

Keras

图像深度学习实战

侯宜军 著



版权信息

书名：Keras图像深度学习实战

作者：侯宜军

本书由北京图灵文化发展有限公司发行数字版。版权所有，侵权必究。

您购买的图灵电子书仅供您个人使用，未经授权，不得以任何方式复制和传播本书内容。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

图灵社区会员 llh_y (446181760@qq.com) 专享 尊重版权

内容摘要

作者简介

1 简介

1.1 keras是什么

1.2 一些基本概念

1.2.1 符号计算

1.2.2 张量

1.2.3 data_format

1.2.4 模型

1.2.5 batch

1.2.6 epochs

1.3 安装Python

1.4 安装Theano

1.5 安装opencv

1.6 安装Keras

2 CNN眼中的世界

2.1 卷积神经网络

2.2 使用Keras探索卷积网络的滤波器

2.3 可视化所有的滤波器

2.4 愚弄神经网络

2.5 展望未来

3 Keras模型

3.1 Sequential模型

3.1.1 add

3.1.2 pop

3.1.3 compile

3.1.4 fit

3.1.5 evaluate

3.1.6 predict

3.1.7 predict_classes

3.1.8 train_on_batch

3.1.9 test_on_batch

3.1.10 predict_on_batch

3.1.11 fit_generator

3.1.12 evaluate_generator

3.1.13 predcit_generator

- 3.2 常用层
 - 3.2.1 Dense层
 - 3.2.2 Activation层
 - 3.2.3 Dropout层
 - 3.2.4 Flatten层
 - 3.2.5 Reshape层
 - 3.2.6 Permute层
 - 3.2.7 RepeatVector层
 - 3.2.8 Lambda层
- 3.3 卷积层
 - 3.3.1 Conv1D层
 - 3.3.2 Conv2D层
- 3.4 池化层
 - 3.4.1 MaxPooling1D层
 - 3.4.2 MaxPooling2D层
- 4 目标函数
 - 4.1 MSE
 - 4.2 MAE
 - 4.3 MAPE
 - 4.4 MSLE
 - 4.5 squared_hinge
 - 4.6 hinge
 - 4.7 binary_crossentropy
 - 4.8 categorical_crossentropy
 - 4.9 sparse_categorical_crossentropy
- 5 优化器
 - 5.1 公共参数
 - 5.2 SGD
 - 5.3 RMSprop
 - 5.4 Adagrad
 - 5.5 Adadelta
 - 5.6 Adam
 - 5.7 Adamax
 - 5.8 Nadam
- 6 训练模型的注意事项
 - 6.1 参数初始化

- 6.1.1 零初始化
 - 6.1.2 随机初始化
 - 6.2 数据预处理 (Data Preprocessing)
 - 6.2.1 零均值化 (Mean subtraction)
 - 6.2.2 归一化 (Normalization)
 - 6.2.3 PCA & Whitening
 - 6.2.4 数据扩充 (Data Augmentation)
- 7 图片预处理
 - 7.1 利用小数据量
 - 7.2 图像处理scipy.ndimage
 - 7.3 热点图
 - 7.4 高斯滤波
 - 7.5 图片翻转
 - 7.6 轮廓检测
 - 7.7 角点
 - 7.8 直方图
 - 7.9 形态学图像处理
 - 7.9.1 膨胀和腐蚀
 - 7.9.2 Hit和Miss
- 8 CNN实战
 - 8.1 Mnist
 - 8.1.1 代码
 - 8.1.2 导入导出
 - 8.1.3 预测
 - 8.2 Cifar16
 - 8.2.1 代码
 - 8.2.2 分析
 - 8.3 人脸识别
 - 8.4 视频识别
 - 8.5 用神经网络去噪
 - 8.5.1 对图像添加噪音
 - 8.5.2 去噪神经网络
 - 8.5.3 模型的保存和恢复
 - 8.6 视频抽取图片
- 9 参考

内容摘要

Keras是什么？它是一款非常流行的深度学习计算框架，利用keras只要十几行代码就能写出一个简单的神经网络训练模型。

Keras本身并不提供深度学习的计算引擎，实际它是利用TensorFlow或者Theano作为后端计算引擎的，但它封装了众多API接口，使用者只要了解其封装层的特性就能灵活应用于各种应用场景。是作为深度学习开发者的编程利器。

Keras有两大最显著的特点：一是编程接口简单，封装了众多TensorFlow和Theano细节；二是可在多种机器学习引擎之间自由切换，目前支持TensorFlow和Theano两种，其作者有意未来扩展到其他引擎。

本书不打算涉及Keras的各个方面，而只是聚焦在图像处理领域，并结合图像处理的其他函数综合运用到神经网络模型中，通过此书的学习和实战能达到熟练运用神经网络进行常规图像处理的程度。

本书共分成8个章节，第1章是Keras简介和环境搭建；第2~6章整体介绍Keras的软件框架，包含卷积层，池化层，损失函数，优化器等关键部件的使用说明；第7章介绍了常用的图像预处理技术，包括高斯滤波，轮廓检测等常用操作的介绍；第8章是实战篇，介绍Keras神经网络模型中常用的图像模型的设计方法，以及应用于视频领域的入门级介绍。本书是Keras神经网络中专注于图像识别领域的专业书籍，具有较强的实战性。

本书的目标人群主要定位为具有一定python编程基础，对深度学习原理有一定了解，并且对人工智能图像处理领域有浓厚兴趣的人群。

作者简介

侯宜军，男，南京邮电大学计算机系研究生毕业，先后在电信设计院、摩托罗拉、医疗互联网初创公司等工作过，居住在南京。

具有多年分布式系统开发、数据分析从业经历，对Keras, Storm, Spark, Kafka等大数据技术框架较熟悉。目前研究方向集中在分布式系统、深度学习框架等领域。2015~2016年曾经与他人共同创办六度服务号中医在线平台，2017年初因个人原因退出创业团队，目前在苏宁云商任职高级技术经理。

1 简介

1.1 keras是什么

Keras是基于Theano或TensorFlow的一个深度学习框架，它的设计参考了Torch，用Python语言编写，是一个高度模块化的神经网络库，支持GPU和CPU。

Keras可以通过pip安装，国内可以换一个豆瓣pip源，网速快的惊人！

临时使用时可以使用下述命令：

```
pip install pythonModuleName -i https://pypi.douban.com/simple
```

也可以永久更改：/root/.pip/pip.conf：

```
\[global]
index-url = https://pypi.douban.com/simple
```

在pip.conf中，添加以上内容，就修改了默认的软件源。

到 <https://pypi.python.org/pypi/> 下载所需的python库，直到安装Keras。

安装以下软件包，排名不分先后：

Scipy, Numpy, Theano, Tensowflow, Pyyaml, Six, pycparser, cffi, Keras等。

如果是.whl则运行pip install xxx.whl安装，如果是setup.py则运行python setup.py install安装（如果是源码先运行python setup.py build），如果缺少依赖包会有提示，到pypi.python.org下载对应的依赖包再重新安装就可以了。

安装过程中需要用到vc++ for python编译器，到微软官网上下载，也可从<http://aka.ms/vcpython27>进去。

安装scipy的时候报错no lapack/blas resources found。最后找到一个方法，到<http://www.lfd.uci.edu/~gohlke/pythonlibs/>下载scipy对应的whl安装即可。

最后运行keras的例子：python imdb_lstm.py，如果正常运行说明Keras安装成功了！

如果没有安装TensorFlow但安装了Theano，则import keras会提示找不到TensorFlow，因为Keras默认的backend是TensorFlow，这时候找到用户目录下的.keras/keras.json，将backend从“tensorflow”改成“theano”。

例如目录下：C:\Users\17020405.keras

如果还不行则修改Lib\site-packages\keras\backend下的_init_.py文件，将_BACKEND从“tensorflow”改成“theano”。

1.2 一些基本概念

在开始学习Keras之前，我们希望传递一些关于Keras，关于深度学习的基本概念和技术，这将减少学习过程中的困惑。

1.2.1 符号计算

Keras的底层库使用Theano或TensorFlow，这两个库也称为Keras的后端。无论是Theano还是TensorFlow，都是一个“符号式”的库。

因此，这也使得Keras的编程与传统的Python代码有所差别。笼统的说，符号主义的计算首先定义各种变量，然后建立一个“计算图”，计算图规定了各个变量之间的计算关系。建立好的计算图需要编译以确定其内部细节，然而，此时的计算图还是一个“空壳子”，里面没有任何实际的数据，只有当你把需要运算的输入放进去后，才能在整个模型中形成数据流，从而形成输出值。

就像用管道搭建供水系统，当你在拼水管的时候，里面是没有水的。只有所有的管子都接完了，才能送水。

Keras的模型搭建形式就是这种方法，在你搭建Keras模型完毕后，你的模型就是一个空壳子，只有实际生成可调用的函数后（`K.function`），输入数据，才会形成真正的数据流。

使用计算图的语言，如Theano，以难以调试而闻名，当Keras的Debug进入Theano这个层次时，往往也令人头痛。没有经验的开发者很难直观的感受到底在干些什么。尽管很让人头痛，但大多数的深度学习框架使用的都是符号计算这一套方法，因为符号计算能够提供关键的计算优化、自动求导等功能。

通常建议在使用Keras前稍微了解一下Theano或TensorFlow，百度一下即可。

1.2.2 张量

张量，或tensor，是本文档会经常出现的一个词汇，在此稍作解释。

使用这个词汇的目的是为了表述统一，张量可以看作是向量、矩阵的自然推广，我们用张量来表示广泛的数据类型。

规模最小的张量是0阶张量，即标量，也就是一个数。

当我们把一些数有序的排列起来，就形成了1阶张量，也就是一个向量。

如果我们继续把一组向量有序的排列起来，就形成了2阶张量，也就是一个矩阵。

把矩阵摞起来，就是3阶张量，我们可以称为一个立方体，具有3个颜色通道的彩色图片就是一个这样的立方体。

把立方体摞起来，好吧这次我们真的没有给它起别名了，就叫4阶张量了，不要去试图想像4阶张量是什么样子，它就是个数学上的概念。

张量的阶数有时候也称为维度，或者轴，轴这个词翻译自英文axis。譬如一个矩阵[[1,2], [3,4]]，是一个2阶张量，有两个维度或轴，沿着第0个轴（为了与python的计数方式一致，本文档维度和轴从0算起）你看到的是[1,2]，[3,4]两个向量，沿着第1个轴你看到的是[1,3]，[2,4]两个向量。

