

卷积神经网络原理

一、实验介绍

1.1 实验内容

本课程将会先带你理解卷积神经网络的原理，了解卷积神经网络的一些特性。然后动手使用 `caffe` 深度学习框架训练一个卷积神经网络模型，并用训练好的模型进行图片分类。

学习本课程之前，请先学习课程 [814 使用 python 实现深度神经网络](#) 以了解必要的基本概念，本实验中涉及到的深度学习基本概念将不会再次阐述。同时建议学习课程 [744 深度学习初探—入门 DL 主流框架](#) (该课为训练营课程) 中的 `caffe` 相关部分，但本课程中涉及到 `caffe` 的内容也会尽量讲解。

本实验作为本课程的第一次实验，将会向大家讲解卷积神经网络的相关基本原理。除非你已经知道什么是卷积神经网络，否则我强烈建议你仔细学习本次实验，理解原理将非常有助于你在实际问题中合理地使用卷积神经网络。

1.2 实验知识点

- 卷积神经网络的结构

1.3 先学课程

- [814 使用 python 实现深度神经网络](#) (必学，否则可能无法理解本课程中的一些概念)
- [744 深度学习初探—入门 DL 主流框架](#) (选学)

1.4 实验环境

- python 2.7
- `caffe` 1.0.0 (实验楼环境中已经预先安装)

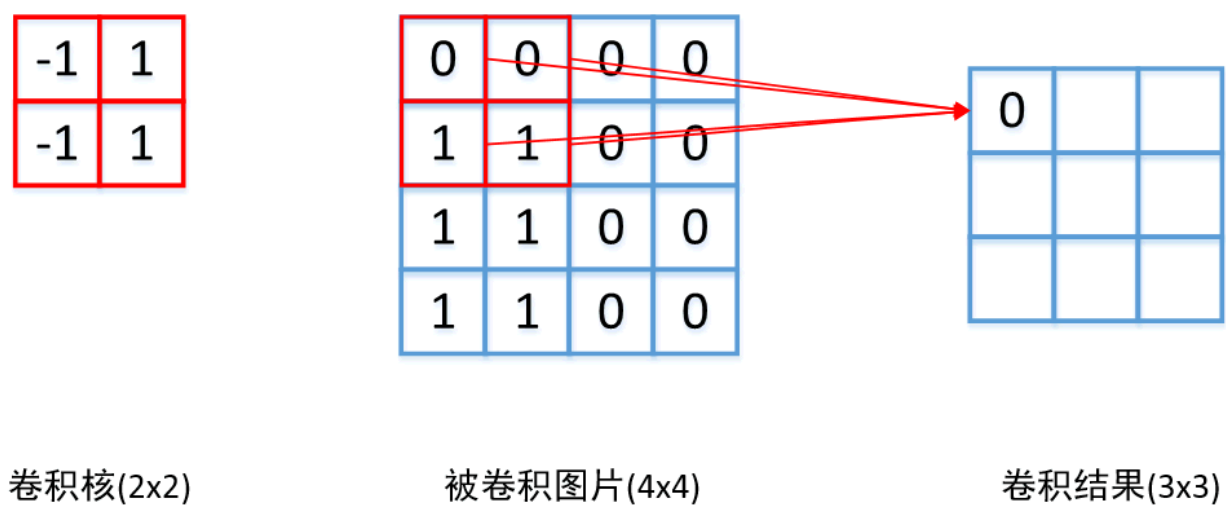
二、实验步骤

2.1 什么是“卷积”

在课程 [814 使用 python 实现深度神经网络](#) 中，我们已经构建过深度神经网络，注意在那个网络中，除了激活函数层和损失函数层之外，发挥主要作用的是全连接层，我们的网络参数全部都在全连接层中。那么既然使用全连接层构成的神经网络就已经实现了对图片的分类，为什么我们还需要“卷积神经网络”呢？为了回答这个问题，我们先来了解到底什么是 **卷积(convolution)**。

2.1.1 卷积操作

“卷积”这一词在多个领域里都有定义（比如信号处理领域的傅里叶变换中也有卷积）。具体在图像处理领域，卷积操作是指使用一个小的“模板窗口”对一个图片中的所有与模板大小相同的区域进行“卷积运算”。“卷积运算”其实很简单，就是将模板中的每一个数字与图片中相同大小区域的对应数字（像素值）进行相乘，再求和。具体操作如下图：



“模板窗口”每移动到一个位置，就和图片中的对应元素进行一次卷积运算，注意我们一般把“模板窗口”称为卷积核(kernel)。

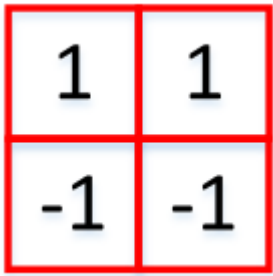
比如在第一个位置，图片上的四个像素值为 $[[0, 0], [1, 1]]$ ，而卷积核中的数值为 $[[-1, 1], [-1, 1]]$ ，对应元素相乘再求和，得到 $0 \times -1 + 0 \times 1 + 1 \times -1 + 1 \times 1 = 0$

比如在第二个位置，图片上的四个像素值为 $[[0, 0], [1, 0]]$ ，对应元素相乘再求和，得到 $0 \times -1 + 0 \times 1 + 1 \times -1 + 1 \times 0 = -1$

对图片中的所有可能位置进行卷积操作，就得到了最终的卷积结果。

2.1.2 为什么卷积操作适合用来处理图像

上图中的卷积核，其实是经过精心设置的。观察被卷积的图像和卷积结果你会发现，该卷积核其实可以用来检测图片中的垂直边缘。如果卷积后得到的数字绝对值大，就说明图片上的对应地方有一条垂直方向的边缘（即像素数值变化较大）。如果卷积后得到的数字绝对值很小，则说明图片上的对应地方像素值变化不大，没有边缘存在。同样的道理，你可以构造一个检测水平方向边缘的卷积核：



你可以尝试用这个卷积核去对上面的图片进行卷积操作，验证这个卷积核是不是检测出了水平方向的图片边缘。

卷积操作还可以有更多强大的功能，在数字图像处理中经常会用到。

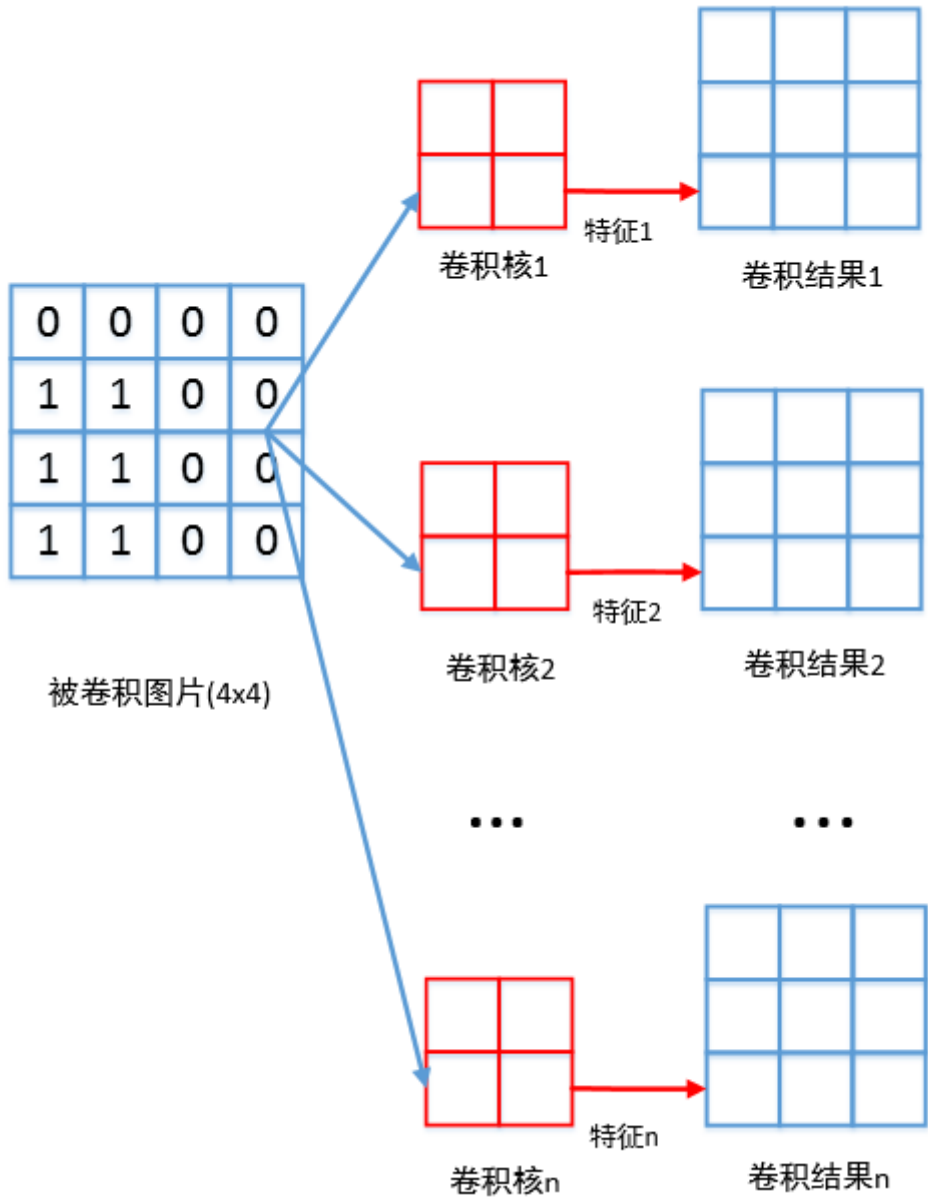
说到这里，你也许已经能够体会为什么我们需要卷积神经网络。对于将整个图片视为一个向量的全连接层来说，它忽略了图像本身所具有的“二维空间特性”，或者说局部特性。而卷积操作则十分擅长处理这种局部特性，能够更有效地提取出图片中的更多有用信息（比如图片中的边缘）。实际上，卷积神经网络几乎已经成为了神经网络中处理图像的标配。

2.2 神经网络中的卷积层

在具体的将卷积操作运用到神经网络的过程当中，有许多细节需要注意（包括很多超参数）。首先需要明确的是，和课程 814 使用 python 实现深度神经网络中介绍的一样，卷积层中的参数（即卷积核中的每个数字）也是通过学习算法（随机梯度下降算法）学习得到的，不需要手工设定。

2.2.1 特征个数

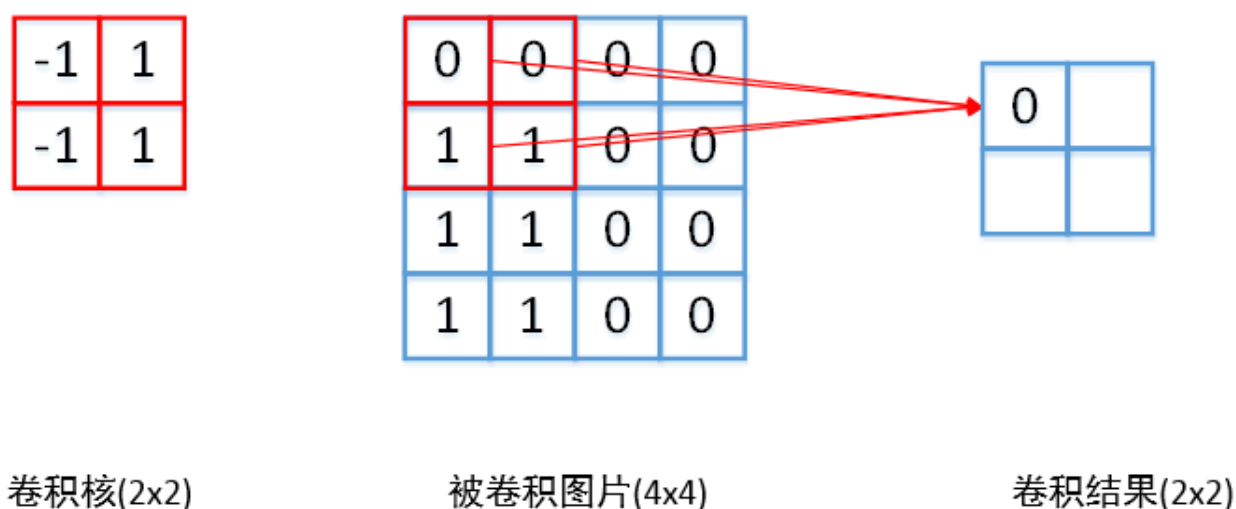
一个卷积核，只能检测一种特征(feature)（比如垂直方向的边缘），而图片中的信息往往很复杂，一个卷积核显然不够。所以神经网络中，一个卷积层往往会有多个卷积核，这样卷积层的输出就会有多层。如下图：



为了简便，在一些深度学习框架（比如本课程会用到的 **caffe**）中，在运算时可以将特征个数与图像通道数等同看待。所以对于一个通道数为 1 的图片，若卷积层特征个数为 3，则可将输出的卷积结果视为通道数为 3 的图片。对于一个通道数为 3 的图片（此时，卷积核是一个三维的“体”），若卷积层特征个数为 16，则可将输出的卷积结果视为通道数为 16 的图片。

