**基于深度学习的验证码自动识别**

# 实验原理

1.1  LeNet网络模型

LeNet是一个典型的卷积神经网络结构，由人工智能神经网络领域的大牛Yann LeCun提出。Yann LeCun被业界誉为“卷积神经网络之父”，纽约大学终身教授，同时现在也是Facebook人工智能实验室负责人，LeNet网络结构如下：

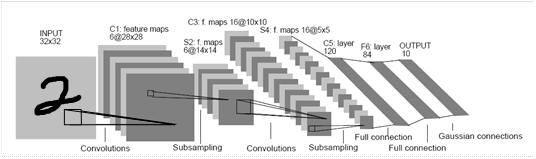


图1-2  LeNet网络结构示意图

这个网络包含两个卷积层（Convolutions layers），两个池化层(Subsampling layers)，用于逐层提取特征。两个全连接层(Full connection layers)用于表示数据的高层特征表示，最后一层用于分类。

7.3.2实验流程

整体实验分为四个部分，数据集准备、模型设计、模型训练、模型测试。

### （1）数据集准备

下载数据：对于模型实现分类、识别或检测等功能，深度学习模型训练首先需要准备大量数据。数据集一般分为训练集和测试集，训练集用于训练模型，测试集用于测试训练过程中模型的准确率。实验所用到的验证码数据集是已被人工标注过未被切分的图片，包含了1000张训练集和100张测试集，可直接从深度学习实验平台中获取数据，平台的指定位置放置了各种现成的数据集，可直接将验证码数据集复制到项目的工作目录下。

格式转换：二进制格式的文件，不能直接作为Caffe的训练数据，需要将其转换为Caffe可以识别的LMDB或LevelDB格式。相比LevelDB格式，LMDB读取效率更高，现转换格式多设置为LMDB。格式转化脚本文件为create\_mnist.sh，内容如下：

#!/usr/bin/env sh

# This script converts the mnist data into lmdb/leveldb format,

# depending on the value assigned to $BACKEND.

EXAMPLE=train or test \_lmdb #转换成功后数据存位置文件夹

DATA=data/mnist #原始数据位置

BUILD=caffe-master/build/tools/convert\_imageset #执行数据转换程序所在的位置

BACKEND="lmdb" #指定数据格式

echo "Creating ${BACKEND}..." #输出提示信息,$在shell中,表示引用变量

rm -rf $EXAMPLE/mnist\_train\_${BACKEND}

rm -rf $EXAMPLE/mnist\_test\_${BACKEND}

#开始训练数据转换

$BUILD/convert\_mnist\_data.bin $DATA/train-images-idx3-ubyte \

$DATA/train-labels-idx1-ubyte $EXAMPLE/mnist\_train\_${BACKEND} --backend=${BACKEND}

$BUILD/convert\_mnist\_data.bin $DATA/t10k-images-idx3-ubyte \

$DATA/t10k-labels-idx1-ubyte $EXAMPLE/mnist\_test\_${BACKEND} --backend=${BACKEND}

echo "Done."

格式转化脚本实际调用的是convert\_mnist\_data.bin，由convert\_minst\_data.cpp在Caffe编译后生成。运行脚本，会在data目录下下生成test\_lmdb和train\_lmdb两个文件夹，如果操作失误确定把已经生成的lmdb文件删掉重新执行，否则会报错。

### （2）模型设计

模型采用LeNet网络结构文件主要是三个：

solver.prototxt 网络参数文件

trainval.prototxt 训练验证网络

deploy.prototxt 测试网络

与论文描述的网络结构相比，LeNet在Caffe上的实现略有不同，其trainval.prototxt网络结构定义如下，定义了各个层的一些参数。具体代码如下：

name: "LeNet" # 定义网络名字

################train#################

layer {

name: "mnist"

type: "Data" #层的类型是Data，则数据集格式需要为LMDB或LevelDB格式

top: "data" #当前层的输出blob有两个，data和label

top: "label" 这边top有两个brob什么意思？

include {

phase: TRAIN #当前层仅在train阶段有效

}

transform\_param { #转换参数

scale: 0.00390625 #将将输入数据由0-255归一化到0-1之间

}

data\_param { #数据层参数

source: "data/train\_lmdb" #LMDB格式的训练集的路径（此处为相对路径）

batch\_size: 64 #批量处理的数目，一次迭代读取64张图

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

}

}

##################test#############

layer { #定义一个层

name: "mnist"