要理解“沿着某个轴”是什么意思，不妨试着运行一下下面的代码：

```
import numpy as np
a = np.array([[1,2],[3,4]])
sum0 = np.sum(a, axis=0)
sum1 = np.sum(a, axis=1)
print sum0
print sum1
```

关于张量，目前知道这么多就足够了。

1.2.3 data_format

这是一个无可奈何的问题，在如何表示一组彩色图片的问题上，Theano和TensorFlow发生了分歧，‘th’模式，也即Theano模式会把100张RGB三通道的16×32（高为16宽为32）彩色图表示为下面这种形式（100,3,16,32），Caffe采取的也是这种方式。第0个维度是样本维，代表样本的数目，第1个维度是通道维，代表颜色通道数。后面两个就是高和宽了。这种theano风格的数据组织方法，称为“channels_first”，即通道维靠前。

而TensorFlow的表达形式是（100,16,32,3），即把通道维放在了最后，这种数据组织方式称为“channels_last”。

Keras默认的数据组织形式在~/.keras/keras.json中规定，可查看该文件的image_data_format一项，也可在代码中通过K.image_data_format()函数返回，请在网络的训练和测试中保持维度顺序一致。

1.2.4 模型

在Keras0.x中有两种模型，一种是Sequential模型，又称为序贯模型，也就是单输入单输出，一条路通到底，层与层之间只有相邻关系，没有跨层连接。这种模型编译速度快，操作上也比较简单。

第二种模型称为Graph，即图模型，这个模型支持多输入多输出，层与层之间想怎么连怎么连，但是编译速度慢。可以看到，Sequential其实是Graph的一个特殊情况。

在Keras1和Keras2中，图模型被移除，从而增加了“functional model API”这个东西，更加强调了Sequential模型是特殊的一种。一般的模型就称为Model。

由于functional model API在使用时利用的是“函数式编程”的风格，我们这里将其称为函数

式模型。总而言之，只要这个东西接收一个或一些张量作为输入，然后输出的也是一个或一些张量，但不管它是什么，统统都叫做“模型”。

1.2.5 batch

batch这个概念与Keras无关，老实讲不应该出现在这里的，但是因为它频繁出现，而且不了解这个术语的话看函数会很头痛，所有这里还是简单说一下。

深度学习的优化算法，说白了就是梯度下降。每次的参数更新有两种方式。

第一种，遍历全部数据集算一次损失函数，然后算函数对各个参数的梯度，更新梯度。这种方法每更新一次参数都要把数据集里的所有样本都看一遍，计算量开销大，计算速度慢，不支持在线学习，这称为Batch gradient descent，批梯度下降。

另一种，每看一个数据就算一下损失函数，然后求梯度更新参数，这个称为随机梯度下降，stochastic gradient descent。这个方法速度比较快，但是收敛性能不太好，可能在最优点附近晃来晃去，hit不到最优点。两次参数的更新也有可能互相抵消掉，造成目标函数震荡的比较剧烈。

为了克服两种方法的缺点，现在一般采用的是一种折中手段，mini-batch gradient decent，小批的梯度下降，这种方法把数据分为若干个批，按批来更新参数，这样，一个批中的一组数据共同决定了本次梯度的方向，下降起来就不容易跑偏，减少了随机性。另一方面因为批的样本数与整个数据集相比小了很多，所以计算量也不是很大。

基本上现在的梯度下降都是基于mini-batch的，所以Keras的模块中经常会出现batch_size，就是指这个。

顺便说一句，Keras中用的优化器SGD是stochastic gradient descent的缩写，但不代表是一个样本就更新一回，而是基于mini-batch的。

1.2.6 epochs

简单说，epochs指的就是训练过程中数据将被“轮询”多少次，就这样。

1.3 安装Python

安装Python2.7版本，具体安装方法这里就不详细说明了。

1.4 安装Theano

从<https://github.com/Theano/Theano>下载Theano的主版本，将theano目录放到Python的Lib\site-packagesKeras目录中。

从github下载Keras源码并安装到Python环境中。

1.5 安装opencv

图像识别需要我们首先安装python环境下的图像处理库。Python社区提供了许多的图像处理库，这里我们使用opencv作为图像处理库。笔者是下载opencv的whl安装文件进行安装的，命令为：

```
pip install opencv_python-3.2.0.7-cp27-cp27m-win_amd64.whl
```

如果安装过程中由于网络的原因等造成依赖包无法下载安装，则可以根据提示从：

<https://pypi.python.org/pypi>

下载所需的依赖包版本，先安装依赖再安装opencv，直到最后安装成功。

1.6 安装Keras

从<https://github.com/fchollet/keras>下载Keras源码，将keras目录放到Python的Lib\site-packages目录中。

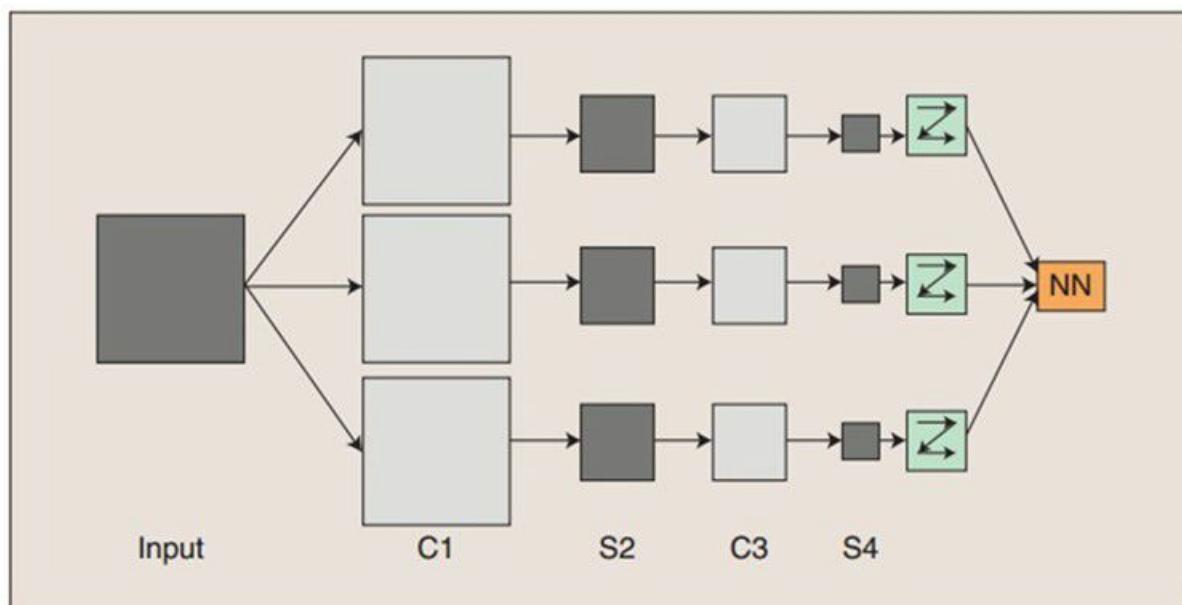
注意这里Keras和Theano的版本要保持一致，否则容易出现不兼容的情况而导致示例代码无法运行，笔者安装的是github上的master版本，可以正常运行示例。

2 CNN眼中的世界

我们首先以Keras官网中一篇经典的关于CNN原理的科普文章作为开始，从这篇文章中我们希望能了解到CNN模型的前世今生，以及对CNN原理有一个初步的感性认识。

2.1 卷积神经网络

卷积神经网络是一个多层的神经网络，每层由多个二维平面组成，而每个平面由多个独立神经元组成。



上图是卷积神经网络的概念示范：输入图像通过和三个可训练的滤波器及可加偏置进行卷积，卷积后在C1层产生三个特征映射图，然后特征映射图中每组的四个像素再进行求和，加权值，加偏置，通过一个Sigmoid函数得到三个S2层的特征映射图。这些映射图再经过滤波得到C3层。这个层级结构再和S2一样产生S4。最终，这些像素值被光栅化，并连接成一个向量输入到传统的神经网络，从而得到输出。

每个层有多个Feature Map，每个Feature Map通过一种卷积滤波器提取输入的一种特征，然后每个Feature Map有多个神经元。

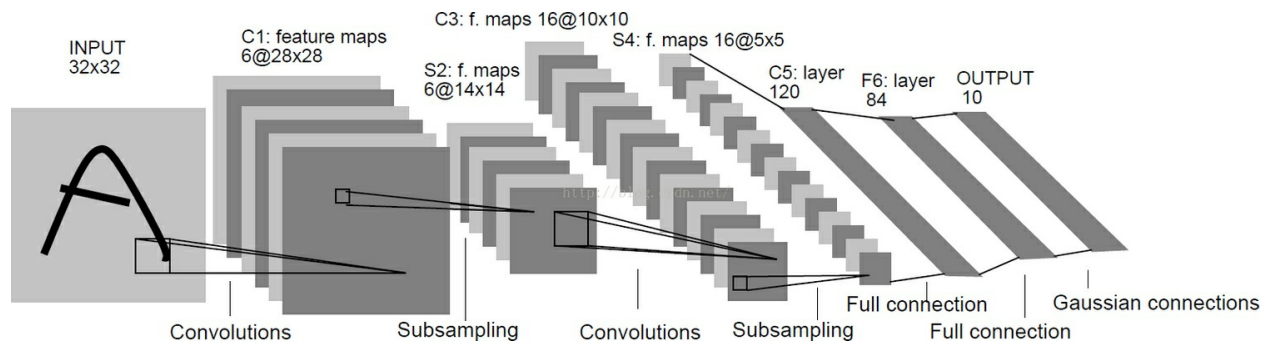
卷积过程如下，比如矩阵 $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ ，假设卷积核为 $\begin{bmatrix} 2 & 2 \end{bmatrix}$ ，初始偏移为0，假设权重为0.2，则卷积之后的结果为 $\begin{bmatrix} 0.6 & 0.8 \end{bmatrix}$ 。计算过程如下：

$$(1 \times 0.2 + 2 \times 0.2 + 4 \times 0.2 + 5 \times 0.2) / 4 = 2.4 / 4 = 0.6$$

$$(2 \times 0.2 + 3 \times 0.2 + 5 \times 0.2 + 6 \times 0.2) / 4 = 3.2 / 4 = 0.8$$