2.2.2 stride

上面的示意图中，卷积核在水平和垂直方向上每次都移动一个像素的距离，在实际的卷积神经网络中，卷积核可能一次移动不止一个像素。卷积核每次移动的“步长”被称为 **stride**。**stride** 的大小会影响最后卷积结果的大小。比如将水平和垂直方向的 **stride** 都从 1 改成 2，原来的示意图就会变成这样：



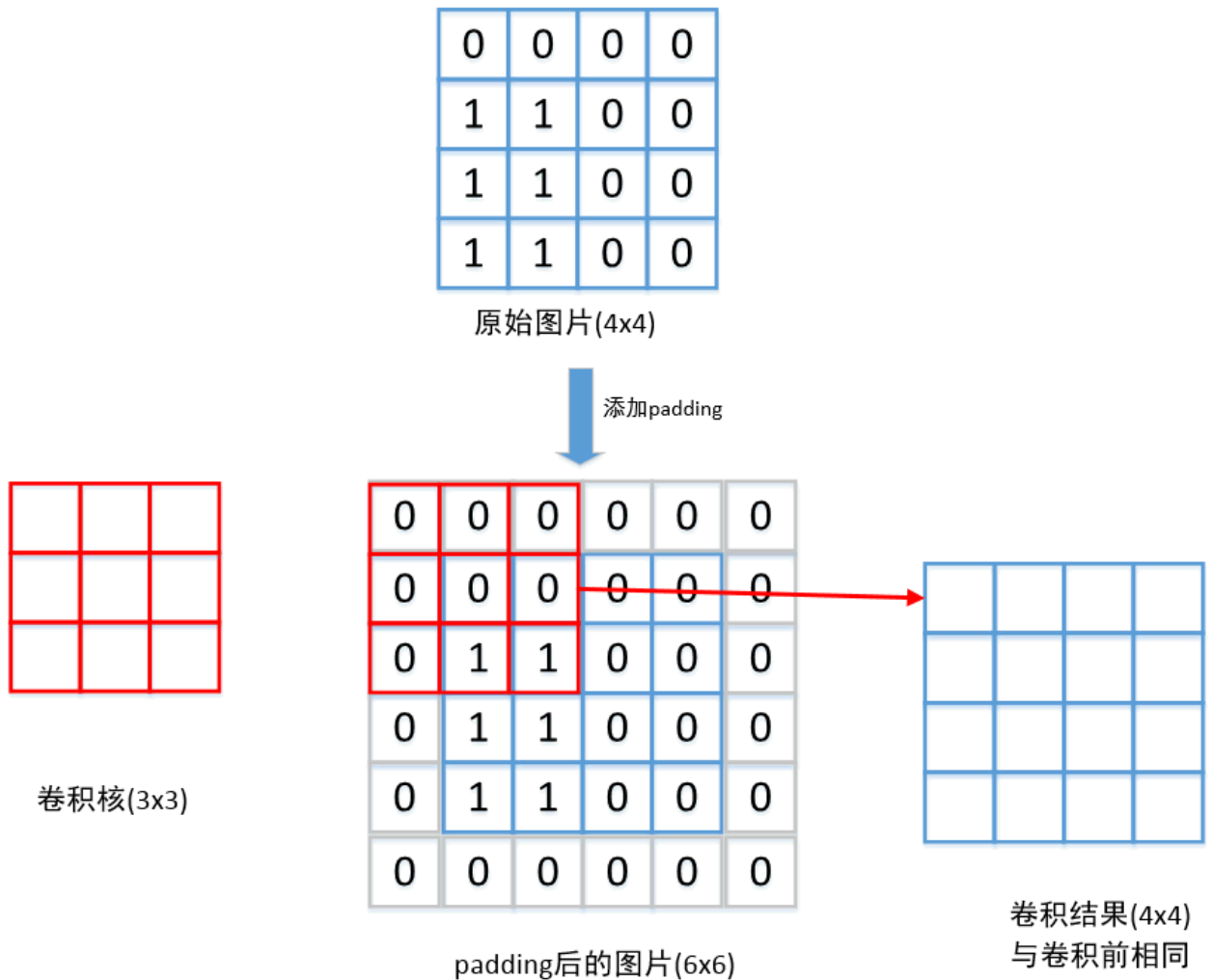
2.2.3 卷积核尺寸(kernel size)

上面的示意图中卷积核的尺寸是 2x2，但在实际当中，很少见到边长为偶数的卷积核。且一般卷积核的最小边长为 3（也存在边长为 1 的卷积核，你也许会觉得奇怪，边长为 1 的卷积层与全连接层岂不是几乎没有区别了，这个问题保留在这里，我们以后也许会来回答这个问题）。

2.2.4 padding

观察上面的图片你会发现，卷积操作得到的卷积结果与原来的图片相比尺寸变小了。这样会带来一个问题，有的卷积神经网络会非常“深”（比如几十层，甚至上百上千层）如果每经过一个卷积层我们的图片都变小一点，那到最后，有可能我们的图片都没有卷积核大，没办法继续卷积下去了。为了解决这个问题，人们对卷积层引入了**边缘填充(padding)**：在卷积前，先将图片扩大一点点，然后再进行卷积。对于扩大部分，一般会填入 0 值。如下图（为了作图方便，卷积核与卷积结果之间只使用了一个箭头进行指

示)：



2.2.5 计算卷积结果的尺寸

对于一个尺寸为 $w \times w$ 的正方形图片，卷积结果的尺寸（假设其尺寸是 $o \times o$ ），会受到上面的特征个数 $feature$ 、步长 $stride$ 、卷积核尺寸 $kernel$ 、边缘 $padding$ 四个超参数的影响。你可以尝试一下推导卷积结果尺寸的公式，其实很简单：

$$o = (w + 2 \times padding - kernel) / stride + 1$$
 且输出结果的通道数等于 $feature$

这个公式只适合于输入图片为正方形，卷积核也为正方形，步长在水平和垂直方向一致， $padding$ 在四个方向都一样的情况（不满足这些条件的情况你可以自己推导）。比如对于尺寸为 17×17 的输入图片， $padding$ 为 1， $kernel$ 的尺寸为 3×3 ， $stride$ 为 1，则卷积之后的输出边长为： $(17 + 2 \times 1 - 3) / 1 + 1 = 17$ ，没有变化。

注意根据上面的公式，你会发现卷积层的几个超参数之间需要满足如下关系：

$$(w + 2 \times padding - kernel) \% stride == 0$$

其中的 $\%$ 为取模符号，即 $(w + 2 \times padding - kernel)$ 需要是 $stride$ 的整数倍，不然无法除尽。

2.3 卷积层的实现

卷积层相对全连接层来说比较复杂（从其中超参数的个数很多你就能看出来）。你可以接着课程 814 尝试自己使用 `python` 实现卷积层，其实原理都是一样的，关键都是运用链式法则去求梯度。不过除非你

有特别的兴趣，不然不推荐自己“造轮子”（作者自己造了一个，如果你对代码有兴趣，请到 [Github](#) 查看）。幸运的是好多现有的深度学习框架已经帮我们实现好了卷积层（比如本课程会用到的 [caffe](#)），直接拿过来用就好了。

三、实验总结

本次实验我们学习了卷积操作的基本概念。掌握了这些基本概念，之后的实验对你来说会非常容易。如果你对其中的一些概念还感到困惑，不用感到受挫，刚开始接触新东西都会有这种感觉。请你再仔细思考一遍这些概念的含义，然后去继续做之后的实验，当碰到相应的概念时就回过头来看一看，相信这些对你来说都不难掌握。

本次实验，我们学习了：

- 对于图像处理，卷积层比全连接层更有优势。
- 可以将使用卷积核对图像进行的卷积操作视作对图像提取特征的过程。
- 卷积层中含有多个超参数，包括特征数 `feature`、卷积核尺寸 `kernel`、边缘填充 `padding`、卷积步长 `stride`。

四、课后作业

以下的作业都能根据上面的文档推测出答案。

1. 请你设计一个卷积核，用来检测图片对角线方向的边缘。
2. 请你计算，对于一个 $3 \times 100 \times 100$ （3 通道图片）尺寸的图片，使用 `padding=1`（四个方向都填充一个像素的长度），`kernel=7`（`kernel` 为正方形），`stride=5`（水平和垂直方向的 `stride` 相同）的卷积层进行卷积后，得到的输出的尺寸。

使用 `caffe` 构建卷积神经网络

一、实验介绍

1.1 实验内容

上一次实验我们介绍了卷积神经网络的基本原理，本次实验我们将学习如何使用深度学习框架 `caffe` 构建卷积神经网络，你将看到在深度学习框架上搭建和训练模型是一件非常简单快捷的事情（当然，是在你已经理解了基本原理的前提下）。如果上一次实验中的一些知识点你还理解的不够透彻，这次以及之后的实验正是通过实际操作加深对它们理解的好机会。

1.2 实验知识点

- `caffe` 网络总体结构
- `caffe` 训练数据的准备

- `caffe network` 定义文件的编写
- `SoftmaxWithLoss` 损失层
- `Accuracy` 准确率层
- `ReLU` 激活函数层

1.3 实验环境

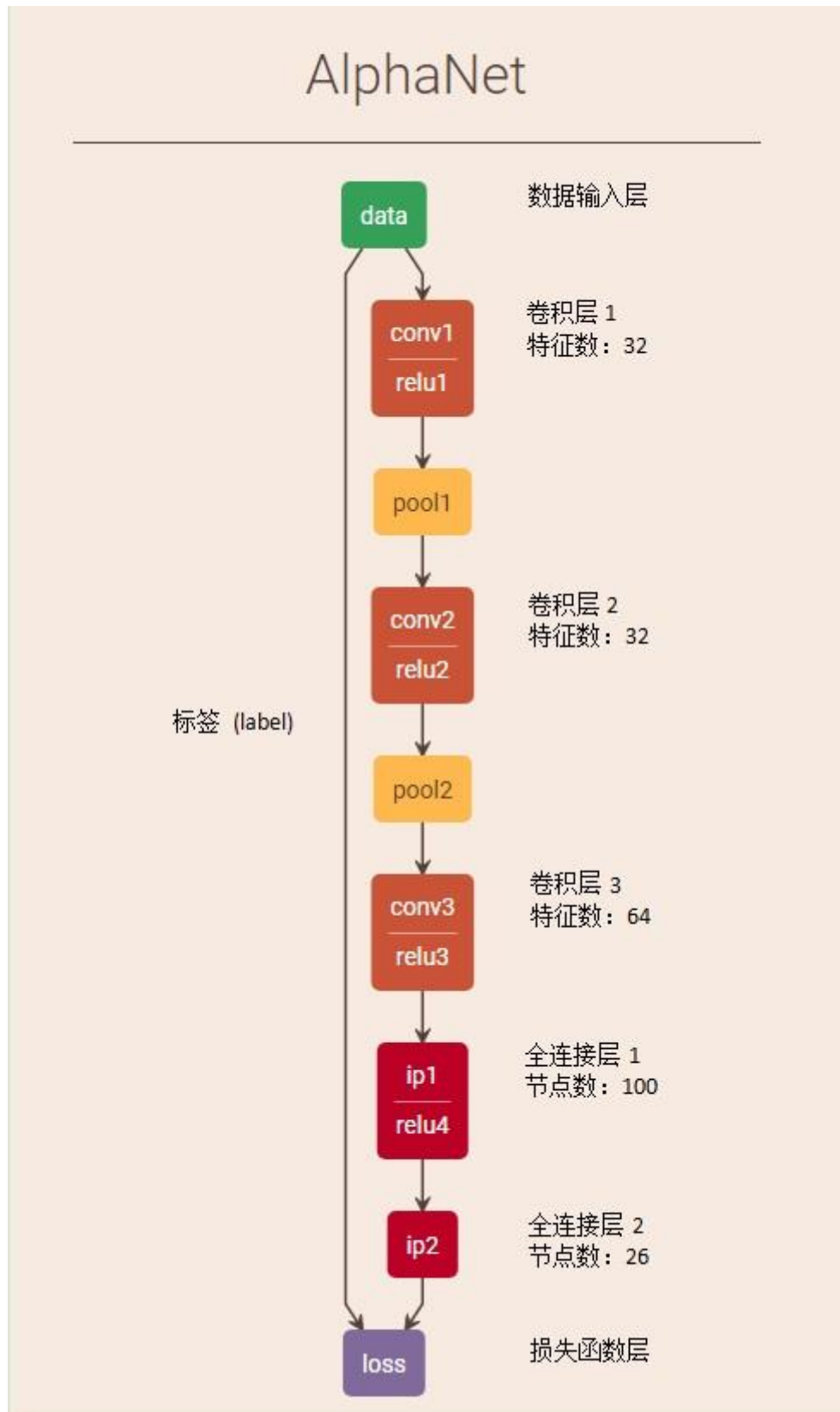
- `python 2.7`
- `caffe 1.0.0` (实验楼环境已经预先安装)

二、实验步骤

2.1 项目的引入 & 网络总体结构

在正式介绍和使用 `caffe` 之前，我们先来大致确定我们的深度神经网络结构。和课程 814 一样，本课程也是识别图片中的英文字母。不过这次我们是使用 `caffe` 来训练我们的模型，并且我们会向网络模型中加入前面介绍的卷积层。我们的网络模型大致是这样的：