type: "Data"

top: "data"

top: "label"

include {

phase: TEST #当前层仅在test阶段有效

}

transform\_param {

scale: 0.00390625

}

data\_param {

source: "data/test\_lmdb"

batch\_size: 40

backend: LMDB

}

}

############conv##################

layer { #定义一个新的卷积层conv1，输入blob为data，输出blob为conv1

name: "conv1"

type: "Convolution"

bottom: "data"

top: "conv1"

param {

lr\_mult: 1 #权值学习速率倍乘因子，1表示1\*base\_lr，此层权值的学习率为基础学习率的1倍

}

param {

lr\_mult: 2 #bias学习速率倍乘因子，2表示此层偏置的学习率为base\_lr的2倍

}

convolution\_param { #卷积计算参数

engine:CAFFE

num\_output: 20 #当前层有20个卷积核

kernel\_size: 5   #卷积核的大小是5\*5

stride: 1 #步长为1

weight\_filler {

type: "xavier" #权值使用xavier填充器进行初始化

}

bias\_filler {

type: "constant" #偏置使用常数填充器进行初始化，默认为0

}

}

}

layer {

name: "pool1"

type: "Pooling"

bottom: "conv1"

top: "pool1"

pooling\_param {

pool: MAX

kernel\_size: 2

stride: 2

}

}

layer {

name: "conv2"

type: "Convolution"

bottom: "pool1"

top: "conv2"

param {

lr\_mult: 1

}

param {

lr\_mult: 2

}

convolution\_param {

engine:CAFFE

num\_output: 50

kernel\_size: 5

stride: 1

weight\_filler {

type: "xavier"

}

bias\_filler {

type: "constant"

}

}

}

layer {

name: "pool2"

type: "Pooling"

bottom: "conv2"

top: "pool2"

pooling\_param {

pool: MAX

kernel\_size: 2

stride: 2

}

}

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"

}

}

}

layer {

name: "relu1"

type: "ReLU"

bottom: "ip1"

top: "ip1"

}

layer {

name: "ip2"

type: "InnerProduct" #InnerProduct表示全连接层

bottom: "ip1"

top: "ip2"

param {

lr\_mult: 1

}

param {

lr\_mult: 2

}

inner\_product\_param {

num\_output: 56 #网络最后一个全连接层的输出个数56，对应最后输出56了个分类

weight\_filler {

type: "xavier"

}

bias\_filler {

type: "constant"

}

}

}

layer { #accuracy层，输出在数据集上的准确率

name: "accuracy"

type: "Accuracy"

bottom: "ip2"

bottom: "label"

top: "accuracy"

include {

phase: TEST #只在测试阶段有效

}

}

layer {   #loss层，损失函数采用softmaxLoss，输出迭代过程中的损失值。

name: "loss"

type: "SoftmaxWithLoss"

bottom: "ip2"

bottom: "label"

top: "loss"

}

### （3）模型训练

训练模型命令如下：

./build/tools/caffe train --solver=solver.prototxt

脚本中实际调用Caffe根目录build/tools/caffe命令，其指定了两个参数，第一个参数“train”表示这是用于训练，第二个参数“solver”为必选参数，指定了训练超参数文件solver.prototxt。超参文件中配置了一些参数信息，文件内容如下：

#用于训练/预测的网络描述文件（ProtoBuffer文本格式）

net: "trainval.prototxt"

test\_iter: 10 #测试集数量：10\*40=400 每迭代500次验证10张

#训练时每迭代500次进行一次预测

test\_interval: 500

#网络基础学习率、动量和权重衰减

base\_lr: 0.01

momentum: 0.9

weight\_decay: 0.0005

#学习率衰减策略

lr\_policy: "inv"

gamma: 0.0001

power: 0.75

#每迭代100次，在屏幕上打印一次运行log

display: 200

#最大迭代次数

max\_iter: 40000

#指定生成模型的保存位置

snapshot: 5000

snapshot\_prefix: "model/verifycode"