CNN一个极具创新的地方就在于通过权值共享减少了神经网络需要训练的参数数目，所谓权值共享就是同一个Feature Map中神经元权值共享，该Feature Map中的所有神经元使用同一个权值。因此参数个数与神经元的个数无关，只与卷积核的大小及Feature Map的个数相关。但是共有多少个连接个数就与神经元的个数相关了，神经元的个数也就是特征图的大小。

下面以最经典的LeNet-5例子来逐层分析各层的参数及连接个数。



C1层是一个卷积层，由6个特征图Feature Map构成。特征图中每个神经元与输入为 5×5 的邻域相连。特征图的大小为 28×28 ，这样能防止输入的连接掉到边界之外（ $32 - 5 + 1 = 28$ ）。C1有156个可训练参数（每个滤波器 $5 \times 5 = 25$ 个unit参数和一个bias参数，一共6个滤波器，共 $(5 \times 5 + 1) \times 6 = 156$ 个参数），共 $156 \times (28 \times 28) = 122,304$ 个连接。

S2层是一个下采样层，有6个 14×14 的特征图。特征图中的每个单元与C1中相对应特征图的 2×2 邻域相连接。S2层每个单元的4个输入相加，乘以一个可训练参数，再加上一个可训练偏置。每个单元的 2×2 感受野并不重叠，因此S2中每个特征图的大小是C1中特征图大小的 $1/4$ （行和列各 $1/2$ ）。S2层有12（ $6 \times (1 + 1) = 12$ ）个可训练参数和5880（ $14 \times 14 \times (2 \times 2 + 1) \times 6 = 5880$ ）个连接。

C3层也是一个卷积层，它同样通过 5×5 的卷积核去卷积层S2，然后得到的特征map就只有 10×10 个神经元，但是它有16种不同的卷积核，所以就存在16个特征map了。C3中每个特征图由S2中所有6个或者几个特征map组合而成。为什么不把S2中的每个特征图连接到每个C3的特征图呢？原因有2点。第一，不完全的连接机制将连接的数量保持在合理的范围内。第二，也是最重要的，其破坏了网络的对称性。由于不同的特征图有不同的输入，所以迫使他们抽取不同的特征（希望是互补的）。

例如，存在的一个方式是：C3的前6个特征图以S2中3个相邻的特征图子集为输入。接下来6个特征图以S2中4个相邻特征图子集为输入。然后的3个特征图以不相邻的4个特征图子集为输入。最后一个将S2中所有特征图作为输入。这样C3层有1516（ $6 \times (3 \times 25 + 1) + 6 \times (4 \times 25 + 1) + 3 \times (4 \times 25 + 1) + (25 \times 6 + 1) = 1516$ ）个可训练参数和151600（ $10 \times 10 \times 1516 = 151600$ ）个连接。

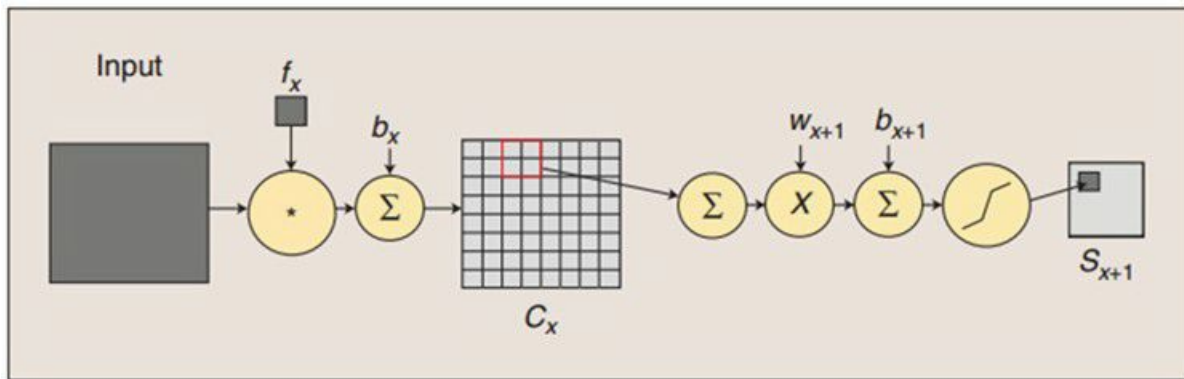
S4层是一个下采样层，由16个 5×5 大小的特征图构成。特征图中的每个单元与C3中相应特征图的 2×2 邻域相连接，跟C1和S2之间的连接一样。S4层有32个可训练参数（每个特征图1个因子和一个偏置 $16 \times (1 + 1) = 32$ ）和2000（ $16 \times (2 \times 2 + 1) \times 5 \times 5 = 2000$ ）个连接。

C5层是一个卷积层，有120个特征图。每个单元与S4层的全部16个单元的 5×5 邻域相连。由于S4层特征图的大小也为 5×5 （同滤波器一样），故C5特征图的大小为 1×1 （ $5 - 5 + 1 = 1$ ）：这构成了S4和C5之间的全连接。之所以仍将C5标示为卷积层而非全相联层，是因为如果LeNet-5的输入变大，而其他的保持不变，那么此时特征图的维数就会比 1×1 大。C5层有48120（ $120 \times (16 \times 5 \times 5 + 1) = 48120$ 由于与全部16个单元相连，故只加一个偏置）个可训练连接。

F6层有84个单元（之所以选这个数字的原因来自于输出层的设计），与C5层全相连。有10164（ $84 \times (120 \times (1 \times 1) + 1) = 10164$ ）个可训练参数。如同经典神经网络，F6层计算输入向

量和权重向量之间的点积，再加上一个偏置。然后将其传递给sigmoid函数产生单元i的一个状态。

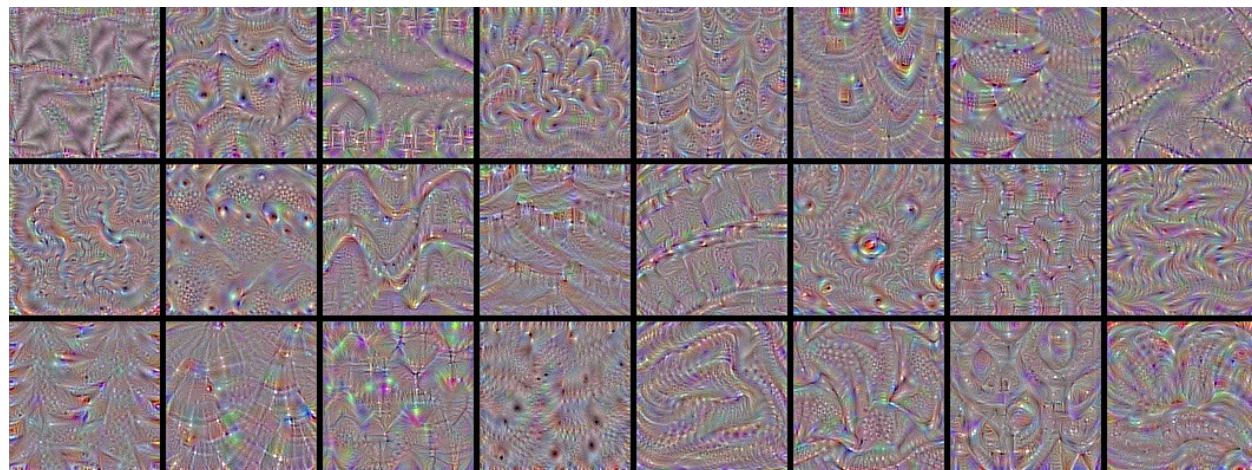
最后，输出层由欧式径向基函数（Euclidean Radial Basis Function）单元组成，每类一个单元，每个有84个输入。



卷积过程包括：用一个可训练的滤波器 f_x 去卷积一个输入的图像（第一阶段是输入的图像，后面的阶段就是卷积特征map了），然后加一个偏置 b_x ，得到卷积层 C_x 。子采样过程包括：每邻域四个像素求和变为一个像素，然后通过标量 w_{x+1} 加权，再增加偏置 b_{x+1} ，然后通过一个sigmoid激活函数，产生一个大概缩小四倍的特征映射图 S_{x+1} 。

2.2 使用Keras探索卷积网络的滤波器

本文中我们将利用Keras观察CNN到底在学些什么，它是如何理解我们送入的训练图片的。我们将使用Keras来对滤波器的激活值进行可视化。本文使用的神经网络是VGG-16，数据集为ImageNet。本文的代码可以在github找到。



VGG-16又称为OxfordNet，是由牛津视觉几何组（Visual Geometry Group）开发的卷积神经网络结构。该网络赢得了ILSVR（ImageNet）2014的冠军。时至今日，VGG仍然被认为是一个杰出的视觉模型——尽管它的性能实际上已经被后来的Inception和ResNet超过了。

Lorenzo Baraldi将Caffe预训练好的VGG16和VGG19模型转化为了Keras权重文件，所以我们可以简单的通过载入权重来进行实验。

该权重文件的下载需要自备梯子（这里有一个网盘保持的
vgg16: http://files.heuritech.com/weights/vgg16_weights.h5赶紧下载）。

