

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，最后两个全连接层类似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）

```
caffe.proto (-caffe-master/src/caffe/proto) - gedit
// Copyright 2014 BVLC and contributors.

package caffe;

message BlobProto {
  optional int32 num = 1 [default = 0];
  optional int32 channels = 2 [default = 0];
  optional int32 height = 3 [default = 0];
  optional int32 width = 4 [default = 0];
  repeated float data = 5 [packed = true];
  repeated float diff = 6 [packed = true];
}

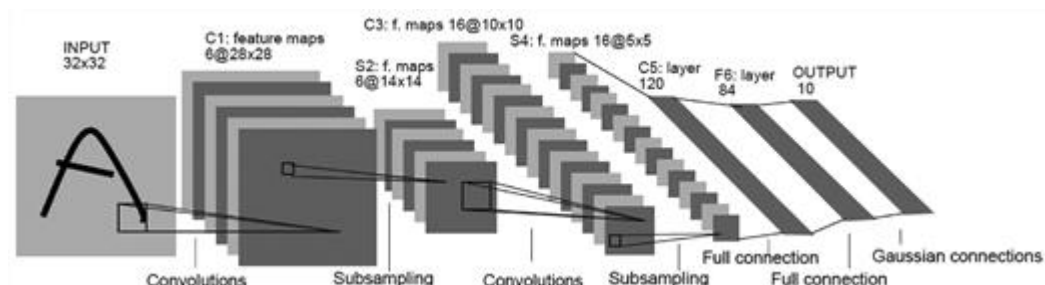
// The BlobProtoVector is simply a way to pass multiple blobproto instances
// around.
message BlobProtoVector {
  repeated BlobProto blobs = 1;
}

message Datum {
  optional int32 channels = 1;
  optional int32 height = 2;
  optional int32 width = 3;
  // the actual image data, in bytes
  optional bytes data = 4;
  optional int32 label = 5;
  // Optionally, the datum could also hold float data.
  repeated float float_data = 6;
}

message KC11wBaxmawar_F
```

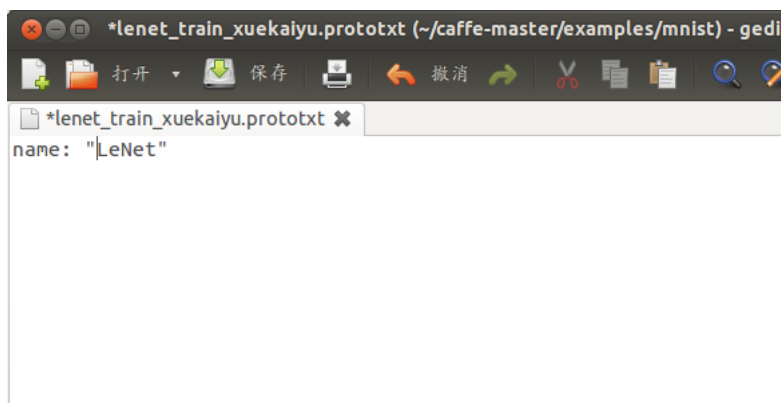
这个文档可以在我们建立自己的网络时，方便我们查到我们需要的格式。

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"
```

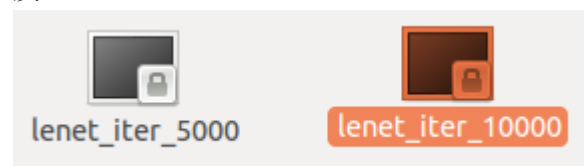
然后在终端执行指令：

```
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。

```
xuekaiyu@xuekaiyu-NS6VZ: ~/caffe-master/examples/mnist
I0720 10:46:11.396272 5088 solver.cpp:87] Iteration 400, loss = 0.285091
I0720 10:46:15.560693 5088 solver.cpp:237] Iteration 500, lr = 0.00964069
I0720 10:46:15.560940 5088 solver.cpp:87] Iteration 500, loss = 0.136647
I0720 10:46:15.560969 5088 solver.cpp:106] Iteration 500, Testing net
I0720 10:46:17.564424 5088 solver.cpp:142] Test score #0: 0.9702
I0720 10:46:17.564470 5088 solver.cpp:142] Test score #1: 0.0946137
I0720 10:46:21.720006 5088 solver.cpp:237] Iteration 600, lr = 0.0095724
I0720 10:46:21.720201 5088 solver.cpp:87] Iteration 600, loss = 0.158516
I0720 10:46:25.887486 5088 solver.cpp:237] Iteration 700, lr = 0.00950522
I0720 10:46:25.887670 5088 solver.cpp:87] Iteration 700, loss = 0.0620009
I0720 10:46:30.084939 5088 solver.cpp:237] Iteration 800, lr = 0.00943913
I0720 10:46:30.085108 5088 solver.cpp:87] Iteration 800, loss = 0.0933172
I0720 10:46:34.252975 5088 solver.cpp:237] Iteration 900, lr = 0.00937411
I0720 10:46:34.253160 5088 solver.cpp:87] Iteration 900, loss = 0.026676
I0720 10:46:38.420560 5088 solver.cpp:237] Iteration 1000, lr = 0.00931012
I0720 10:46:38.420747 5088 solver.cpp:87] Iteration 1000, loss = 0.0174171
I0720 10:46:38.420774 5088 solver.cpp:106] Iteration 1000, Testing net
I0720 10:46:40.425196 5088 solver.cpp:142] Test score #0: 0.9799
I0720 10:46:40.425235 5088 solver.cpp:142] Test score #1: 0.0604038
I0720 10:46:44.566381 5088 solver.cpp:237] Iteration 1100, lr = 0.00924715
I0720 10:46:44.566571 5088 solver.cpp:87] Iteration 1100, loss = 0.0479316
I0720 10:46:48.734275 5088 solver.cpp:237] Iteration 1200, lr = 0.00918515
I0720 10:46:48.734442 5088 solver.cpp:87] Iteration 1200, loss = 0.0871936
```

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>