AlphaNet



图中有一些你还没见过的东西，不过现在你只需要关注绿色的数据输入层、紫色的损失函数层、三个橙色的卷积层和两个红色的卷积层。

可以看到，我们的神经网络先用三个卷积层提取图片中的特征，然后是一个含有 100 个节点的全连接层 ip1，最后是含有 26 个节点的用于分类的全连接层 ip2 和损失函数层。

和课程 814 相比，我们似乎只是多了卷积层。

2.2 caffe 简要介绍

caffe(Convolutional Architecture for Fast Feature Embedding)和好多软件项目一样，它看起来复杂且有些让人摸不着头脑的英文全称很可能只是为了获得一个有意思的缩写名。所以不要被它的名字迷惑，不过名字里的 **Convolutional** 也暗示了 **caffe** 特别擅长构建卷积神经网络。目前，**caffe** 是使用最多的深度学习框架之一。其作者 **Yangqing Jia**(没错，他是中国人)在 Facebook。**caffe** 支持使用 **CPU** 或 **GPU** 进行训练，当使用 **GPU** 时，能够获得更快的训练速度（实验楼环境中目前没有 GPU 资源可以使用，不过这不影响我们的实验）。

2.3 为 **caffe** 准备训练数据

2.3.1 获取数据

caffe 支持多种训练数据输入方式，其中最常用的一种是先将训练图片存储到 **lmdb** 数据库中，在训练的过程中直接从 **lmdb** 数据库中读取数据。除非你有兴趣，不然你暂时不必关心如何掌握 **lmdb** 数据库，因为 **caffe** 已经为我们提供了将图片转化成 **lmdb** 数据的脚本。

我已经事先准备好了一些训练图片，过为了方便，我把图片的分辨率改成了 16*16。你可以使用以下命令获取这些训练数据：

```
wget http://labfile.oss.aliyuncs.com/courses/820/cnndata.tar.gz
tar zxvf cnndata.tar.gz
```

注：**cnndata.tar.gz** 中也包括之后我们会用到的 **network.prototxt** 和 **solver.prototxt**。解压之后，你当前的目录结构应该如下：

```
.
├── cnndata.tar.gz
├── network.prototxt
├── pic
├── solver.prototxt
├── test.txt
├── train.txt
└── validate.txt

1 directory, 6 files
```

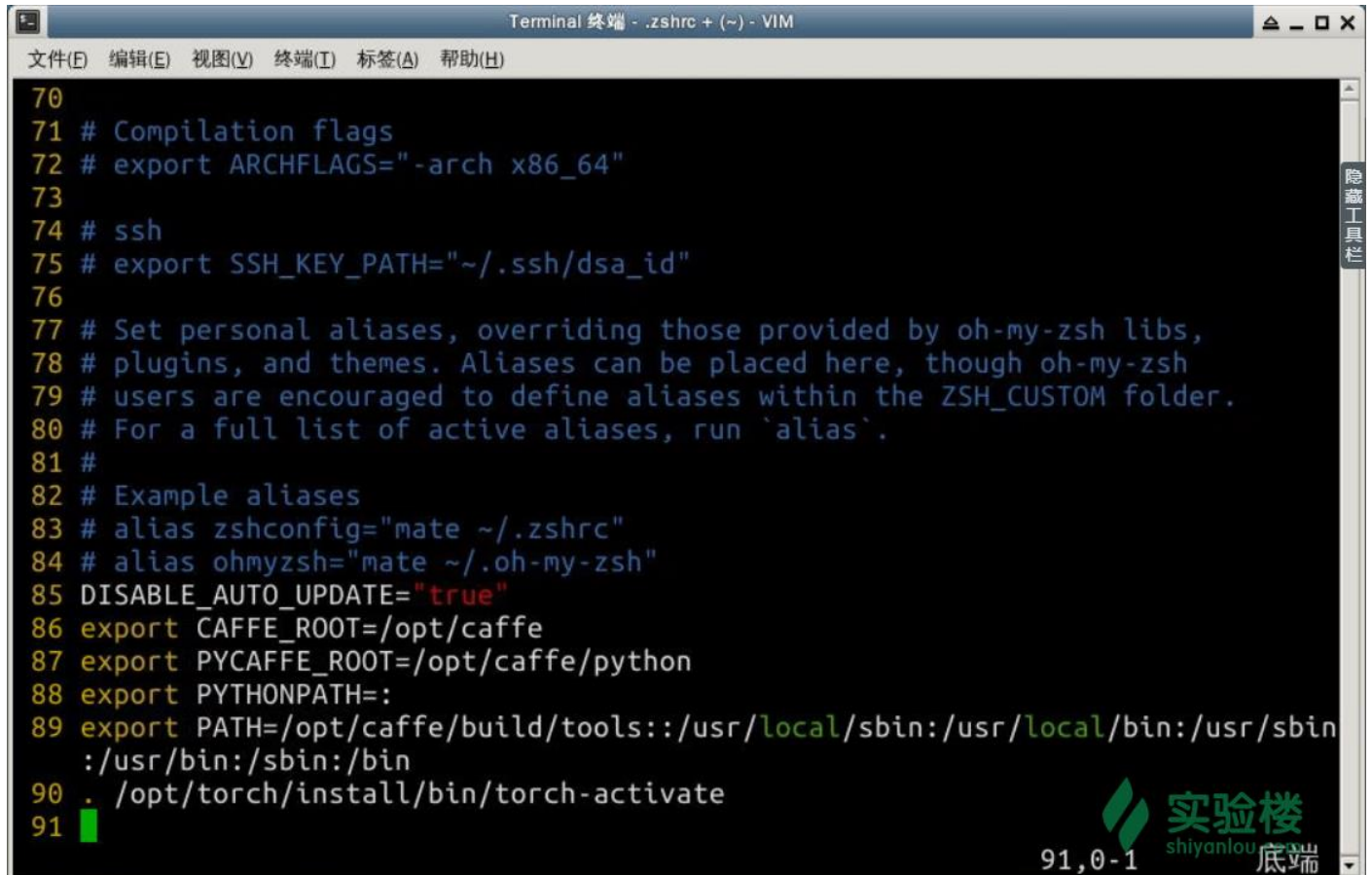
注意之后我们会编写的 **network.prototxt** 和 **solver.prototxt** 也包含在了里面，进行下面的实验时，你可以打开 **network.prototxt** 对照着查看完整的网络定义。

其实这个项目对于 **caffe** 来说有些太简单了，而且也无法完全体现出 **caffe** 的优势，本来我是想做人脸性别识别的项目的，但最后考虑到训练时间的问题，还是决定采用这个更加简单的项目。完成本课程后，你可以自己尝试其他更具有挑战性的项目。

2.3.2 生成 **lmdb** 数据库

实验楼环境中的 `caffe` 安装在 `/opt/caffe` 目录下，在 `/opt/caffe/build/tools` 目录下你可以找到一个名为 `convert_imageset` 的可执行程序，不过实验楼使用的环境，在 `~/.zshrc` 文件中已经将 `convert_imageset` 程序所在的目录添加到了环境变量 `PATH` 中，所以你可以直接在 `terminal` 中输入 `"convert_imageset"` 执行命令。如果你自己的电脑中安装有 `caffe` 而没有配置环境变量 `PATH`，可能无法直接执行 `convert_imageset` 命令。

`~/.zshrc` 文件中，`caffe`（也包括 `torch`）相关环境的配置如下：



The image shows a terminal window titled "Terminal 终端 - .zshrc + (~) - VIM". The window displays the contents of the `~/.zshrc` file, with line numbers 70 through 91. The configuration includes setting compilation flags, SSH keys, aliases, and environment variables for Caffe and Torch. The Caffe environment is configured with `CAFFE_ROOT`, `PYCAFFE_ROOT`, and `PYTHONPATH`. The Torch environment is configured by sourcing the `torch-activate` script. The `PATH` variable is updated to include the Caffe build tools directory. The terminal window has a menu bar with options like "文件(E)", "编辑(E)", "视图(V)", "终端(T)", "标签(A)", and "帮助(H)". On the right side, there is a vertical toolbar with icons for window management and a "隐藏工具栏" (Hide Toolbar) button. In the bottom right corner, there is a logo for "实验楼" (Shiyanlou) with the text "91,0-1" and "shiyancelou 底端".

```
70
71 # Compilation flags
72 # export ARCHFLAGS="-arch x86_64"
73
74 # ssh
75 # export SSH_KEY_PATH="~/.ssh/dsa_id"
76
77 # Set personal aliases, overriding those provided by oh-my-zsh libs,
78 # plugins, and themes. Aliases can be placed here, though oh-my-zsh
79 # users are encouraged to define aliases within the ZSH_CUSTOM folder.
80 # For a full list of active aliases, run `alias`.
81 #
82 # Example aliases
83 # alias zshconfig="mate ~/.zshrc"
84 # alias ohmyzsh="mate ~/.oh-my-zsh"
85 DISABLE_AUTO_UPDATE="true"
86 export CAFFE_ROOT=/opt/caffe
87 export PYCAFFE_ROOT=/opt/caffe/python
88 export PYTHONPATH=:
89 export PATH=/opt/caffe/build/tools::usr/local/sbin:usr/local/bin:usr/sbin
90 :usr/bin:/sbin:/bin
91 . /opt/torch/install/bin/torch-activate
```

输入“convert_imageset”命令，你可以看到这个命令有哪些选项：

```
Terminal 终端 - shiyanlou@85067e46795f: ~
文件(F) 编辑(E) 视图(V) 终端(T) 标签(A) 帮助(H)
shiyanlou:~/ $ convert_imageset [13:44:08]
convert_imageset: Convert a set of images to the leveldb/lmdb
format used as input for Caffe.
Usage:
  convert_imageset [FLAGS] ROOTFOLDER/ LISTFILE DB_NAME
The ImageNet dataset for the training demo is at
http://www.image-net.org/download-images

Flags from tools/convert_imageset.cpp:
  -backend (The backend {lmdb, leveldb} for storing the result) type: string
    default: "lmdb"
  -check_size (When this option is on, check that all the datum have the same
    size) type: bool default: false
  -encode_type (Optional: What type should we encode the image as
    ('png','jpg',...)) type: string default: ""
  -encoded (When this option is on, the encoded image will be save in datum)
    type: bool default: false
  -gray (When this option is on, treat images as grayscale ones) type: bool
    default: false
  -resize_height (Height images are resized to) type: int32 default: 0
  -resize_width (Width images are resized to) type: int32 default: 0
  -shuffle (Randomly shuffle the order of images and their labels) type: bool
    default: false
shiyanlou:~/ $
```

你可以自己研究如何利用这些选项来自定义训练数据的生成过程，对于我们的训练数据，可以直接输入以下命令将图片转化成 `lmdb` 数据：

```
convert_imageset --check_size --gray -shuffle ./ train.txt train
convert_imageset --check_size --gray -shuffle ./ validate.txt validate
convert_imageset --check_size --gray -shuffle ./ test.txt test
```

这三条命令的第一个参数 `--check_size` 检查每一张图片的尺寸是否相同。第二个参数 `--gray` 将图片转换为单通道灰度图片。第三个参数 `-shuffle` 将所有图片的顺序打乱。第三个参数 `./` 指明图片文件所在的父目录，由于这里的 `train.txt` 等文件中已经包含了前缀 `pic`，所以这里的父目录就是当前目录 `./`。第四个参数指明图片列表文件。第五个参数指明最后生成的 `lmdb` 数据库文件夹的位置。

这三条命令执行完毕后，你的目录结构应该是这样的：

```
.
├── cnndata.tar.gz
├── network.prototxt
├── pic
└── solver.prototxt
```

```
|— test
|— test.txt
|— train
|— train.txt
|— validate
|— validate.txt
```

```
4 directories, 6 files
```

多出的三个文件夹 `train` `validate` `test` 包含我们生成的 `lmdb` 数据库。

2.3.3 计算图片像素均值

对于神经网络，我们希望输入的数据分布能够有正有负（具体原因这里不赘述，若有兴趣你可以查阅资料了解为什么希望数据有正有负）。而图片像素值都是大于 0 的，所以 `caffe` 为我们提供了另一个脚本 `compute_image_mean`，运行这个命令获得训练数据在每个通道上的均值，在处理训练数据时减去这个均值就可以保证图片有正有负且其分布“以 0 为中心”。

需要注意的是，我们只对训练集 `train` 计算均值，训练和测试的时候都是减去这个均值，而不是对于测试集单独计算。因为如果对于训练集和测试集的预处理操作不一样的话，可能会影响模型在测试集上的实际效果。

运行以下命令计算训练集图片均值：

```
compute_image_mean train train.binaryproto
```

```
shiyanolou:~/ $ compute_image_mean train train.binaryproto [13:51:21]
I0518 13:51:54.201035 390 db_lmdb.cpp:35] Opened lmdb train
I0518 13:51:54.204975 390 compute_image_mean.cpp:70] Starting iteration
I0518 13:51:54.225471 390 compute_image_mean.cpp:95] Processed 10000 files.
I0518 13:51:54.248494 390 compute_image_mean.cpp:95] Processed 20000 files.
I0518 13:51:54.289822 390 compute_image_mean.cpp:95] Processed 30000 files.
I0518 13:51:54.307072 390 compute_image_mean.cpp:101] Processed 38000 files.
I0518 13:51:54.309870 390 compute_image_mean.cpp:108] Write to train.binaryproto
I0518 13:51:54.310138 390 compute_image_mean.cpp:114] Number of channels: 1
I0518 13:51:54.310215 390 compute_image_mean.cpp:119] mean_value channel [0]:
59.1442
```