首先，我们在Keras中定义VGG网络的结构：

```
from keras.models import Sequential
from keras.layers import Convolution2D, ZeroPadding2D, MaxPooling2D
img_width, img_height = 128, 128
# build the VGG16 network
model = Sequential()
model.add(ZeroPadding2D((1, 1), batch_input_shape=(1, 3, img_width, img_height)))
first_layer = model.layers[-1]
# this is a placeholder tensor that will contain our generated images
input_img = first_layer.input
# build the rest of the network
model.add(Convolution2D(64, 3, 3, activation='relu', name='conv1_1'))
model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(64, 3, 3, activation='relu', name='conv1_2'))
model.add(MaxPooling2D((2, 2), strides=(2, 2)))
model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(128, 3, 3, activation='relu', name='conv2_1'))
model.add(ZeroPadding2D((1, 1)))
```

```

model.add(Convolution2D(128, 3, 3, activation='relu', name='conv2_2'))
model.add(MaxPooling2D((2, 2), strides=(2, 2)))

model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(256, 3, 3, activation='relu', name='conv3_1'))
model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(256, 3, 3, activation='relu', name='conv3_2'))
model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(256, 3, 3, activation='relu', name='conv3_3'))
model.add(MaxPooling2D((2, 2), strides=(2, 2)))
model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(512, 3, 3, activation='relu', name='conv4_1'))
model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(512, 3, 3, activation='relu', name='conv4_2'))
model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(512, 3, 3, activation='relu', name='conv4_3'))
model.add(MaxPooling2D((2, 2), strides=(2, 2)))
model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(512, 3, 3, activation='relu', name='conv5_1'))
model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(512, 3, 3, activation='relu', name='conv5_2'))
model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(512, 3, 3, activation='relu', name='conv5_3'))
model.add(MaxPooling2D((2, 2), strides=(2, 2)))

# get the symbolic outputs of each "key" layer (we gave them unique names).
layer_dict = dict([(layer.name, layer) for layer in model.layers])

```

注意我们不需要全连接层，所以网络就定义到最后一个卷积层为止。使用全连接层会将输入大小限制为 224×224 ，即ImageNet原图片的大小。这是因为如果输入的图片大小不是 224×224 ，在从卷积过度到全链接时向量的长度与模型指定的长度不相符。

下面，我们将预训练好的权重载入模型，一般而言我们可以通过`model.load_weights()`载入，但这里我们只载入一部分参数，如果使用该方法的话，模型和参数形式就不匹配了。所以我们需要手工载入：

```

import h5py
weights_path = 'vgg16_weights.h5'
f = h5py.File(weights_path)
import h5py
weights_path = 'vgg16_weights.h5'
f = h5py.File(weights_path)
for k in range(f.attrs['nb_layers']):
    if k >= len(model.layers):
        # we don't look at the last (fully-connected) layers in the savefile
        break
    g = f['layer_{}'.format(k)]
    weights = [g['param_{}'.format(p)] for p in range(g.attrs['nb_params'])]
    model.layers[k].set_weights(weights)
f.close()
print('Model loaded.')

```

下面，我们要定义一个损失函数，这个损失函数将用于最大化某个指定滤波器的激活值。以该函数为优化目标优化后，我们可以真正看一下使得这个滤波器激活的究竟是些什么东

西。

现在我们使用Keras的后端来完成这个损失函数，这样代码不用修改就可以在TensorFlow和Theano之间切换了。TensorFlow在CPU上进行卷积要快的多，而目前为止Theano在GPU上进行卷积要快一些。

```
from keras import backend as K
layer_name = 'conv5_1'
filter_index = 0
# build a loss function that maximizes the activation
# of the nth filter of the layer considered
layer_output = layer_dict[layer_name].output
loss = K.mean(layer_output[:, filter_index, :, :])
# compute the gradient of the input picture wrt this loss
grads = K.gradients(loss, input_img)[0]
# normalization trick: we normalize the gradient
grads /= (K.sqrt(K.mean(K.square(grads))) + 1e-5)
# this function returns the loss and grads given the input picture
iterate = K.function([input_img], [loss, grads])
```

注意这里有个小trick，计算出来的梯度进行了正规化，使得梯度不会过小或过大。这种正规化能够使梯度上升的过程平滑进行。

根据刚刚定义的函数，现在可以对某个滤波器的激活值进行梯度上升。

```
import numpy as np
# we start from a gray image with some noise
input_img_data = np.random.random((1, 3, img_width, img_height)) * 20 + 128.
# run gradient ascent for 20 steps
for i in range(20):
    loss_value, grads_value = iterate([input_img_data])
    input_img_data += grads_value * step
```

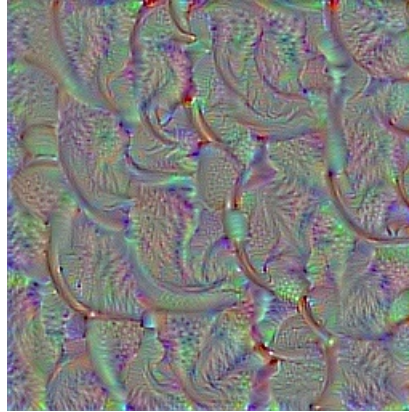
使用TensorFlow时，这个操作大概只要几秒。

然后我们可以提取出结果，并可视化：

```
from scipy.misc import imsave
# util function to convert a tensor into a valid image
def deprocess_image(x):
    # normalize tensor: center on 0., ensure std is 0.1
    x -= x.mean()
    x /= (x.std() + 1e-5)
    x *= 0.1
    # clip to [0, 1]
    x += 0.5
    x = np.clip(x, 0, 1)
    # convert to RGB array
    x *= 255
    x = x.transpose((1, 2, 0))
    x = np.clip(x, 0, 255).astype('uint8')
    return x
img = input_img_data[0]
```

```
img = deprocess_image(img)
imsave('%s_filter_%d.png' % (layer_name, filter_index), img)
```

这里是第5卷基层第0个滤波器的结果：

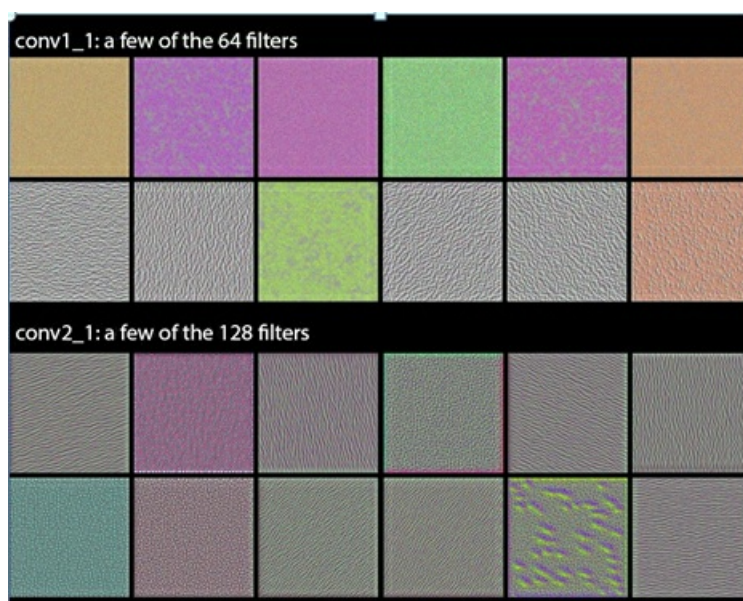


2.3 可视化所有的滤波器

下面我们系统的可视化一下各个层的各个滤波器结果，看看CNN是如何对输入进行逐层分解的。

第一层的滤波器主要完成方向、颜色的编码，这些颜色和方向与基本的纹理组合，逐渐生成复杂的形状。

可以将每层的滤波器想像为基向量，这些基向量一般是过完备的。基向量可以将层的输入紧凑地编码出来。滤波器随着其利用的空域信息的拓宽而更加精细和复杂。



上图只是展示了部分内容。可以观察到，很多滤波器的内容其实是一样的，只不过旋转了一个随机的角度（如90度）而已。这意味着我们可以通过使得卷积滤波器具有旋转不变性而显著减少滤波器的数目，这是一个有趣的研究方向。

令人震惊的是，这种旋转的性质在高层的滤波器中仍然可以被观察到，如Conv4_1。

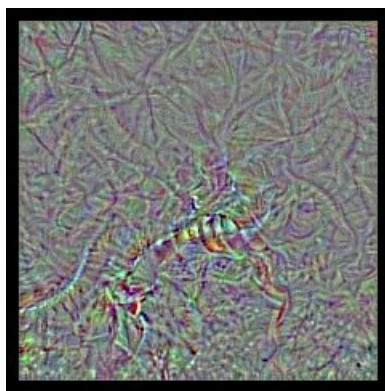
2.4 愚弄神经网络

如果我们添加上VGG的全连接层，然后试图最大化某个指定类别的激活值呢？你会得到一张很像该类别的图片吗？让我们试试。

这种情况下我们的损失函数长这样：

```
layer_output = model.layers\[-1\].get_output()
loss = K.mean(layer_output\[:, output_index])
```

比方说我们来最大化输出下标为65的那个类，在ImageNet里，这个类是蛇。很快，我们的损失达到了0.999，即神经网络有99.9%的概率认为我们生成的图片是一条海蛇，它长这样：



不太像呀，换个类别试试，这次选喜鹊类（第18类）。



OK，我们的网络认为是喜鹊的东西看起来完全不是喜鹊，往好了说，这个图里跟喜鹊相似的，也不过就是一些局部的纹理，如羽毛、嘴巴之类的。那么，这就意味着卷积神经网络是个很差的工具吗？当然不是，我们按照一个特定任务来训练它，它就会在那个任务上表现的不错。但我们不能有网络“理解”某个概念的错觉。我们不能将网络人格化，它只是

工具而已。比如一条狗，它能识别其为狗只是因为它能以很高的概率将其正确分类而已，而不代表它理解关于“狗”的任何外延。

2.5 展望未来

所以，神经网络到底理解了什么呢？我认为有两件事是它们理解的。

其一，神经网络理解了如何将输入空间解耦为分层次的卷积滤波器组。其二，神经网络理解了从一系列滤波器的组合到一系列特定标签的概率映射。神经网络学习到的东西完全达不到人类的“看见”的意义，从科学的角度讲，这当然也不意味着我们已经解决了计算机视觉的问题。想得别太多，我们才刚刚踏上计算机视觉天梯的第一步。