#使用GPU 训练，没有GPU则设置为CPU

solver\_mode: GPU

运行脚本后，模型开始训练，训练时间较短。最终在model/verifycode位置生成两种文件，verifycode\_iter\_\*.caffemodel和verifycode\_iter\_\*.solverstate。由于“snapshot”设置为5000，max\_iter设置为40000，最终产生8模型。其中“.caffemodel”文件保存模型权值，即最后使用的权值文件，“.solverstate”保存状态，用于训练中断后继续训练。继续训练的命令如下：

#./build/tools/caffe train \

--solver=solver.prototxt \

--snapshot=model/verifycode\_iter\_20000.solverstate

### （4）模型测试

测试训练好的模型，命令如下：

#./build/tools/caffe.bin test \

-model trainval.prototxt \

-weights model/verifycode\_iter\_20000.caffemodel \

-iterations 100

命令表示，只进行模型的预测，不进行参数更新。即只进行正向传播，不进行反向调整。模型迭代次数为100次。

# 7.4实验步骤

本实验采用Caffe镜像，请在平台中创建一个Caffe镜像，创建成功后，平台中会显示所创环境的内部IP，以及登录名和登录密码。用户可在“工具下载”模块下载OpenVPN和Xshell工具，使用其连接上实验环境。详细使用，可参考平台中的“使用手册”。

过程：

data:

1.train\_nosplit:未分割的训练集验证码图片，共1000张

test\_nosplit:未分割的测试集验证码图片，100张

2.运行split\_verifycode.py:

train\_nosplit分割后为4000张（每张验证码4个字符），保存在data/train\_split文件夹中

t est\_nosplit分割后为400张（每张验证码4个字符），保存在data/test\_split文件夹中

caffe\_verifycode:

3.编写test.txt与train.txt标签文件，label是直接拿的别人用的，后面测试模型要用到。

4.根据txt文件与分割后的数据集转换成lmdb格式保存在data/test or train\_lmdb目录下

5.根据txt文件与分割后的数据集生成均值文件mean.binaryproto。

6.模型训练和测试

Github/ReadMe.mb:

|  |
| --- |
| # -Automatic-recognition-of-verification-code-based-on-deep-learning-method  Automatic recognition of verification code based on deep learning method <br>  data: <br>  1.train\_nosplit:未分割的训练集验证码图片，共1000张 <br>  test\_nosplit:未分割的测试集验证码图片，100张 <br>  2.运行split\_verifycode.py: <br>  train\_nosplit分割后为4000张（每张验证码4个字符），保存在data/train\_split文件夹中 <br>  test\_nosplit分割后为400张（每张验证码4个字符），保存在data/test\_split文件夹中 <br>  caffe\_verifycode: <br>  3.编写test.txt与train.txt标签文件，label是直接拿的别人用的，后面测试模型要用到。 <br>  4.根据txt文件与分割后的数据集转换成lmdb格式保存在data/test or train\_lmdb目录下 <br>  5.根据txt文件与分割后的数据集生成均值文件mean.binaryproto。 <br>  6.模型训练和测试 <br> |

7.4.1 下载数据

验证码数据集data中共有两个文件：train\_nosplit和test\_nosplit。可从平台中的数据集下载模块获得，下载到本地后可通过WINSCP或XFTP等软件上传data文件夹至你的工作目录下。

7.4.2标签和列表文件生成

列表文件也就是train.txt和test.txt生成，第一列是文件名，第二列是对应的标签编号，可以自己设置对应关系，在此给出参考生成方法。

#总共56个label，忽略0,o,O,1,l,L(本来是64个的，这几个不方便区分)

find data/train\_split/ -type f | awk '{print substr($0,length($0)-4,1)}' | sort | uniq > label.uniq

find data/train\_split/ -type f | awk '{print $0" "substr($0,length($0)-4,1)}' | cut -f3 -d'/' > train\_tmp.txt

awk -F' ' 'BEGIN{id=0}NR==FNR{a[$1]=id;id+=1}NR>FNR{print $1,a[$2]}' label.uniq train\_tmp.txt > train.txt

find data/test\_split/ -type f | awk '{print $0" "substr($0,length($0)-4,1)}' | cut -f3 -d'/' > test\_tmp.txt

awk -F' ' 'BEGIN{id=0}NR==FNR{a[$1]=id;id+=1}NR>FNR{print $1,a[$2]}' label.uniq test\_tmp.txt > test.txt