其中第一个参数 `train` 指定对在我们刚生成的 `lmdb` 数据库 `train` 中的数据计算均值，第二个参数 `train.binaryproto` 指定计算出的均值保存在 `train.binaryproto` 文件中。稍后我们会用到这个均值文件。

2.4 caffe 如何定义网络结构

就像课程 814 中所说的，好多深度学习框架都提供层次化的网络结构，网络结构中的每一层为一个小的模块，将多个模块组合起来，就构成了一个神经网络模型，就像是搭积木一样。`caffe` 正是如此。

2.4.1 通过 `protobuf` 文件定义网络结构

`caffe` 通过编写 `protobuf` 文件来定义网络(network)结构，`protobuf` 是一种数据交换格式。`protobuf` 使用起来很简单，为了完成本课程的实验，你不需要专门去学习 `protobuf`，只需要参照别人写好的 `protobuf` 文件照葫芦画瓢就行了。

使用编辑器创建一个 `network.prototxt`(注意这里的后缀名是 `prototxt`)文件，这里使用的是 `vim` 编辑器，如果你不会使用 `vim`，也可以使用其他编辑器。

```
vim network.prototxt
```

在新建文件的第一行，你可以定义网络模型的名字：

```
name: "AlphaNet"
```

2.4.2 Blobs、bottom、top

在课程 814 中，我们直接使用 `numpy ndarray` 存储网络中的数据。在 `caffe` 中，网络中的数据使用 `Blobs` 存储，其实你可以直接把 `Blobs` 看成一个 `n*c*h*w` 的四维数组，其中 `n` 代表一个 `batch` 中图片的数量，`c` 代表图片通道数（对于卷积层，代表特征个数），`h` 代表图片高度，`w` 代表图片宽度。

`caffe` 中的每一个网络层可以有多个 `bottom` 和 `top`，`bottom` 其实就是一个网络层的数据流入口，`top` 就是一个网络层的数据流出口。当层 A 的 `bottom` 和层 B 的 `top` 相同时，就代表层 A 以层 B 的输出作为输入。

2.4.3 定义数据输入层

我们已经准备好了训练和测试数据，为了让我们的卷积神经网络能够读取这些数据，需要在 `network.prototxt` 添加数据输入层。

数据层需要从 `lmdb` 中读取数据，然后产生两个输出 `top`：一个 `data` 代表图片数据，一个 `label` 代表该图片的标签。

```
layer{  
  name:"data"  
  type:"Data"  
  top:"data"  
  top:"label"  
  data_param{  
    source: "train"
```



```

        batch_size:16
        backend:LMDB
    }
    transform_param{
        scale: 0.00390625
        mean_file: "train.binaryproto"
    }
    include{
        phase:TRAIN
    }
}

```

我们逐个对这个数据层的各部分进行解释。

`name` 代表这一层的名字。`type` 代表类型，`caffe` 中提供了很多种类型的网络层，你可以到[这里](#)查看有哪些层，这里的 `Data` 指定是从数据库中读取数据。

两个 `top` 代表了数据层有两个输出，一个是 `data` (与层的名字相同)，代表 `lmdb` 数据库中的图片数据，一个是 `label` 代表每张图片对应的标签。

`data_param` 中的内容指定了 `Data` 类型数据层需要的参数，其中 `source` 指定数据库的位置，在这里就是我们之前生成的 `train` 文件夹；`batch_size` 指定每一个 `batch` 一次性处理多少张图片（还记得课程 814 里说的 `batch` 吗）；`backend` 指定数据库的种类，我们使用的是 `lmdb` 数据库，所以这里为 `LMDB`。

接下来的 `transform_param` 指定对图片进行的预处理操作，这里的 `mean_file` 指定平均值文件的位置，同时，我们指定了对图片像素值的缩放比例 `scale` 为 0.00390625（其实就是 $1/255$ ），将像素值的范围缩小到 1。

`include` 包含了其他一些信息，这里的 `phase` 指定这个数据层是在训练还是在测试阶段使用，`TRAIN` 表明是在训练阶段。

对于测试阶段的数据，可以直接再增加一个数据层，同时设置 `phase` 为 `TEST` 就可以了，如下：

```

layer{
    name:"data"
    type:"Data"
    top:"data"
    top:"label"
    data_param{
        source: "validate"
        batch_size:100
        backend:LMDB
    }
}

```

```

    }

    transform_param{
        scale: 0.00390625
        mean_file: "train.binaryproto"
    }

    include{
        phase:TEST
    }
}

```

当进行训练时，`caffe` 就调用 `phase` 为 `TRAIN` 的数据层，当测试时，`caffe` 就调用 `phase` 为 `TEST` 的数据层。

除了 `phase`、`source` 和 `batch_size`，第二个数据层的设置与第一个数据层一模一样。注意这里的两个 `top` 名必须和第一个数据层一样，因为后面的网络层的输入 `bottom` 通过名称指定数据来源，所以两个数据层的输出 `top` 名设置成一样就可以保证在训练和测试时，后面的网络层都能读取到数据。

2.4.4 定义卷积层

我们一共有三个卷积层，让我们先来看看第一个卷积层的定义：

```

layer{
    name: "conv1"
    type: "Convolution"
    bottom: "data"
    top: "conv1"
    param{
        lr_mult: 1
    }
    param{
        lr_mult: 2
    }
    convolution_param{
        num_output: 32
        kernel_size: 3
        stride: 1
        pad: 1
    }
}

```



```

weight_filler{
    type: "xavier"
}
bias_filler{
    type: "constant"
}
}
}
}

```

和数据层有一些类似（比如 name, bottom, top 的作用），但又有很多不一样的地方。首先这里的 `type` 变成了 `Convolution` 代表这一层是卷积层。

两个 `param` 中的 `lr_mult` 作用有些特殊，它们代表卷积层中对参数的学习速率乘以多少倍（就像它的名字暗示的那样--learning rate multiply），其中第一个 `lr_mult` 代表对卷积核中到的参数 `weight` 学习速率相乘的值，第二个 `lr_mult` 代表对偏移量 `bias` 学习速率相乘的值，一般我们总是把第一个设置为 1（即学习速率不变），第二个设置为 2。这里你可能对这两个参数的作用感到摸不着头脑，但以后你就会明白这两个参数非常有用（它们在 `caffe` 的迁移学习 transfer learning 中发挥作用）。

数据层中的参数在 `data_param` 中定义，类似的，卷积层中的参数在 `convolution_param` 中定义。这里 `kernel_size` `stride` `pad` 你已经知道它们的作用了，分别代表卷积核的尺寸、卷积核移动步长、图片边缘填充的像素数。而 `num_output` 其实就是我们第一次实验所说的特征个数 `feature`。

最后剩下 `weight_filler` 和 `bias_filler`，这两个参数指明 `weight` 和 `bias` 使用什么方式初始化（填充），如果 `type` 为 `constant`，代表用常数 0 填充，而 `xavier` 所代表的填充算法就稍微有点复杂了。这里对 `xavier` 填充算法不做介绍，如果你有兴趣，可以自己查阅资料或者查看提出 `xavier` 算法的[论文](#)

这里需要注意的是，我们设置 `kernel_size=3` `stride=1` `pad=1` 可以保证卷积层输出的宽和高与输入相同，你可以代入第一次实验给出的公式计算验证。我们的输入图片尺寸为 16x16，带入公式和卷积层的参数，得到： $(16+2*1-3)/1+1=16$ 。

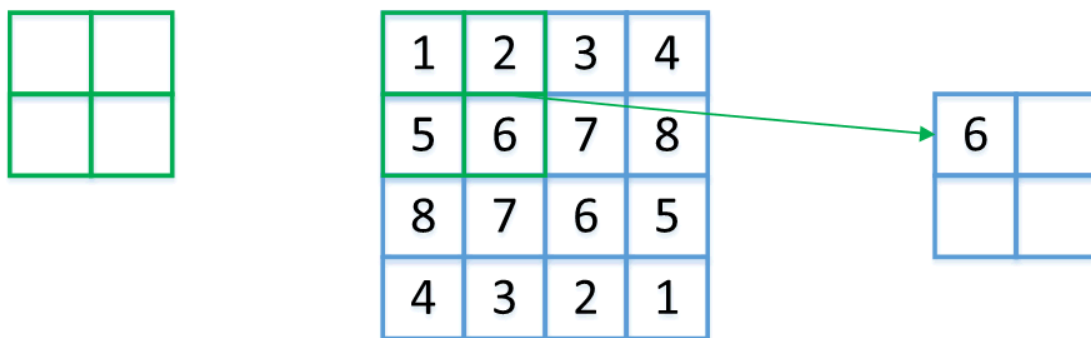
卷积层 2 和卷积层 3 的定义与卷积层 1 几乎一模一样，除了卷积层的 `num_output` 参数被设置成 64。

2.4.5 卷积神经网络中的池化层

池化(Pooling)层是卷积神经网络中几乎必然出现的网络层，第一次实验为了突出卷积层，没有介绍池化层，放到了这里来介绍。

我们之前说过，合理的设置卷积层的参数，可以保证卷积层的输出和输入在宽和高上不变。但我们有时候又希望能减小训练数据的尺寸，这样可以降低模型的复杂度，减少参数的数量，让模型训练的更快，**池化层**就具有这样的作用。

池化层其实和卷积层非常相似，也是有一个“池化核”对整张图片中的所有可能位置进行计算，不同的是，池化层中没有参数，一般池化层会返回“池化核”中最大（或者最小，或者随机）的数字，且池化层的步长 `stride` 一般设置成与“池化核”的尺寸相同，返回“池化核”内最大数值的池化层效果如下：



“池化核” (2x2)
步长也为2，返回最大值

输入图片尺寸(4x4)

池化结果(2x2)

在第一个卷积层之后，就有一个池化层。

```
layer{
  name:"pool1"
  type: "Pooling"
  bottom: "conv1"
  top: "pool1"
  pooling_param{
    pool: MAX
    kernel_size:2
    stride: 2
  }
}
```

其中参数的作用已经很明显了，注意 `pool` 设置为 `MAX` 代表这个“池化核”返回最大值。

注意

池化层的作用不止是降低模型复杂度，比如你可以把返回最大值池化层理解为只保留一个区域最明显的特征。如果你想弄清楚池化层更深层次的作用，请自行查阅资料理解。

2.4.5 caffe 中的内积层

`caffe` 将全连接层称为内积层(`InnerProduct`)，其计算方式大体上与课程 814 中的全连接层一样，内积层的定义如下：

```
layer{
```

```

name: "ip1"
type: "InnerProduct"
bottom: "conv3"
top: "ip1"
param{
  lr_mult:1
}
param{
  lr_mult:2
}
inner_product_param{
  num_output: 100
  weight_filler{
    type: "xavier"
  }
  bias_filler{
    type: "constant"
  }
}
}
}

```

其中的参数我们都已经见过，注意这里的 `type` 为 `InnerProduct` 代表是全连接层，`num_output` 代表的是全连接层的输出节点的数量。

这里我们可以感受到深度学习框架带给我们的便利，课程 814 中我们为了实现全连接层绞尽脑汁，而这里只需要十几行的定义就可以了，其他一切都由 `caffe` 帮我们实现。

2.4.5 损失函数层

课程 814 中，我们介绍了 `QuadraticLoss` 和 `CrossEntropyLoss` 两种损失函数，但这里我们打算使用 `caffe` 提供的另一种损失函数：`SoftmaxWithLoss`。为了弄清楚 `SoftmaxWithLoss` 损失函数是如何工作的，我们先要介绍什么是 `Softmax`，`Softmax` 函数的表达式如下：

$$X=(x_1, x_2, \dots, x_n)$$

$$\text{Softmax}(x_i)=\frac{e^{x_i}}{\sum_1^n e^{x_j}}$$

不要被它的表达式吓到，其实 **Softmax** 的计算很简单，就是对一个数组中的每一个元素先求它对于自然数 e 的指数 e^x ，然后每一个元素的 **Softmax** 函数值就是 e^{x_i} 除以所有元素对自然数 e 的指数的和。

Softmax 函数的性质也很好分析，数组中原本最大的数的函数值最接近 1，最小的数的函数值最接近 0，同时，数组中所有元素的 **Softmax** 函数值加起来为 1。这刚好可以作为概率来看待，实际上，**caffe** 中有单独的 **Softmax** 层存在，我们可以直接用 **Softmax** 层的输出作为我们的模型对每个类别（比如这里是 26 个类别）的概率值的预测。