有些人说，卷积神经网络学习到的对输入空间的分层次解耦模拟了人类视觉皮层的行为。这种说法可能对也可能不对，但目前未知我们还没有比较强的证据来承认或否认它。当然，有些人可以期望人类的视觉皮层就是以类似的方式学东西的，某种程度上讲，这是对我们视觉世界的自然解耦（就像傅里叶变换是对周期声音信号的一种解耦一样自然）。但是，人类对视觉信号的滤波、分层次、处理的本质很可能和我们弱鸡的卷积网络完全不是一回事。视觉皮层不是卷积的，尽管它们也分层，但那些层具有皮质列的结构，而这些结构的真正目的目前还不得而知，这种结构在我们的人工神经网络中还没有出现（尽管乔大帝Geoff Hinton正在在这个方面努力）。此外，人类有比给静态图像分类的感知器多得多的视觉感知器，这些感知器是连续而主动的，不是静态而被动的，这些感受器还被如眼动等多种机制复杂控制。

总而言之，卷积神经网络的可视化工作是很让人着迷的，谁能想到仅仅通过简单的梯度下降法和合理的损失函数，加上大规模的数据库，就能学到能很好解释复杂视觉信息的如此漂亮的分层模型呢。深度学习或许在实际的意义上并不智能，但它仍然能够达到几年前任何人都无法达到的效果。现在，如果我们能理解为什么深度学习如此有效，那……嘿嘿。

3 Keras模型

3.1 Sequential模型

Keras一般用Sequential模型做为搭建神经网络的开始，本节简要论述Sequential模型接口的主要使用方法。

3.1.1 add

定义：add(self, layer)

用途：向模型中添加一个层。

参数layer是Layer对象，也即是层。

3.1.2 pop

定义：pop(self)

用途：弹出模型最后的一层，无返回值，该方法一般很少用到。

3.1.3 compile

定义：

```
compile(self, optimizer, loss, metrics=None, sample_weight_mode=None)
```

该方法编译用来配置模型的学习过程，其参数有以下这些：

- **optimizer**：字符串（预定义优化器名）或优化器对象，参考优化器。
- **loss**：字符串（预定义损失函数名）或目标函数，参考损失函数。
- **metrics**：列表，包含评估模型在训练和测试时的网络性能的指标，典型用法是 `metrics=['accuracy']`。
- **sample_weight_mode**：如果你需要按时间步为样本赋权（2D权矩阵），将该值设为“temporal”。默认为“None”，代表按样本赋权（1D权）。
- **kwargs**：使用TensorFlow作为后端请忽略该参数，若使用Theano作为后端，kwargs的值将会传递给 `K.function`。

例子：

```
model = Sequential()
model.add(Dense(32, input_shape=(500,)))
model.add(Dense(10, activation='softmax'))
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

该例子添加两个Dense层，并用rmsprop优化器进行优化，损失函数定义为categorical_crossentropy，各种损失函数的含义在后面章节中会提高。

模型在使用前必须编译，否则在调用fit或evaluate时会抛出异常。

3.1.4 fit

定义：

```
fit(self, x, y, batch_size=32, epochs=10, verbose=1, callbacks=None, validation_split=0.0, val
```

本函数将模型训练epoch轮，其参数有：

- **x**: 输入数据。如果模型只有一个输入，那么x的类型是numpy array，如果模型有多个输入，那么x的类型应当为list，list的元素是对应于各个输入的numpy array。
- **y**: 标签向量，numpy array类型。
- **batch_size**: 整数，指定进行梯度下降时每个batch包含的样本数。训练时一个batch的样本会被计算一次梯度下降，使目标函数优化一步。
- **epochs**: 训练的轮数，每个epoch会把训练集轮一遍。
- **verbose**: 日志显示，0为不在标准输出流输出日志信息，1为输出进度条记录，2为每个epoch输出一行记录。
- **callbacks**: list，其中的元素是keras.callbacks.Callback的对象。这个list中的回调函数将会在训练过程中的适当时机被调用，参考回调函数
- **validation_split**: 0~1之间的浮点数，用来指定训练集的一定比例数据作为验证集。注意，validation_split的划分在shuffle之前，因此如果你的数据本身是有序的，需要先手工打乱再指定validation_split，否则可能会出现验证集样本不均匀。
- **validation_data**: 形式为（X, y）的tuple，是指定的验证集。此参数将覆盖validation_split参数。
- **shuffle**: 布尔值或字符串，一般为布尔值，表示是否在训练过程中随机打乱输入样本的顺序。若为字符串“batch”，则是用来处理HDF5数据的特殊情况，它将在batch内部将数据打乱。

- **class_weight**: 字典，将不同的类别映射为不同的权值，该参数用来在训练过程中调整损失函数（只能用于训练）。
- **sample_weight**: 权值的numpy array，用于在训练时调整损失函数（仅用于训练）。可以传递一个1D的与样本等长的向量用于对样本进行1对1的加权，或者在面对时序数据时，传递一个的形式为<samples, sequence_length>的矩阵来为每个时间步上的样本赋不同的权。这种情况下请确定在编译模型时添加了sample_weight_mode='temporal'。
- **initial_epoch**: 从该参数指定的epoch开始训练，在继续之前的训练时有用。

fit函数返回一个History的对象，其History.history属性记录了损失函数和其他指标的数值随epoch变化的情况，如果有验证集的话，也包含了验证集的这些指标变化情况。

3.1.5 evaluate

定义：

```
evaluate(self, x, y, batch_size=32, verbose=1, sample_weight=None)
```

本函数按batch计算在某些输入数据上模型的误差，其参数有：

- **x**: 输入数据，与fit一样，是numpy array或numpy array的list。
- **y**: 标签，numpy array。
- **batch_size**: 整数，含义同fit的同名参数。
- **verbose**: 含义同fit的同名参数，但只能取0或1。
- **sample_weight**: numpy array，含义同fit的同名参数。

本函数返回一个测试误差的标量值（如果模型没有其他评价指标），或一个标量的list（如果模型还有其他的评价指标）。model.metrics_names将给出list中各个值的含义。

如果没有特殊说明，以下函数的参数均保持与fit的同名参数相同的含义。

3.1.6 predict

```
predict(self, x, batch_size=32, verbose=0)
```

本函数按batch获得输入数据对应的预测结果，其中x是输入向量，batch_size是每批次选取的数据集数量。

函数的返回值是预测值的numpy array。

3.1.7 predict_classes

```
predict_classes(self, x, batch_size=32, verbose=1)
```

本函数按batch产生输入数据的类别预测结果，它和predict的区别是：

Predict返回各个类别的可能性结果，是一个n维向量，n等于类别的数量；而predict_classes返回的是最可能的类别，对每个输入返回的是类别名称。

函数的返回值是类别预测结果的numpy array或numpy。

3.1.8 train_on_batch

```
train_on_batch(self, x, y, class_weight=None, sample_weight=None)
```

本函数在一个batch的数据上进行一次参数更新。

函数返回训练误差的标量值或标量值的list，与evaluate的情形相同。

3.1.9 test_on_batch

```
test_on_batch(self, x, y, sample_weight=None)
```

本函数在一个batch的样本上对模型进行评估。

函数的返回与evaluate的情形相同。

3.1.10 predict_on_batch

```
predict_on_batch(self, x)
```

本函数在一个batch的样本上对模型进行测试。

函数返回模型在一个batch上的预测结果。

3.1.11 fit_generator

```
fit_generator(self, generator, steps_per_epoch, epochs=1, verbose=1, callbacks=None, validation
```

利用Python的生成器，逐个生成数据的batch并进行训练。生成器与模型将并行执行以提高效率。例如，该函数允许我们在CPU上进行实时的数据提升，同时在GPU上进行模型训

练。

函数的参数是：

- **generator**: 生成器函数，生成器的输出应该为：
 - a) 一个形如 (inputs, targets) 的tuple。
 - b) 一个形如 (inputs, targets, sample_weight) 的tuple。所有的返回值都应该包含相同数目的样本。生成器将无限在数据集上循环。每个epoch以经过模型的样本数达到 samples_per_epoch 时，记一个epoch结束。
- **steps_per_epoch**: 整数，当生成器返回 steps_per_epoch 次数据时计一个epoch结束，执行下一个epoch。
- **epochs**: 整数，数据迭代的轮数。
- **verbose**: 日志显示，0为不在标准输出流输出日志信息，1为输出进度条记录，2为每个epoch输出一行记录。
- **validation_data**: 具有以下三种形式之一。
 - a) 生成验证集的生成器。
 - b) 一个形如 (inputs, targets) 的tuple。
 - c) 一个形如 (inputs, targets, sample_weights) 的tuple。
- **validation_steps**: 当 validation_data 为生成器时，本参数指定验证集的生成器返回次数。
- **class_weight**: 规定类别权重的字典，将类别映射为权重，常用于处理样本不均衡问题。
- **sample_weight**: 权值的numpy array，用于在训练时调整损失函数（仅用于训练）。可以传递一个1D的与样本等长的向量用于对样本进行1对1的加权，或者在面对时序数据时，传递一个的形式为 (samples, sequence_length) 的矩阵来为每个时间步上的样本赋不同的权。这种情况下请确定在编译模型时添加了 sample_weight_mode='temporal'。
- **workers**: 最大进程数。
- **max_q_size**: 生成器队列的最大容量。
- **pickle_safe**: 若为真，则使用基于进程的线程。由于该实现依赖多进程，不能传递non pickleable（无法被pickle序列化）的参数到生成器中，因为无法轻易将它们传入子进程中。
- **initial_epoch**: 从该参数指定的epoch开始训练，在继续之前的训练时有用。

函数返回一个History对象。

例子：

```
def generate_arrays_from_file(path):
    while 1:
        f = open(path)
        for line in f:
            # create Numpy arrays of input data
            # and labels, from each line in the file
            x, y = process_line(line)
            yield (x, y)
        f.close()
model.fit_generator(generate_arrays_from_file('/my_file.txt'),
                    samples_per_epoch=10000, epochs=10)
```

3.1.12 evaluate_generator

