {pr: (net.params[pr][0].data, net.params[pr][1].data) for pr in params}
for fc in params:
    print '{} weights are {} dimensional and biases are {} dimensional'.format(fc,
fc_params[fc][0].shape, fc_params[fc][1].shape)
```

输出：
fc6 weights are (1, 1, 4096, 9216) dimensional and biases are (1, 1, 1, 4096) dimensional
fc7 weights are (1, 1, 4096, 4096) dimensional and biases are (1, 1, 1, 4096) dimensional
fc8 weights are (1, 1, 1000, 4096) dimensional and biases are (1, 1, 1, 1000) dimensional

考虑到内部产生的参数的形状。权值和偏值的第 0 和第 1 个面积为 1，第 2 个，第 3 个权值面积是输出的尺寸和当是最后的偏置面积是输出尺寸时是输入的尺寸。

```
# Load the fully-convolutional network to transplant the parameters.
#  载入全连接网络去移植参数
net_full_conv=caffe.Net(caffe_root+'examples/imagenet/imagenet_full_conv.prototxt',
caffe_root+'examples/imagenet/caffe_reference_imagenet_model')
params_full_conv = ['fc6-conv', 'fc7-conv', 'fc8-conv']
# conv_params = {name: (weights, biases)}
conv_params = {pr: (net_full_conv.params[pr][0].data, net_full_conv.params[pr][1].data) for pr in
params_full_conv}

for conv in params_full_conv:
    print '{} weights are {} dimensional and biases are {} dimensional'.format(conv,
conv_params[conv][0].shape, conv_params[conv][1].shape)
```

输出：
fc6-conv weights are (4096, 256, 6, 6) dimensional and biases are (1, 1, 1, 4096) dimensional
fc7-conv weights are (4096, 4096, 1, 1) dimensional and biases are (1, 1, 1, 4096) dimensional
fc8-conv weights are (1000, 4096, 1, 1) dimensional and biases are (1, 1, 1, 1000) dimensional

卷积层的权重由输出 X 输入 X 高度 X 宽度的尺寸决定，为了匹配大小，我们需要把内部产生的维度化成频道 X 高 X 宽度的滤波器矩阵。

偏值和内连接层相同，让我们先移植因为实在没有重塑的需要。

举个例子说，

本来第 6 个全连接层，层的连接是输入 1 个神经元输出 1 个神经元，每个神经元高 4096 维，宽 9216 维，权重为 1X1X4096X9216。现在做成卷积层，必须和前面的权重一样，因此层的连接是输入 4096 个神经元，输出 256 个神经元，每个神经元高 6 维，宽 6 维，这样权重也是 4096X256X6X6=1X1X4096X9216。

开始转换

```
for pr, pr_conv in zip(params, params_full_conv):
conv_params[pr_conv][1][...] = fc_params[pr][1]
```

输出层的频道数目需要主导内连接和卷积的权重，所以参数通过从内积层重塑输入参数维度矢量转换为频道 X 高 X 宽的形状

输入：

```
for pr, pr_conv in zip(params, params_full_conv):
    out, in_, h, w = conv_params[pr_conv][0].shape
    W = fc_params[pr][0].reshape((out, in_, h, w))
    conv_params[pr_conv][0][...] = W
```

现在，存储新的模型：

```
net_full_conv.save('imagenet/caffe_imagenet_full_conv')
```

这是二进制格式。

1.4 分类

我们运行一张传统的图片运行该模型，他会生成一个 8X8 的区域标记。

```
# load input and configure preprocessing
im = caffe.io.load_image(caffe_root+'examples/images/cat.jpg')
plt.imshow(im)
net_full_conv.set_mean('data', caffe_root+'/python/caffe/imagenet/ilsvrc_2012_mean.npy')
net_full_conv.set_channel_swap('data', (2,1,0))
net_full_conv.set_input_scale('data', 255.0)
# make classification map by forward pass and show top prediction index per location
out = net_full_conv.forward_all(data=np.asarray([net_full_conv.preprocess('data', im)]))
out['prob'][0].argmax(axis=0)
```

```
array([[282, 282, 282, 282, 282, 278, 278, 278],
       [282, 283, 281, 281, 281, 281, 278, 282],
       [283, 283, 283, 283, 283, 287, 259, 278],
       [283, 283, 283, 283, 283, 259, 259, 259],
       [283, 283, 283, 283, 283, 283, 259, 259],
       [283, 283, 283, 283, 283, 283, 259, 259],
       [283, 283, 283, 283, 283, 259, 259, 852],
       [283, 283, 283, 283, 283, 263, 263, 331]])
```

。 分类图包括各种的猫，281 是波斯猫，282 是斑猫，283 是猫科动物，还有其它一些是狐狸还有其他动物。

用这种方式，全卷积层可以用在图像提取密集的特征，这比单看分类图自身更有用。

参考资料:

http://nbviewer.ipython.org/github/BVLC/caffe/blob/master/examples/net_surgery.ipynb