现在我们清楚了 **Softmax** 的作用，但你可能仍然会困惑，**Softmax** 的名字从何而来，为什么把它叫做“软的最大”呢？其实，与 **Softmax** 对应的还有一种“硬的最大”函数，这里我把它叫做 **Hardmax**，它的表达式如下：

$$\text{Hardmax}(x_i) = \frac{x_i - \min(X)}{\max(X) - \min(X)}$$

观察它的表达式，你会发现 **Hardmax** 与 **Softmax** 的性质非常相似，都是数组中原本最大的元素的函数值最大，但不同的是，对于 **Hardmax**，数组中最大的元素的函数值固定为 1，最小的元素的函数值固定为 0。最大元素的函数值一定为 1，所以我把它称为“硬最大”，而 **Softmax** 却相对更加“柔和”，更加的“软”。**Softmax** 的这种特性恰恰适合于作为神经网络中的概率输出，而 **Hardmax** 则会总是把最大可能的类别概率值设置为 1，这是不合理的。

搞清楚了 **Softmax** 的作用，理解 **SoftmaxWithLoss** 损失函数就非常简单了。**caffe** 中的 **SoftmaxWithLoss** 损失函数层其实就是在 **Softmax** 层上增加了一些运算，**SoftmaxWithLoss** 层的定义如下：

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

SoftmaxWithLoss 层有两个 **bottom**，一个 **ip2** 是我们模型对于每种类别可能性大小的预测，注意这个预测值在经过 **Softmax** 层之前是不能作为概率值的（可能有负值，和可能不为 1）。另一个 **label** 就是我们数据层的输出 **label**，代表一个训练图片上实际英文字母的类别。

SoftmaxWithLoss 层先用 **Softmax** 函数计算出模型对每种类别的预测概率。再根据 **label** 的值，选择出预测值中的对应概率。比如 **Softmax** 的输出在这里是一个长度为 26 的数组，而 **label** 中的值为 0（代表图片上的字母实际为 A），就选择出 **Softmax** 函数输出数组中的第一个概率值。显然，当这个概率值接近 1 的时候，说明我们的模型预测的比较准确，**SoftmaxWithLoss** 的输出值应该接近于 0，当这个概率值接近于 0 的时候，说明我们的模型预测的不太准确，**SoftmaxWithLoss** 的输出值应该是一个很大的正值。

实际上，`SoftmaxWithLoss` 层对概率值进行的运算很简单，就是对该值求负对数，这样就满足了上面说的，预测越准损失函数值越小的要求。即 `SoftmaxWithLoss` 层的运算大致如下：

$$\text{SoftmaxWithLoss}(X, \text{label}) = -\log(\text{Softmax}(X)_{\text{label}})$$

2.4.6 准确率层

之前我们说过，为了检验模型的泛化性能，需要在验证/测试集上检验模型的预测准确率。`caffe` 为我们提供了 `Accuracy` 准确率层，其定义如下：

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

其 `bottom` 同样有两个，但注意这里的第一个 `bottom` 是 `ip2` 而不需要是概率值，因为 `Accuracy` 层只需要第一个 `bottom` 的最大值的下标与第二个 `bottom` 相同就认为预测是准确的。

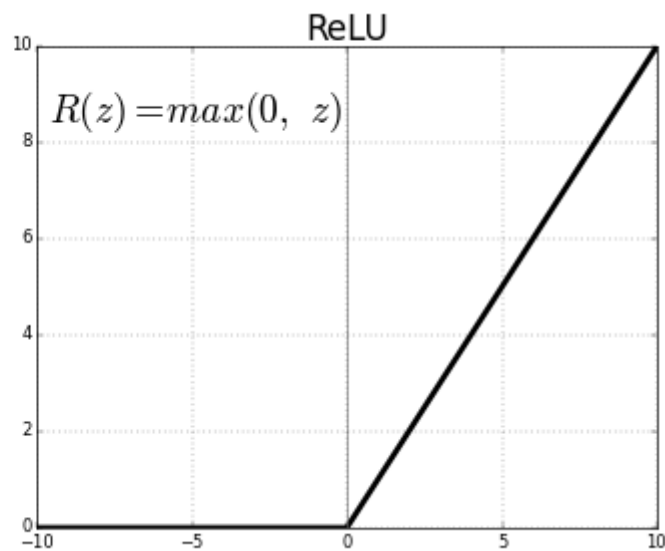
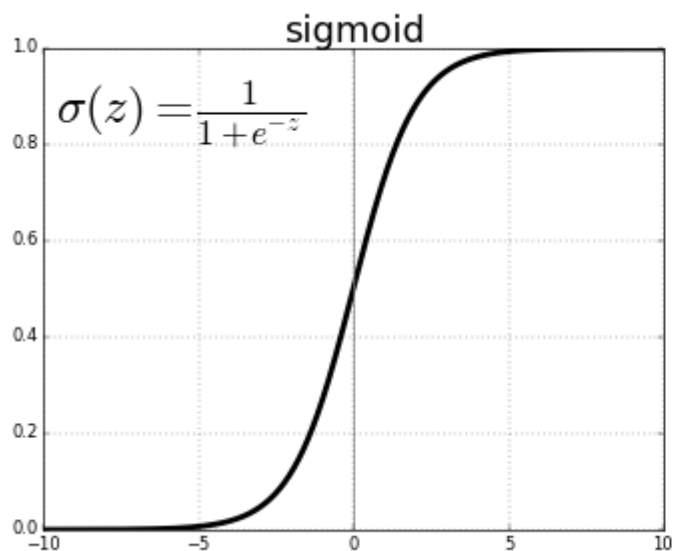
2.4.7 ReLU 激活函数层

在课程 814 中我们说过，`Sigmoid` 激活函数容易导致梯度消失问题，消失的梯度使得神经网络的训练变得非常困难。而这里我们将介绍的 `ReLU(Rectified Linear Unit)` 激活函数层则非常好的避免了梯度消失问题。

`ReLU` 的函数表达式如下：

$$\text{ReLU}(x) = \begin{cases} x & x > 0 \\ 0 & x \leq 0 \end{cases}$$

`Sigmoid` 和 `ReLU` 函数图像对比如下：



ReLU 执行的运算非常简单，就是只让大于 0 的节点的值向前传递。你需要特别注意的是，当反向传播梯度的时候，大于 0 的节点的梯度是由之前的“部分梯度”乘以 1 得到的，而小于等于 0 的节点的梯度则为 0。对于大于 0 的节点，**ReLU** 不会导致梯度值减小，非常有效的避免了梯度消失问题。

同时，**ReLU** 的计算非常简单，而 **Sigmoid** 涉及到的求指数运算对于计算机来说则非常复杂，**ReLU** 激活函数具有更高的执行速度。

我们的网络中层与层之间都存在着 **ReLU** 激活函数层，其定义如下：

```
layer{
  name: "relu1"
  type: "ReLU"
  bottom: "conv1"
  top: "conv1"
}
```

ReLU 不需要参数，所以这里的定义非常简单，不过你需要注意的是，这里的 **bottom** 和 **top** 名字可以设置成一样的，当设置成一样的时候，**ReLU** 的输出结果会保存到输入的 **Blobs** 里，这样能节省显存（或内存）。

2.5 caffe 网络定义总结

至此我们的项目中会用到的各种网络层都已经介绍过了，完整的网络定义请你打开 **network.prototxt** 查看。我建议你仔细一行一行的查看这个文件，理解每个网络层的功能和它的特性，理解每一个参数的作用。这对你之后自己动手编写神经网络定义文件非常重要。

我们现在只写好了网络定义文件，但为了让模型开始训练，我们还有一些东西没有确定，比如超参数学习速率、测试间隔、最大训练周期(epoch)等，下次实验，我们将讲解如何编写 **solver.prototxt** 文件。

三、实验总结

虽然看起来我们的网络定义文件 `network.prototxt` 的行数有点多，但和我们之前自己动手实现神经网络比起来，这里的网络模型构建还是简单多了，熟练之后你可以在几分钟之内就搭建好一个神经网络。`caffe` 还提供了很多其他种类的网络层，如果你有兴趣可以到 [caffe 官网](#) 查看。理解这些层的原理是科学合理地使用这些网络层的基础。

本次实验，我们学习了：

- `caffe` 中的数据由 Blobs 承载，Blobs 可以被看成一个四维数组（或者四维张量）。
- `caffe` 通过编写 `network.prototxt` 文件构建神经网络。
- 池化层能够缩小图片尺寸，降低模型计算量。
- `SoftmaxWithLoss` 损失函数就是对概率值取负对数。
- `ReLU` 激活函数可以有效避免梯度消失。

四、课后作业

1. [选做] 如果你打算深入研究 `caffe`，可能会发现 `caffe` 官网的文档并不是十分全面，你可以查看 `/opt/caffe/src/caffe/proto/caffe.proto` 文件，里面包含了 `caffe` 中所有参数的定义。

控制 `caffe` 模型的训练过程

一、实验介绍

1.1 实验内容

上次实验，我们已经构建好了卷积神经网络，我们的模型已经蓄势待发，准备接受训练了。为了控制训练进程，记录训练过程中的各种数据，`caffe` 还需要定义另一个 `solver.prototxt` 文件，这次实验我们就来完成它，并开始激动人心的训练过程。

1.2 实验知识点

- 可变的学习速率
- 正则化

1.3 实验环境

- `caffe 1.0.0`

二、实验步骤

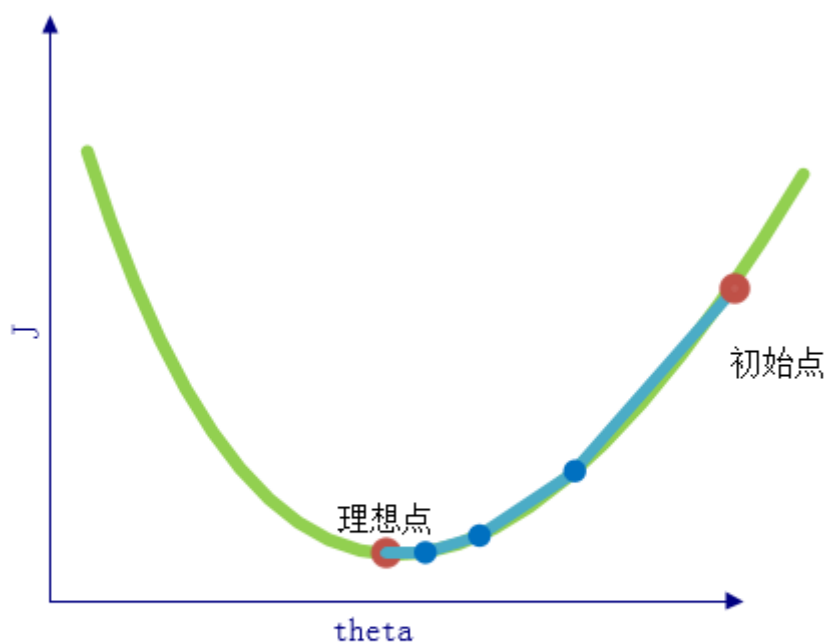
2.1 指定网络定义文件

在 `solver.prototxt` 中，我们需要先指定网络定义文件的位置，我们通过下面的语句指定：

```
net: "network.prototxt"
```

2.2 可变的学习速率

在课程 814 中，有这么一张图：



学习速率适中

当学习速率固定时，每次参数更新的“步长”会慢慢变短。我将为什么步长会变短作为一个选做课后作业留在了那里，其实不难思考，因为学习速率固定，而损失函数图形的“倾斜”程度一直在变小，即损失函数对参数的梯度的数值一直在变小，所以最后更新的“步长”会越来越短。

这个特性有利于我们的模型训练过程，因为越接近损失函数最低点我们希望更新的步长越小，这样才能使参数更逼近最低点。但为了保证每次更新的步长越来越小，也可以在训练的过程中减小学习速率的数值，在 `solver.prototxt` 中可以像下面这样定义：

```
base_lr: 0.001
```

```
lr_policy: "step"
```

```
stepsize: 4000
```

```
gamma: 0.1
```

```
max_iter: 10000
```

其中 `base_lr` 指定在开始训练时的学习速率，这里设置为 0.001，`lr_policy` 设置为 `step` 和 `step_size` 设置为 4000 就指定了学习速率每隔 4000 次训练周期(epoch)就自动减小。而 `gamma` 的数值就是每次减小学习速率时乘以的数值，这里设置为 0.1 就代表每次将学习速率减小到原来的十分之一。

`max_iter` 指定训练的最大迭代周期数，这里设置成 10000 次。

2.3 测试周期

在前一次实验中，我们的测试数据层（`phase` 设置为 TEST 的数据层）中的 `batch_size` 被设置成了 100，而我们的测试数据总共有 10000 张图片，为了每次测试将所有图片都测试一次，这里需要如下设置：

```
test_iter: 100
```

```
test_interval: 500
```