```
evaluate_generator(self, generator, steps, max_q_size=10, workers=1, pickle_safe=False)
```

本函数使用一个生成器作为数据源评估模型，生成器应返回与test_on_batch的输入数据相同类型的数据。该函数的参数与fit_generator同名参数含义相同，steps是生成器要返回数据的轮数。

3.1.13 predict_generator

```
predict_generator(self, generator, steps, max_q_size=10, workers=1, pickle_safe=False, verbose=0)
```

本函数使用一个生成器作为数据源预测模型，生成器应返回与test_on_batch的输入数据相同类型的数据。该函数的参数与fit_generator同名参数含义相同，steps是生成器要返回数据的轮数。

3.2 常用层

Keras的层即是网络层的意思。包括了常用层，卷积层，池化层，嵌入层，循环层，融合层，噪音层等多种类别。本书不打算介绍所有的层，只着重介绍与神经网络识别最相关的几类层。如常用层，卷积层，池化层等。

这一节介绍常用层，常用层对应于core模块，core内部定义了一系列常用的网络层，包括全连接、激活层等。

所有的Keras层对象都有如下方法：

- `layer.get_weights()`: 返回层的权重（numpy array）。
- `layer.set_weights(weights)`: 从numpy array中将权重加载到该层中，要求numpy array的形状与`layer.get_weights()`的形状相同。
- `layer.get_config()`: 返回当前层配置信息的字典，层也可以借由配置信息重构。

例如：

```
layer = Dense(32)
config = layer.get_config()
reconstructed_layer = Dense.from_config(config)
```

或者：

```
from keras import layers
config = layer.get_config()
layer = layers.deserialize({'class_name': layer.__class__.__name__, 'config': config})
```

如果层仅有一个计算节点（即该层不是共享层），则可以通过下列方法获得输入张量、输出张量、输入数据的形状和输出数据的形状：

- `layer.input`
- `layer.output`
- `layer.input_shape`
- `layer.output_shape`

如果该层有多个计算节点（参考层计算节点和共享层）。可以使用下面的方法。

- `layer.get_input_at(node_index)`
- `layer.get_output_at(node_index)`
- `layer.get_input_shape_at(node_index)`
- `layer.get_output_shape_at(node_index)`

3.2.1 Dense层

Dense就是常用的全连接层，所实现的运算是 $\text{output} = \text{activation}(\text{dot}(\text{input}, \text{kernel}) + \text{bias})$ 。其中`activation`是逐元素计算的激活函数，`kernel`是本层的权值矩阵，`bias`为偏置向量，只有当`use_bias=True`才会添加。

```
keras.layers.core.Dense(units, activation=None, use_bias=True, kernel_initializer='glorot_unif
```

如果本层的输入数据的维度大于2，则会先被压为与`kernel`相匹配的大小。这里是一个使用示例，将16个节点的层全连接到32个节点：

```
# as first layer in a sequential model:
model = Sequential()
model.add(Dense(32, input_shape=(16,)))
# now the model will take as input arrays of shape (*, 16)
# and output arrays of shape (*, 32)
```

参数

- **units:** 大于0的整数，代表该层的输出维度。
- **activation:** 激活函数，为预定义的激活函数名（参考激活函数），或逐元素（element-wise）的Theano函数。如果不指定该参数，将不会使用任何激活函数（即使用线性激活函数： $a(x)=x$ ）。
- **use_bias:** 布尔值，是否使用偏置项。
- **kernel_initializer:** 权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考`initializers`。
- **bias_initializer:** 权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考`initializers`。
- **kernel_regularizer:** `kernel`权重向量的正则化函数，是`Regularizer`对象。
- **bias_regularizer:** `bias`偏置向量的正则化函数，是`Regularizer`对象。
- **activity_regularizer:** 输出函数或激活函数的正则化函数，为`Regularizer`对象。

- `kernel_constraints`: `kernel`权重向量的约束函数，是`Constraints`对象。
- `bias_constraints`: `bias`偏移向量的约束函数，是`Constraints`对象。

`kernel_regularizer`, `bias_regularizer`, `activity_regularizer`, `kernel_constraints`, `bias_constraints`这些参数一般都使用默认值，不需要专门设置。

输入

形如`(nb_samples, ..., input_shape[1])`的nD张量，最常见的情况为`(nb_samples, input_dim)`的2D张量。

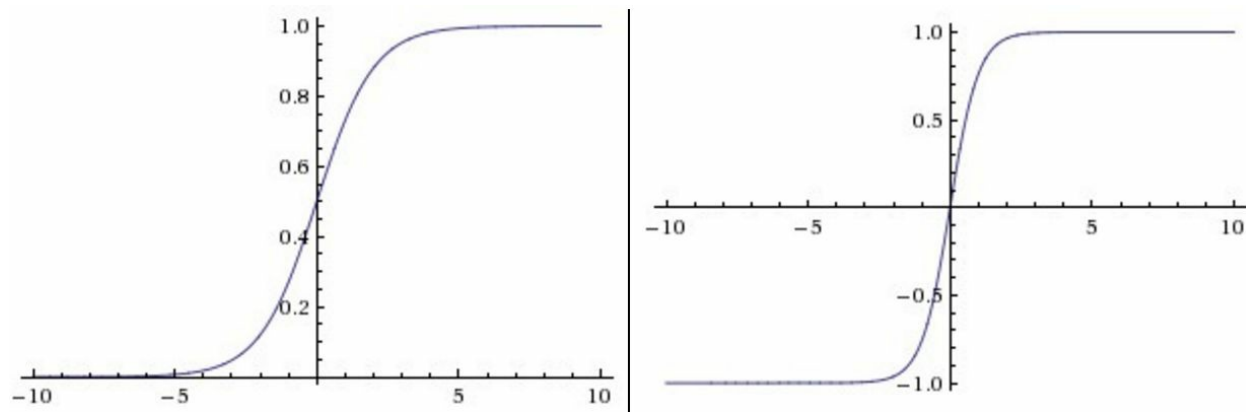
输出

形如`(nb_samples, ..., units)`的nD张量，最常见的情况为`(nb_samples, output_dim)`的2D张量。

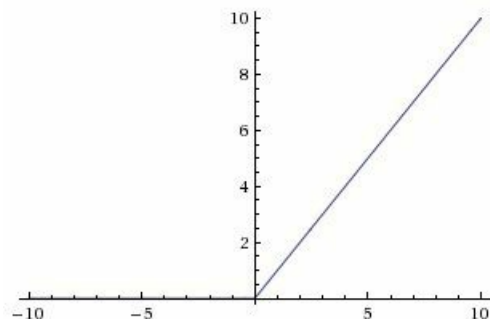
3.2.2 Activation层

`Activation`层用于设置激活函数，激活函数用于在模型中引入非线性。

激活函数`sigmoid`与`tanh`曾经很流行，但现在很少用于视觉模型了，主要原因在于当输入的绝对值较大时，其导数接近于零，梯度的反向传播过程将被中断，出现梯度消散的现象。



ReLU 是一个很好的替代：



相比于 sigmoid 与 tanh，它有两个优势：

- 没有饱和问题，大大缓解了梯度消散的现象，加快了收敛速度。
- 实现起来非常简单，加速了计算过程。

ReLU 有一个缺陷，就是它可能会永远“死”掉：

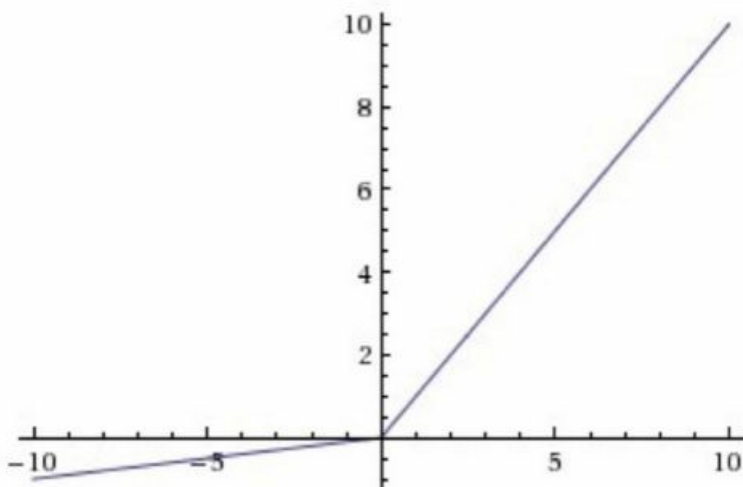
假如有一组二维数据 $X(x_1, x_2)$ 分布在 $x_1:[0,1], x_2:[0,1]$ 的区域内，有一组参数 $W(w_1, w_2)$ 对 X 做线性变换，并将结果输入到 ReLU。

$$F = w_1 * x_1 + w_2 * x_2$$

如果 $w_1 = w_2 = -1$ ，那么无论 X 如何取值， F 必然小于等于零。那么 ReLU 函数对 F 的导数将永远为零。这个 ReLU 节点将永远不参与整个模型的学习过程。

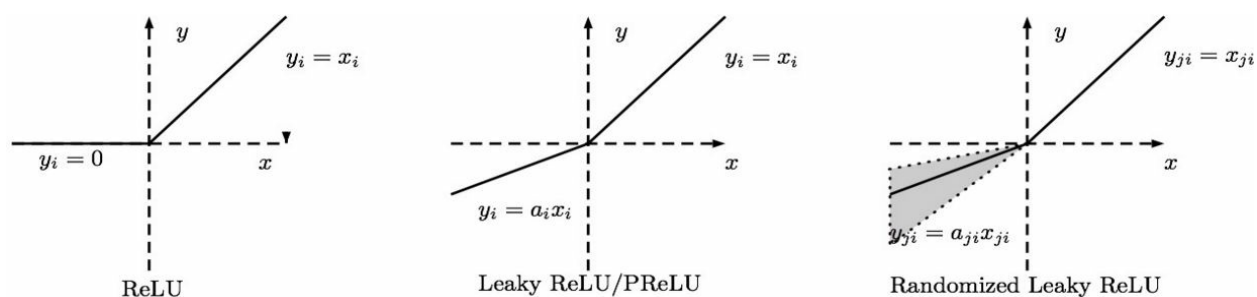
造成上述现象的原因是 ReLU 在负区间的导数为零，为了解决这一问题，人们发明了 Leaky ReLU, Parametric ReLU, Randomized ReLU 等变体。他们的中心思想都是为 ReLU 函数在负区间赋予一定的斜率，从而让其导数不为零（这里设斜率为 α ）。

Leaky ReLU 就是直接给 α 指定一个值，整个模型都用这个斜率：



Parametric ReLU 将 α 作为一个参数，通过学习获取它的最优值。Randomized ReLU 为

alpha 规定一个区间，然后在区间内随机选取 alpha 的值。



在实践中，Parametric ReLU 和 Randomized ReLU 都是可取的。

Keras中设置激活函数使用下列方法：