#删除临时文件只留下train.txt和test.txt

rm label.uniq

rm train\_tmp.txt

rm test\_tmp.txt

# vim examples/mnist/label.txt

在项目根目录下编辑标签文件

# vim label.txt

标签label.txt内容如下，由于window和linux换行符步态，此文件需在linux环境下编写，不可直接复制粘贴此处内容。且Linux编辑文件时，不可有多余的空格或回车，尤其是最后一行。

0-2

1-3

2-4

3-5

4-6

5-7

6-8

7-9

8-a

9-A

10-b

11-B

12-c

13-C

14-d

15-D

16-e

17-E

18-f

19-F

20-g

21-G

22-h

23-H

24-i

25-j

26-J

27-k

28-K

29-m

30-M

31-n

32-N

33-O

34-p

35-P

36-q

37-Q

38-r

39-R

40-s

41-S

42-t

43-T

44-u

45-U

46-v

47-V

48-w

49-W

50-x

51-X

52-y

53-Y

54-z

55-Z

7.4.3格式转换为LMDB

由于lmdb是caffe爱用的一种数据格式，所以我们需要对原始的数据进行格式转换，使用caffe自带的工具。

#将训练集转换成lmdb格式

caffe-master/build/tools/convert\_imageset --shuffle \

--resize\_height=26 --resize\_width=22 \

data/train\_split/ \

./train.txt \

data/train\_lmdb

#将测试集转换成lmdb格式

caffe-master/build/tools/convert\_imageset --shuffle \

--resize\_height=26 --resize\_width=22 \

data/test\_split/ \

./test.txt \

data/test\_lmdb

注：若执行此脚本转化失败后，再次执行时，生成的文件并不会“覆盖写”。因此若第一次执行失败后，请删除生成的错误的文件后，再重新执行脚本。

此外Caffe中的大多脚本在调用框架的工具时，路径都会以caffe-master的路径为默认路径，所以执行过程中请，在Caffe的项目目录下执行（即平台中的/root/cDeep/Caffe/caffe-master目录下）。若报“file not find”相关错误，请仔细检查路径是否正确。

7.4.4 均值计算

mean.binaryproto是均值文件，为加快计算的效率和训练的模型准确率而设计的，也通过caffe自带的工具compute\_image\_mean 来实现，数据集就采用train\_lmdb就可以。

caffe-master/build/tools/compute\_image\_mean data/train\_lmdb ./mean.binaryproto

7.4.5模型训练和测试

该准备的数据都准备好，接下来我们就开始训练。先看下模型定义文件，看看哪里有没有什么需要改动的地方（其实得做一点小修改，否则不能训练）。打开：trainval.prototxt，注意最后ip2层的num\_output，应该与标签个数56一致。

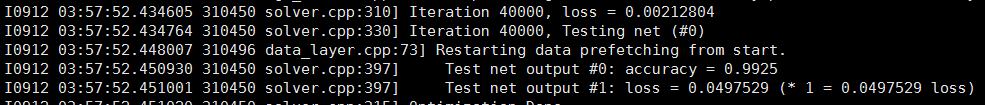
开始训练，同样使用caffe核心模块caffe：

caffe-master/build/tools/caffe train -solver solver.prototxt

注：平台手册中的Caffe相关的所有“\”加回车，均可用空格代替，若直接复制粘贴手册中的命令，遇到“\”请一行一行粘贴。

# 7.5实验结果

模型训练：训练部分日志如下：



模型测试：大概半个小时左右，模型已经训练好了，就是verify\_code\_iter\_40000.caffemodel，而且已经对测试数据进行了预测，准确率达到99%，可能每次会有少许差别。我们现在可以在单个图片上测试下了，同样使用caffe自带的工具： 先说下deploy.prototxt文件，是基于trainval.prototxt修改的，主要将输入层变成了Input层，输出的Accuracy和loss层去掉并加入了prob层。

命令如下：

./build/examples/cpp\_classification/classification.bin \

deploy.prototxt \

model/verifycode\_iter\_40000.caffemodel\

mean.binaryproto \

label.txt \

data/test\_split/1100-2-Y.jpg

测试结果：