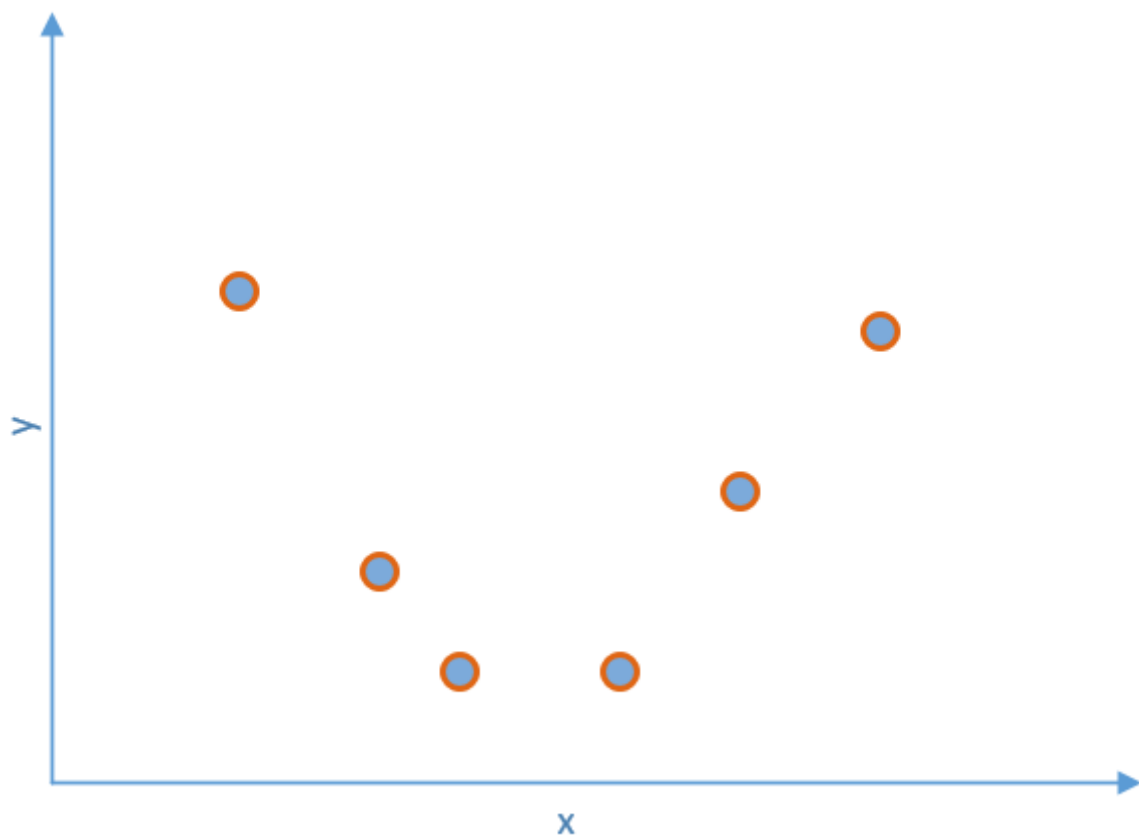
`test_iter` 为 100 即测试的迭代次数为 100 次，这样 100×100 等于 10000，刚好把所有图片都测试一次。

同时，这里设置 `test_interval` 为 500 表明每训练 500 个周期执行一次测试。

2.4 正则化

其实早在课程 814 中，我就非常想讲解正则化，但是一直担心一次性介绍太多内容会让人难以消化，这里我终于迎来了必须讲解正则化的机会。

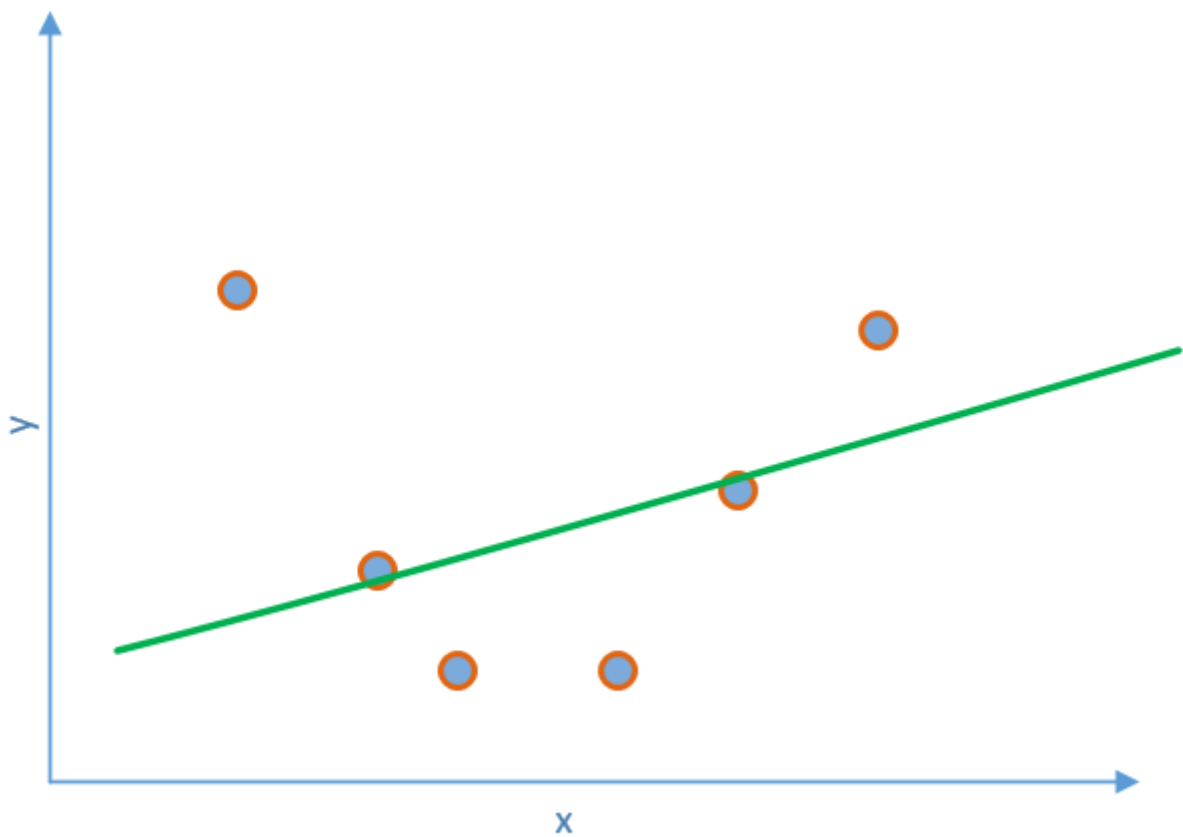
为了理解正则化在深度学习中的作用，我们以回归问题为例讲解。



如图，假设我们现在要根据图中的六个点拟合出数据的分布曲线，用于预测其他 x 坐标对应的 y 值，我们使用如下的多项式进行拟合：

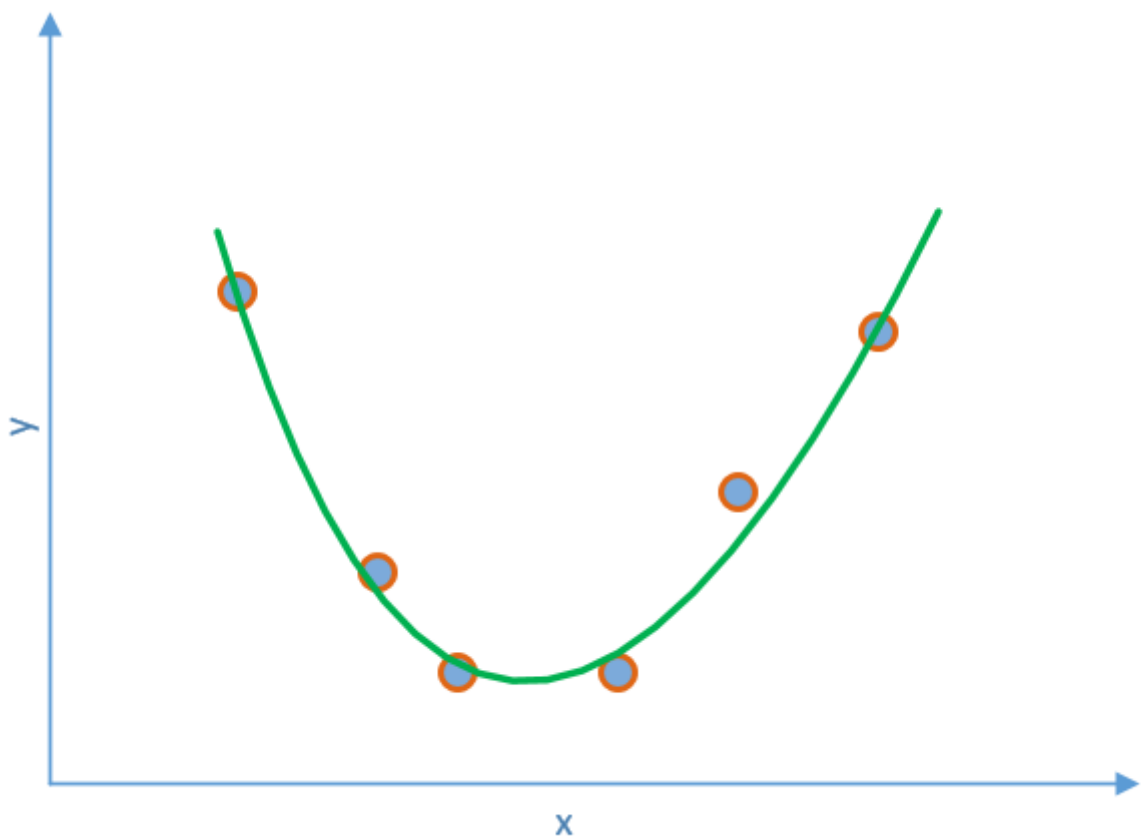
$$y = a_0 + a_1x + a_2x^2 + a_3x^3 + \dots$$