```
keras.layers.core.Activation(activation)
```

参数

activation: 将要使用的激活函数，为预定义激活函数名或一个Tensorflow/Theano的函数。可选值包括：tanh, relu, elu, softsign, linear, softmax等等，具体可参考预定义激活函数。

输入**shape**

任意，当使用激活层作为第一层时，要指定input_shape。

输出**shape**

与输入shape相同。

预定义激活函数

- softmax: 对输入数据的最后一维进行softmax，输入数据应形如(nb_samples, nb_timesteps, nb_dims)或(nb_samples, nb_dims)
- elu
- softplus
- softsign
- relu
- tanh
- sigmoid

- hard_sigmoid
- linear

高级激活函数

对于简单的Theano/TensorFlow不能表达的复杂激活函数，如含有可学习参数的激活函数，可通过高级激活函数实现，如PReLU，LeakyReLU等。

3.2.3 Dropout层

```
keras.layers.core.Dropout(rate, noise_shape=None, seed=None)
```

为输入数据施加Dropout。Dropout将在训练过程中每次更新参数时随机断开一定百分比（rate）的输入神经元，Dropout层用于防止过拟合。

参数

- rate: 0~1的浮点数，控制需要断开的神经元的比例。
- noise_shape: 整数张量，为将要应用在输入上的二值Dropout mask的shape，例如你的输入为(batch_size, timesteps, features)，并且你希望在各个时间步上的Dropout mask都相同，则可传入noise_shape=(batch_size, 1, features)。
- seed: 整数，使用的随机数种子。

3.2.4 Flatten层

```
keras.layers.core.Flatten()
```

Flatten层用来将输入“压平”，即把多维的输入一维化，常用在从卷积层到全连接层的过渡。Flatten不影响batch的大小。

例子

```
model = Sequential()
model.add(Convolution2D(64, 3, 3,
                        border_mode='same',
                        input_shape=(3, 32, 32)))
# now: model.output_shape == (None, 64, 32, 32)

model.add(Flatten())
# now: model.output_shape == (None, 65536)
```

卷积操作将（3,32,32）格式的图像数据转换成（64,32,32）格式，Flatten的结果则是64*32*32=65536。

3.2.5 Reshape层

```
keras.layers.core.Reshape(target_shape)
```

Reshape层用来将输入shape重新转换为特定大小的shape。

参数

target_shape: 目标shape，为整数的tuple，不包含样本数目的维度（batch大小）。

输入**shape**

任意，但输入的shape必须固定。当使用该层为模型首层时，需要指定input_shape参数。

输出**shape**

(batch_size,)+target_shape

例子

```
model = Sequential()
model.add(Reshape((3, 4), input_shape=(12,)))
model.add(Reshape((6, 2)))
model.add(Reshape((-1, 2, 2)))
# now: model.output_shape == (None, 3, 2, 2)
```

3.2.6 Permute层

```
keras.layers.core.Permute(dims)
```

Permute层将输入的维度按照给定模式进行重排，例如，当需要将RNN和CNN网络连接时，可能会用到该层。

参数

dims: 整数tuple，指定重排的模式，不包含样本数的维度。重排模式的下标从1开始。例如（2，1）代表将输入的第二个维度重拍到输出的第一个维度，而将输入的第一个维度重排到第二个维度。

例子

```
model = Sequential()
```

```
model.add(Permute((2, 1), input_shape=(10, 64)))
```

这时候 model.output_shape 等于 (None, 64, 10)。

输入**shape**

任意，当使用激活层作为第一层时，要指定input_shape。

输出**shape**

与输入相同，但是其维度按照指定的模式重新排列。

3.2.7 RepeatVector层

```
keras.layers.core.RepeatVector(n)
```

RepeatVector层将输入重复n次。

参数

n: 整数，重复的次数。

输入**shape**

形如 (nb_samples, features) 的2D张量。

输出**shape**

形如 (nb_samples, n, features) 的3D张量。

例子

```
model = Sequential()
model.add(Dense(32, input_dim=32))
#这时候 model.output_shape等于 (None, 32)。
#继续执行:
model.add(RepeatVector(3))
#执行完之后 model.output_shape 等于 (None, 3, 32)
```

3.2.8 Lambda层

```
keras.layers.core.Lambda(function, output_shape=None, mask=None, arguments=None)
```

本函数对上一层的输出施以任意Theano/TensorFlow表达式。

参数

- function: 要施加的函数，该函数仅接受一个变量，即上一层的输出。
- output_shape: 函数应该返回的值的shape，可以是一个tuple，也可以是一个根据输入

shape计算得出输出shape的函数。

- mask: 掩膜。
- arguments: 可选，字典，用来记录向函数中传递的其他关键字参数。

例子

```
# add a x -> x^2 layer
model.add(Lambda(lambda x: x ** 2))
# add a layer that returns the concatenation
# of the positive part of the input and
# the opposite of the negative part
def antirectifier(x):
    x -= K.mean(x, axis=1, keepdims=True)
    x = K.l2_normalize(x, axis=1)
    pos = K.relu(x)
    neg = K.relu(-x)
    return K.concatenate([pos, neg], axis=1)
def antirectifier_output_shape(input_shape):
    shape = list(input_shape)
    assert len(shape) == 2 # only valid for 2D tensors
    shape[-1] *= 2
    return tuple(shape)
model.add(Lambda(antirectifier,
                  output_shape=antirectifier_output_shape))
```

输入**shape**

任意，当使用该层作为第一层时，要指定input_shape。

输出**shape**

由output_shape参数指定的输出shape，当使用tensorflow时可自动推断。

3.3 卷积层

以下内容来自于Keras官网，为简单起见，本书只讨论1D和2D卷积，并且举一个2D卷积的例子。其他卷积方式可参考官网文档。

什么叫卷积？比如（3,3）卷积，其含义就是每次将3*3=9个特征值根据卷积算法合并成一个特征值，常用的卷积算法有以下几种：

- 平均值：取9个特征值的平均值作为新的特征值。
- 最大值：取9个特征值中最大值作为新的特征值。
- 最小值：取9个特征值中最小值作为新的特征值。

当然还可以有其他类型的卷积算法。

卷积的时候还有个重要的概念就是步长，用参数strides表示。下面举例说明：

如果步长是(1,1)，那么第一个卷积从[0,0]到[2,2]这9个特征合并成一个值，下一次卷积从[1,1]到[3,3]，对于9*9大小的矩阵，全部卷积后就是7*7大小的矩阵。

3.3.1 Conv1D层

```
keras.layers.convolutional.Conv1D(filters, kernel_size, strides=1, padding='valid', dilation_r
```

一维卷积层（即时域卷积），作用是在一维输入信号上进行邻域滤波。当使用该层作为首层时，需要提供关键参数input_shape。例如(10,128)代表一个长为10的序列，序列中每个信号为128向量。而(None, 128)代表变长的128维向量序列。

该层将输入信号与卷积核按照单一的空域（或时域）方向进行卷积。如果use_bias=True，则还会加上一个偏置项，若activation不为None，则输出为经过激活函数的输出。

参数

- filters: 卷积核的数目（即输出的维度）。
- kernel_size: 整数或由单个整数构成的list/tuple，卷积核的空域或时域窗长度。
- strides: 整数或由单个整数构成的list/tuple，为卷积的步长。任何不为1的滑动窗口。
- padding: 补0策略，为“valid”，“same”或“causal”，“causal”将产生因果（膨胀的）卷积，即output[t]不依赖于input[t+1:]。“valid”代表只进行有效的卷积，即对边界数据不处理。“same”代表保留边界处的卷积结果，通常会导致输出shape与输入shape相同。

- **activation**: 激活函数，为预定义的激活函数名（参考激活函数），或逐元素（element-wise）的Theano函数。如果不指定该参数，将不会使用任何激活函数（即使用线性激活函数： $a(x)=x$ ）。
- **dilation_rate**: 整数或由单个整数构成的list/tuple，指定dilated convolution中的膨胀比例。任何不为1的dilation_rate均与任何不为1的strides不兼容。
- **use_bias**: 布尔值，是否使用偏置项。
- **kernel_initializer**: 权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考initializers。
- **bias_initializer**: 权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考initializers。
- **kernel_regularizer**: 施加在权重上的正则项，为Regularizer对象。
- **bias_regularizer**: 施加在偏置向量上的正则项，为Regularizer对象。
- **activity_regularizer**: 施加在输出上的正则项，为Regularizer对象。
- **kernel_constraints**: 施加在权重上的约束项，为Constraints对象。
- **bias_constraints**: 施加在偏置上的约束项，为Constraints对象。

输入**shape**

形如（samples, steps, input_dim）的3D张量。

输出**shape**

形如（samples, new_steps, nb_filter）的3D张量，因为有向量填充的原因，steps的值会改变。

可以将Convolution1D看作Convolution2D的快捷版，对例子中（10, 32）的信号进行1D卷积相当于对其进行卷积核为（filter_length, 32）的2D卷积。

3.3.2 Conv2D层

```
keras.layers.convolutional.Conv2D(
    filters, kernel_size, strides=(1,1),
    padding='valid',
    data_format=None,
    dilation_rate=(1,1),
    activation=None,
    use_bias=True,
    kernel_initializer='glorot_uniform',
    bias_initializer='zeros',
    kernel_regularizer=None,
    bias_regularizer=None,
```