假如一开始，只有 a_0 和 a_1 不为 0，其他系数 a 都为 0，拟合函数就变成了： $y = a_0 + a_1x$ ，拟合曲线如下：



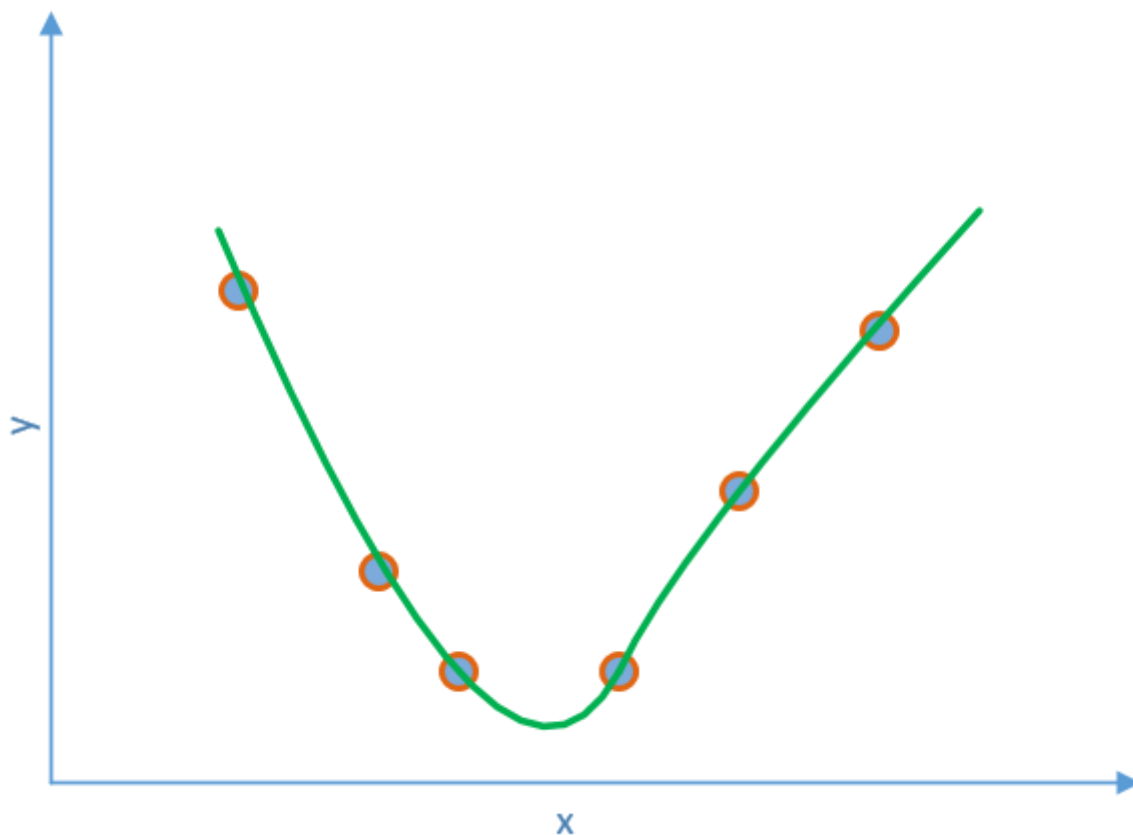
由于此时的系数比较少，曲线不够灵活，所以此时拟合的误差较大，损失函数较大。

如果再增加一个不为 0 的系数 a_2 ，拟合函数变成了： $y=a_0+a_1x+a_2x^2$ ，拟合曲线如下：



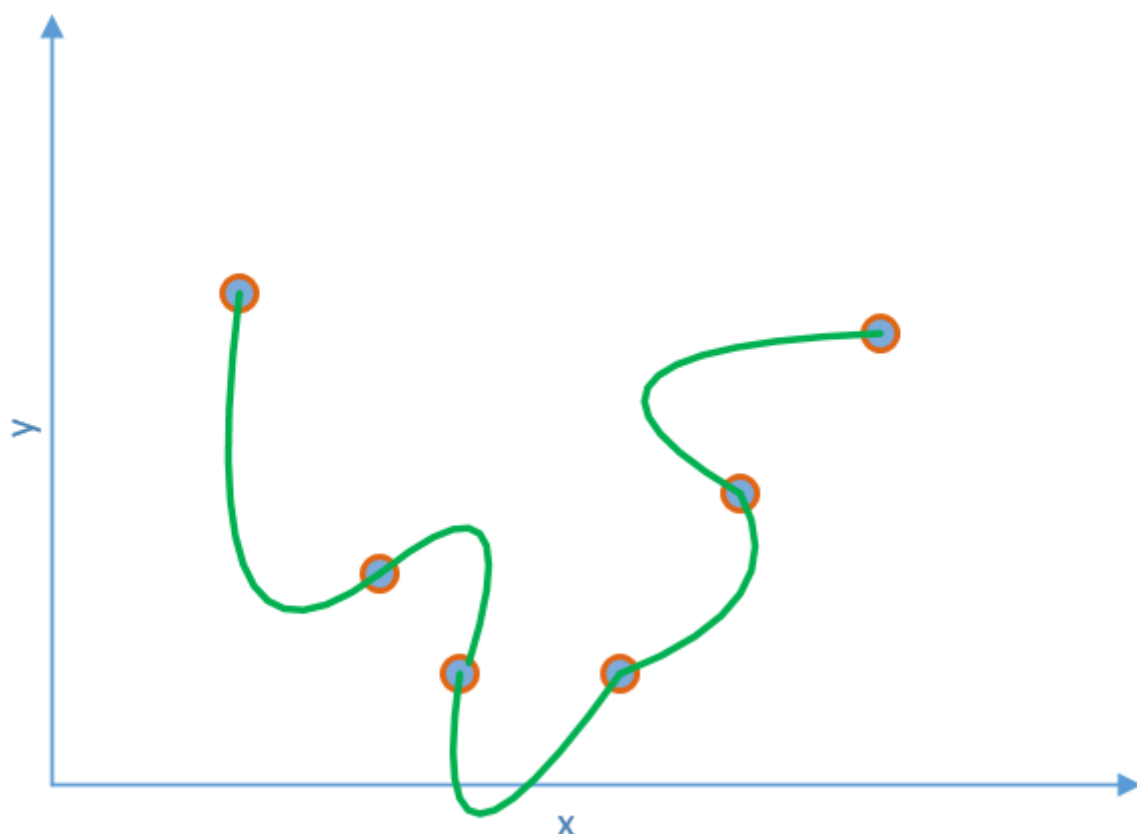
这时的拟合效果已经非常好了，但注意仍然有一些数据点的中心不在拟合曲线上，即损失函数值大于 0。

假如此时再增加一个不为 0 的系数 a_3 ，拟合函数变成了： $y=a_0+a_1*x+a_2*x^2+a_3*x^3$ ，注意每次增加一个不为 0 的系数，相当于是拟合函数变得更加的灵活。这时拟合曲线可能如下：



此时拟合函数经过每一个数据点的中心，损失函数为 0。但拟合函数的形状已经有些奇怪了。

如果我们继续增加更多的不为 0 的系数，拟合函数曲线甚至可能变成这样：



拟合函数非常精确的经过了每一个数据点且此时的损失函数值为 0，但使用这个拟合函数来预测新的 x 点对应的 y 的值可能不会取得非常好的结果。联想之前我们学过的泛化性能，这里的泛化性能会非常差。或者说，这里出现了过拟合 (overfitting)。

我们的深度神经网络模型就可能会出现这个问题，虽然在训练集上的损失函数值已经非常低，甚至为 0，但可能仍然无法在验证/测试集上获得很好的泛化性能。

为了避免过拟合，就需要我们这里的正则化 (regularization)，在 solver.prototxt 里像下面这样设置正则化参数：

```
regularization_type: "L2"
```

```
weight_decay:0.0001
```

那么具体正则化是如何防止过拟合发生的呢？实际上，这里的正则化，是通过在前向计算的过程中，将网络中所有的参数的平方与 `weight_decay` 相乘，再加入到损失函数值上；而反向传递梯度时，则仍然根据链式法则对参数进行更新。

比如这里的 `weight_decay` 为 0.0001，假设只有一个参数 $a=3$ ，则损失函数值计算时就变成了： $L=L_0+0.0001*3^2=L_0+0.0009$ ，这里的 L_0 是不添加正则化时的损失函数值。反向传递梯度时，对参数的更新就变成了 $a=a-lr*(d_0+2*0.0001*3)$ ，其中 d_0 是不添加正则化时的梯度值，而 $2*0.0001*3$ 是误差函数中的正则化项对参数求导（梯度）的结果。实际上，由于这里的 a 值就是 3，之前的式子可以直接改成： $a=(1-lr*2*0.0001)*a-lr*d_0$ 。合理的设置学习速率 `lr` 和正则化参数 `weight_decay` 的值，可以使 $(1-lr*2*weight_decay)$ 的值小于 1，这样的效果实际上是使得参数 a 值每次都以一定的比例缩小，防止参数变得过大。这样可以在一定程度上使高次项的系数变小，从而防止高次项对整个模型的影响过大。从而最终达到防止过拟合的目的。

正则化在这里属于稍难的内容，如果你不需要非常深刻的理解深度学习而只是想能够实际上手，可以暂时不去深究正则化的原理，只需要记住 `caffe` 里的这两个参数是用来进行正则化，从而提高模型效果就行了。

`regularization_type` 设置为 `L2` 就是代表对损失函数加上参数的平方项，还可以将其设置为 `L1`，这样加上的就是参数的绝对值。（实际上，这里的 `L2` 代表的是 2-范数，而 `L1` 代表的是 1-范数）

2.5 其他设置

我们的 `solver.prototxt` 还剩下下面这些设置项：

```
display: 100
snapshot: 2000
snapshot_prefix: "snapshot/alpha"
solver_mode: CPU
```

其中 `display: 100` 代表训练是每隔 100 隔训练周期显示一次损失函数值
`snapshot: 2000` 代表每隔 2000 个训练周期将当前模型的参数 `caffemodel` 和训练过程中的其他数据 `solverstate` 存入快照，快照存入的位置由 `snapshot_prefix` 指定
`solver_mode` 指定训练是在 `CPU` 还是 `GPU` 上进行，这里是 `CPU`。

2.6 开始训练

终于，我们的 `solver.prototxt` 也写好了，可以开始我们的训练了。训练前，你先要确保存放快照文件的文件夹存在，由于这里 `snapshot_prefix` 为 `snapshot/alpha`，所以我们的快照最终后保存在 `snapshot` 目录中，我们运行以下命令创建这个目录：

```
mkdir snapshot
```

通过以下命令执行训练过程：

```
caffe train -solver solver.prototxt
```

`train` 代表了现在我们要进行训练，`-solver solver.prototxt` 指定了我们的 `solver.prototxt` 文件的位置。

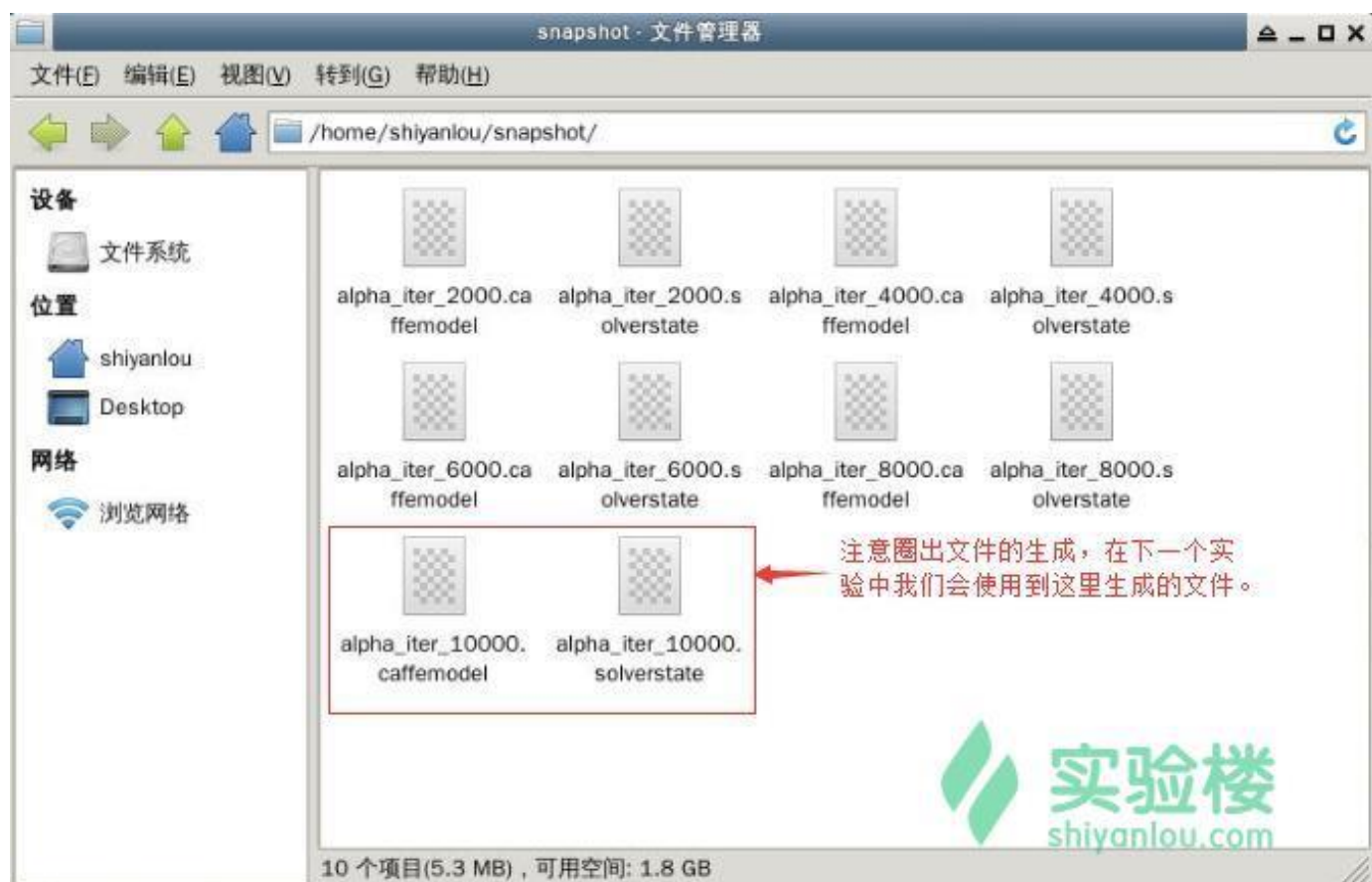
如果没有出错的话，你会先看到很长的一串输出（`caffe` 创建模型时的日志输出），接着就是类似这样的损失函数值输出：


```
Terminal 终端 - caffe train -solver solver.prototxt
文件(E) 编辑(E) 视图(V) 终端(T) 标签(A) 帮助(H)
I0512 15:06:44.425514 416 data_layer.cpp:73] Restarting data prefetching from start.
I0512 15:06:44.968644 414 solver.cpp:397] Test net output #0: accuracy = 0.0389
I0512 15:06:44.968868 414 solver.cpp:397] Test net output #1: loss = 74.7415 (* 1 = 74.7415 loss)
I0512 15:06:45.042495 414 solver.cpp:218] Iteration 0 (-7.31311e-40 iter/s, 12.898s/100 iters), loss = 70.5116
I0512 15:06:45.048666 414 solver.cpp:237] Train net output #0: loss = 70.5116 (* 1 = 70.5116 loss)
I0512 15:06:45.048714 414 sgd_solver.cpp:105] Iteration 0, lr = 0.001
I0512 15:06:50.345038 414 solver.cpp:218] Iteration 100 (18.8822 iter/s, 5.296s/100 iters), loss = 0.299049
I0512 15:06:50.348565 414 solver.cpp:237] Train net output #0: loss = 0.29905 (* 1 = 0.29905 loss)
I0512 15:06:50.348597 414 sgd_solver.cpp:105] Iteration 100, lr = 0.001
I0512 15:06:55.501802 414 solver.cpp:218] Iteration 200 (19.4062 iter/s, 5.153s/100 iters), loss = 0.287529
I0512 15:06:55.508658 414 solver.cpp:237] Train net output #0: loss = 0.287529 (* 1 = 0.287529 loss)
I0512 15:06:55.508790 414 sgd_solver.cpp:105] Iteration 200, lr = 0.001
I0512 15:07:00.793861 414 solver.cpp:218] Iteration 300 (18.9215 iter/s, 5.285s/100 iters), loss = 0.231157
I0512 15:07:00.793956 414 solver.cpp:237] Train net output #0: loss = 0.231157 (* 1 = 0.231157 loss)
I0512 15:07:00.793980 414 sgd_solver.cpp:105] Iteration 300, lr = 0.001
I0512 15:07:06.009563 414 solver.cpp:218] Iteration 400 (19.1755 iter/s, 5.215s/100 iters), loss = 0.465671
I0512 15:07:06.013636 414 solver.cpp:237] Train net output #0: loss = 0.46567 (* 1 = 0.46567 loss)
I0512 15:07:06.013667 414 sgd_solver.cpp:105] Iteration 400, lr = 0.001
I0512 15:07:11.093629 414 solver.cpp:330] Iteration 500, Testing net (#0)
I0512 15:07:23.359252 416 data_layer.cpp:73] Restarting data prefetching from start.
I0512 15:07:23.888907 414 solver.cpp:397] Test net output #0: accuracy = 0.9662
I0512 15:07:23.889101 414 solver.cpp:397] Test net output #1: loss = 0.15743 (* 1 = 0.15743 loss)
I0512 15:07:23.974369 414 solver.cpp:218] Iteration 500 (5.56793 iter/s, 17.96s/100 iters), loss = 0.0434722
I0512 15:07:23.974536 414 solver.cpp:237] Train net output #0: loss = 0.043472 (* 1 = 0.043472 loss)
I0512 15:07:23.974614 414 sgd_solver.cpp:105] Iteration 500, lr = 0.001
```

可以看到，每隔 100 个训练周期，会有损失函数值的输出，损失函数值大致是呈递减的趋势。每隔 500 个训练周期，会有测试准确率输出，准确率大致呈递增的趋势。

整个训练过程 10000 次迭代大致需要三分钟，训练结束后，差不多能够达到 0.994 的准确率。

就这样，我们使用 caffe 训练出了第一个卷积神经网络模型（鼓掌）。在 snapshot 文件夹下面，你可以找到训练完毕的模型参数文件。



三、实验总结

至此，模型的构建和训练过程已经全部完成了，我们拥有了一个准确率超过 0.99 的卷积神经网络模型。下次实验，我们会利用这个模型去开发一个图片字母识别程序，让我们的模型真正的能够发挥作用。

本次实验，我们学习了：

- 让学习速率随着训练的过程逐渐变小可以使最终的参数更接近理想点。
- 正则化是保证模型获得较高的泛化性能的重要手段之一。

四、课后作业

1. `solver.prototxt` 中的超参数我已经提前设置好了，请你自己尝试不同的超参数设置，观察超参数的变化对模型训练过程的影响。

利用训练好的模型开发图片分类程序

一、实验介绍

1.1 实验内容

在 `snapshot` 目录下已经有我们训练好的模型的参数，为了利用我们的卷积神经网络模型和这些参数去对图像进行分类，我们这次实验就来编写代码实现一个图片分类程序。

1.2 实验知识点

- caffe python api

1.3 实验环境

- python 2.7
- opencv 2.4.11
- caffe 1.0.0

二、实验步骤

2.1 准备 `deploy.prototxt`

在第二次实验中，我们编写了 `network.prototxt` 用于进行模型的训练，当模型训练完成需要使用模型进行实际的分类时，就不能再继续用 `network.prototxt` 来定义我们的网络结构了。因为此时我们不需要区分 `TRAIN` 和 `TEST` 这两种 `phase`，同时我们模型的需要输出对每种类别的概率预测，而不是损失函数值，并且我们也不需要准确率层了。

但是我们的网络结构是不会变的，所以我们可以直接在 `network.prototxt` 的基础上进行修改。首先我们需要去掉一个数据层，并且将 `include{phase:TRAIN/TEST}` 参数删掉，然后我们需要删掉 `Accuracy` 层和 `SoftmaxWithLoss` 层，最后需要添加一个概率预测层 `prob`，其定义如下：

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

我们之前说过，通常都会将 `Softmax` 层作为模型对概率的预测输出，所以这里的 `type` 参数为 `Softmax`。

完整的 `deploy.prototxt` 文件可以通过以下命令获取：

```
wget http://labfile.oss.aliyuncs.com/courses/820/deploy.tar.gz  
tar zxvf deploy.tar.gz
```

注意里面还包含了我们接下来会介绍的 `classifier.py` 文件

2.2 转换平均值文件

在前面获取训练数据时，我们使用 `compute_image_mean` 命令计算出了平均值文件 `train.binaryproto`，在我们将要实现的 `python` 程序中，也必须对图片减去均值，为了让 `python` 能够读取平均值文件，我们先将 `train.binaryproto` 转换成 `npz` 文件，以下的脚本实现了这个转换，你可以把这个脚本保存下来以备以后使用。创建 `bin2npz.py` 文件，编写如下代码：

```
#!/usr/bin/env python  
  
import sys  
  
sys.path.append('/opt/caffe/python')  
  
import numpy as np  
import sys, caffe  
  
if len(sys.argv) != 3:
```

```

print "Usage: python bin2numpy.py mean.binaryproto mean.npy"

sys.exit()

blob = caffe.proto.caffe_pb2.BlobProto()
bin_mean = open(sys.argv[1], 'rb').read()
blob.ParseFromString(bin_mean)
arr = np.array(caffe.io.blobproto_to_array(blob))
numpy_mean = arr[0]
print 'channel:', len(numpy_mean), 'row:', len(numpy_mean[0]), 'col:', len(numpy_mean[0][0])
np.save(sys.argv[2], numpy_mean)

```

在 terminal 运行如下命令进行转换:

```
python bin2numpy.py train.binaryproto train.npy
```

```

shiyanolou:~/ $ python bin2numpy.py train.binaryproto train.npy [14:25:40]
channel: 1 row: 16 col: 16
shiyanolou:~/ $ [14:27:15]

```

就将均值文件转换成了 `python` 可以直接读取的 `np` 格式。

2.3 caffe python api

`caffe` 本身是用 `C++` 和 `CUDA` 编写的，但它提供了 `python` 和 `MATLAB` 编程借接口。本次实验我们就是使用 `python` 编程接口来编写图片分类程序。本次实验只会用到很少的几个 `api`，更多功能强大的 `api` 你可以查看 `caffe` 官网的教程。这里创建 `classifier.py` 文件，编写如下代码：

```

# Created by wz on 17-5-12.

# encoding=utf-8

import sys

sys.path.append('/opt/caffe/python') # 先将 pycaffe 路径加入环境变量中

import caffe, cv2, numpy as np

class Classifier: # 将模型封装入一个分类器类中

    def __init__(self, deploy, model, mu):

```

```

self.net = caffe.Net(deploy, model, caffe.TEST) # 初始化网络结构及其中
的参数

self.mu = mu

def classify(self, img):

    img = (img - self.mu) * 0.00390625 # 减去均值后再进行缩放

    self.net.blobs['data'].data[...] = img # 将图片数据送入 data 层的 blobs

    out = self.net.forward()['prob'] # 执行前向计算，并得到最后 prob 层的输出
结果

    return out


def main():

    mean_file = 'train.npy'

    mean = np.load(mean_file) # 加载均值文件

    classifier = Classifier('deploy.prototxt', 'snapshot/alpha_iter_10000.ca
ffemodel', mean) # 创建我们的分类器

    with open('test.txt') as f: # 读取测试集中的图片

        l = f.readlines()

        for i in l:

            print i

            name, label = i.split(' ')

            img = cv2.imread(name) # 读取图片

            img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

            prob = classifier.classify(img) # 使用分类器分类，得到概率

            print prob # 输出概率值

            print chr(np.argmax(prob) + 65) # 输出概率最大值对应的英文字母

            if np.argmax(prob) == int(label):

                cv2.imshow('img', img) # 输出原始图片

                cv2.waitKey() # 等待按键

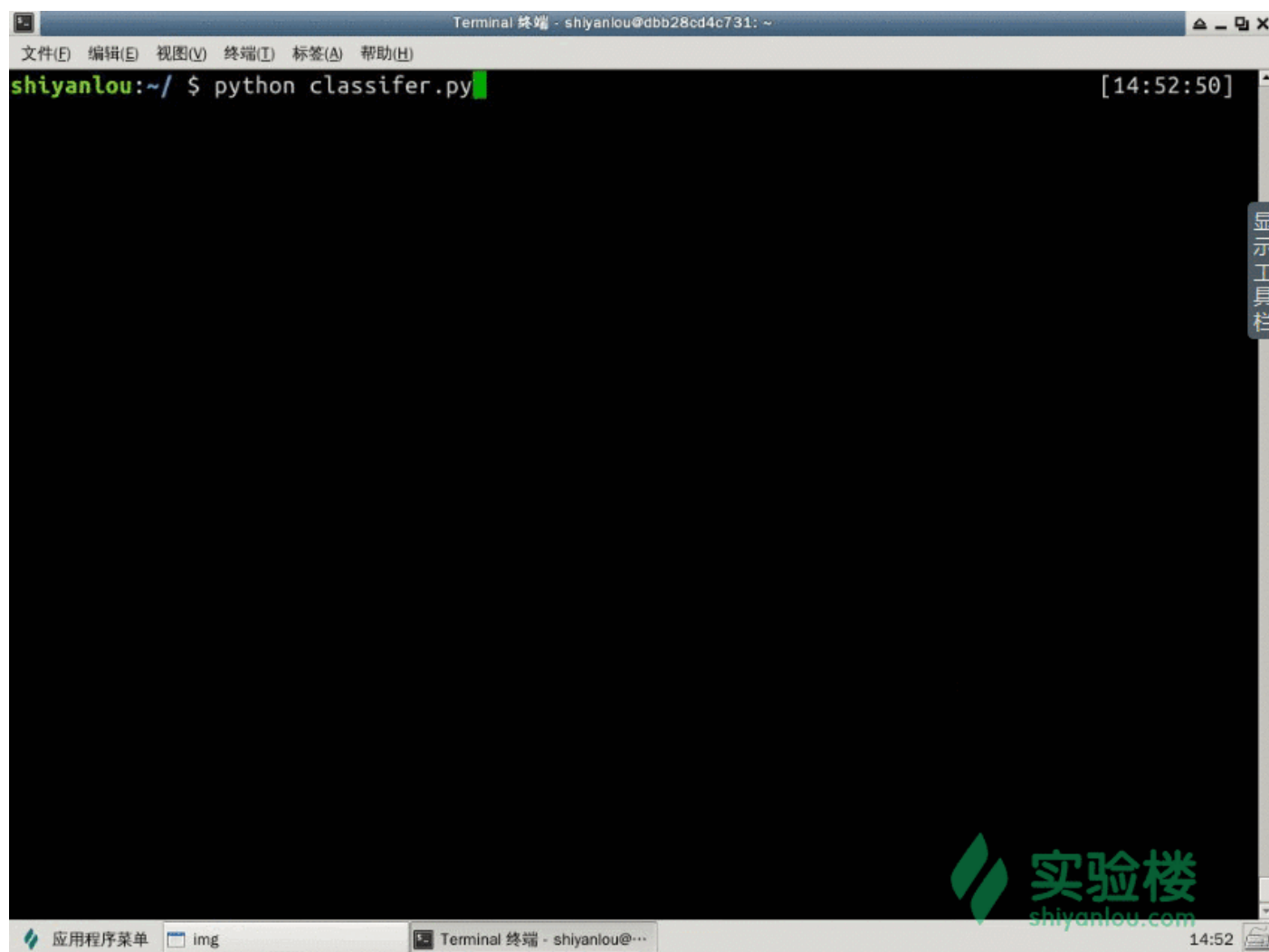

if __name__ == '__main__':

```

```
main()
```

首先我们要将 `pycaffe` 的目录添加进环境变量 `PATH` 中，不然 `import caffe` 会报错。可以看到，我们将 `caffe` 模型封装进一个 `Classifier` 类，注意在 `Classifier.classify()` 函数中，对输入图片的预处理要与训练模型时的预处理完全一样，不然识别准确率可能很低。

在 `main` 函数中，我们对 `test.txt` 里的的每一张图片都调用模型去识别，在 `terminal` 输出预测的概率和预测概率最大的字母，同时显示出原图片，这样你可以直观的感受模型的准确率如何。注意一张图片显示出来后，需要按一下键盘才能显示下一张图片：



三、实验总结

至此，本课程就结束了。虽然我们做的项目十分简单，但相信你也已经体会到了 `caffe` 深度学习框架的强大。而且我们第一次利用训练好的模型开发出了一个直观的图片分类程序，这个程序再一次让我们感受到了深度学习的神奇。

如果你在学习的过程中受到了许多挫折，没有获得理想的学习效果，也不要灰心，这正能体现出深度学习的价值。如果人人都能轻易学会，那还有什么特意花时间去学习的必要呢？如果你对深度学习仍然怀有兴趣，请回过头去再看看没有理解的部分。或者如果你觉得本课程的文档写的不够好，也可以再去查阅相关的资料，随着深度学习的影响力逐渐扩大，一定会有越来越多优质的中文资料出现。

本次实验，我们学习了：

- caffe 提供的 python api 可以使我们方便地通过 python 去调用 caffe 中的一些功能

四、课后作业

1. 请你修改我们的 `classifier.py` 文件，使这个程序不再逐个显示出 `test.txt` 里的文件，而是遍历其中的每一个图片，并计算出模型在测试集上的准确率。
2. [选做] caffe 的 python api 还可以用来构建网络并进行训练，请你自行查阅资料，不使用 `network.prototxt` 和 `solver.prototxt` 文件而直接通过 `python` 程序进行模型的构建和训练。