```
activity_regularizer=None,  
kernel_constraint=None,  
bias_constraint=None)
```

将二维向量进行卷积，当使用该层作为第一层时，应提供input_shape参数。

二维卷积层对二维输入进行滑动窗卷积，当使用该层作为第一层时，应提供input_shape参数。例如input_shape = (128,128,3)代表128*128的彩色RGB图像

（data_format='channels_last'），通道数放在最后一个维度，还有一种是input_shape = (3,128,128)，通道数放在第一个维度。

参数

- **filters:** 卷积核的数目（即输出的维度）。
- **kernel_size:** 单个整数或由两个整数构成的list/tuple，卷积核的宽度和长度。如为单个整数，则表示在各个空间维度的相同长度。
- **strides:** 单个整数或由两个整数构成的list/tuple，为卷积的步长。如为单个整数，则表示在各个空间维度的相同步长。任何不为1的strides均与任何不为1的dilation_rate均不兼容。
- **padding:** 补0策略，为“valid”，“same”。“valid”代表只进行有效的卷积，即对边界数据不处理。“same”代表保留边界处的卷积结果，通常会导致输出shape与输入shape相同。
- **activation:** 激活函数，为预定义的激活函数名（参考激活函数），或逐元素（element-wise）的Theano函数。如果不指定该参数，将不会使用任何激活函数（即使用线性激活函数：a(x)=x）。
- **dilation_rate:** 单个整数或由两个整数构成的list/tuple，指定dilated convolution中的膨胀比例。任何不为1的dilation_rate均与任何不为1的strides均不兼容。
- **data_format:** 字符串，“channels_first”或“channels_last”之一，代表图像的通道维的位置。该参数是Keras 1.x中的image_dim_ordering，“channels_last”对应原本的“tf”，“channels_first”对应原本的“th”。以128x128的RGB图像为例，“channels_first”应将数据组织为（3,128,128），而“channels_last”应将数据组织为（128,128,3）。该参数的默认值是~/.keras/keras.json中设置的值，若从未设置过，则为“channels_last”。
- **use_bias:**布尔值，是否使用偏置项。
- **kernel_initializer:** 权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考initializers。
- **bias_initializer:** 权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考initializers。

- `kernel_regularizer`: 施加在权重上的正则项, 为`Regularizer`对象。
- `bias_regularizer`: 施加在偏置向量上的正则项, 为`Regularizer`对象。
- `activity_regularizer`: 施加在输出上的正则项, 为`Regularizer`对象。
- `kernel_constraints`: 施加在权重上的约束项, 为`Constraints`对象。
- `bias_constraints`: 施加在偏置上的约束项, 为`Constraints`对象。

输入shape

‘channels_first’模式下, 输入形如 (samples, channels, rows, cols) 的4D张量。

‘channels_last’模式下, 输入形如 (samples, rows, cols, channels) 的4D张量。

注意这里的输入shape指的是函数内部实现的输入shape, 而非函数接口应指定的input_shape。

输出shape

‘channels_first’模式下, 为形如 (samples, nb_filter, new_rows, new_cols) 的4D张量。

‘channels_last’模式下, 为形如 (samples, new_rows, new_cols, nb_filter) 的4D张量。

输出的行列数和卷积核的大小以及填充方法有关。

例子

一般我们说一张图片的宽度为width: W, 高度为height: H, 这张图片的通道数为D, 一般目前都用RGB三通道的D=3, 但是为了通用性表示为D。

卷积核大小为K*K, 因为处理的图片是D通道的, 因此卷积核其实也就是K*K*D大小的, RGB三通道, 指定了kernel_size的前提下, 真正的卷积核大小是kernel_size*kernel_size*3。

对于D各通道而言, 是在每个通道上分别执行二维卷积, 然后将D个通道加起来, 得到该位置的二维卷积输出, 对于RGB三通道而言, 就是在R,G,B三个通道上分别使用对应的每个通道上的kernel_size*kernel_size大小的核去卷积每个通道上的W*H的图片, 然后将三个通道卷积得到的输出相加, 得到M个二维卷积输出结果, 在有padding的情况下, 能保持输出图片大小和原来的一样。

在VALID模式下, 抛弃右边不够卷积的元素, 本例子采用默认滑动窗口大小 (1, 1)。

对于Samples为10的‘channels_first’张量, 假设输入shape为 (10, 3, 28, 28), 并设置卷积核数目为32, 卷积核大小为 (3, 3), 则输出张量为 (10, 32, 26, 26)。

对于VALID模式, 用 (3, 3) 卷积核来卷积shape大小为 (28, 28) 的向量时, 最右边的 (2, 2) 元素被抛弃, 因此输出shape为 (26, 26)。

3.4 池化层

与卷积层类似，为简单起见，本章只讨论1D和2D最大池化，其他还有均值池化等，可参考官网文档。

3.4.1 MaxPooling1D层

```
keras.layers.pooling.MaxPooling1D(pool_size=2, strides=None, padding='valid')
```

对时域1D信号进行最大值池化。

参数

pool_size: 整数，池化窗口大小

strides: 整数或None，下采样因子，例如设2将会使得输出shape为输入的一半，若为None则默认值为pool_size。

padding: ‘valid’或者‘same’。

输入**shape**

形如（samples, steps, features）的3D张量。

输出**shape**

形如（samples, downsampled_steps, features）的3D张量。

3.4.2 MaxPooling2D层

```
keras.layers.pooling.MaxPooling2D(pool_size=(2, 2), strides=None, padding='valid', data_format
```

为空域信号施加最大值池化。

参数

- **pool_size:** 整数或长为2的整数tuple，代表在两个方向（竖直，水平）上的下采样因子，如取（2，2）将使图片在两个维度上均变为原长的一半。为整数意为各个维度值相同且为该数字。
- **strides:** 整数或长为2的整数tuple，或者None，步长值。
- **border_mode:** ‘valid’或者‘same’。

- **data_format**: 字符串, “channels_first”或“channels_last”之一, 代表图像的通道维的位置。该参数是Keras 1.x中的image_dim_ordering, “channels_last”对应原本的“tf”, “channels_first”对应原本的“th”。以128x128的RGB图像为例, “channels_first”应将数据组织为(3,128,128), 而“channels_last”应将数据组织为(128,128,3)。该参数的默认值是~/.keras/keras.json中设置的值, 若从未设置过, 则为“channels_last”。

输入**shape**

‘channels_first’模式下, 为形如(samples, channels, rows, cols)的4D张量。

‘channels_last’模式下, 为形如(samples, rows, cols, channels)的4D张量。

输出**shape**

‘channels_first’模式下, 为形如(samples, channels, pooled_rows, pooled_cols)的4D张量。

‘channels_last’模式下, 为形如(samples, pooled_rows, pooled_cols, channels)的4D张量。

例子

对(3, 28, 28)的向量, 用(2, 2)池化的结果是(3, 14, 14)向量。

对(3, 28, 28)的向量, 用(4, 4)池化的结果是(3, 7, 7)向量。

4 目标函数

目标函数，或称损失函数，是编译一个神经网络模型必须的两个参数之一。例如：

```
model.compile(loss='mean_squared_error', optimizer='sgd')
```

可以通过传递预定义目标函数名字指定目标函数，也可以传递一个Theano/TensorFlow的符号函数作为目标函数，该函数对每个数据点应该只返回一个标量值，并以下列两个参数为参数：

- `y_true`：真实的数据标签，Theano/TensorFlow张量。
- `y_pred`：预测值，与`y_true`相同shape的Theano/TensorFlow张量。

Keras中支持多种目标函数算法，比较常用的有MSE，MAE，`binary_crossentropy`等几种，下面就简要介绍这几种目标函数的定义。

4.1 MSE

MSE: Mean Squared Error, 均方误差, 又叫mean_squared_error。

代价函数经常用方差代价函数（即采用均方误差MSE），比如对于一个神经元（单输入单输出，sigmoid函数），定义其代价函数为：

$$C = \frac{(y - a)^2}{2}$$

其中y是我们期望的输出，a为神经元的实际输出【 $a = \sigma(z)$, where $z = wx + b$ 】。

在训练神经网络过程中，我们通过梯度下降算法来更新w和b，因此需要计算代价函数对w和b的导数：

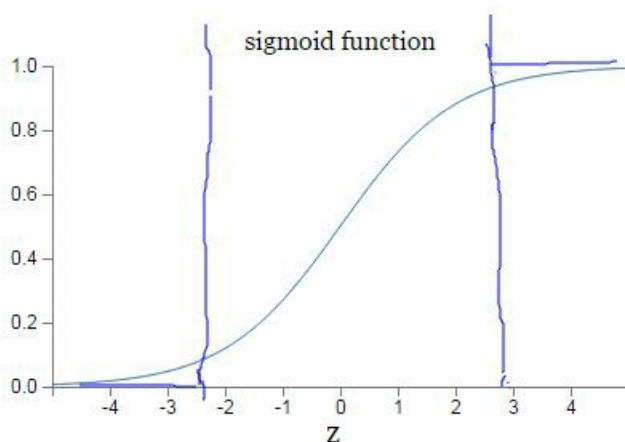
$$\begin{aligned}\frac{\partial C}{\partial w} &= (a - y)\sigma'(z)x = a\sigma'(z) \\ \frac{\partial C}{\partial b} &= (a - y)\sigma'(z) = a\sigma'(z),\end{aligned}$$

然后更新w、b：

$$w \leftarrow w - \eta * \frac{\partial C}{\partial w} = w - \eta * a * \sigma'(z)$$

$$b \leftarrow b - \eta * \frac{\partial C}{\partial b} = b - \eta * a * \sigma'(z)$$

因为sigmoid函数的性质，导致 $\sigma'(z)$ 在z取大部分值时会很小（如下图标出来的两端，几近于平坦），这样会使得w和b更新非常慢（因为 $\eta * a * \sigma'(z)$ 这一项接近于0）。



4.2 MAE

mae或mean_absolute_error，绝对误差，它的定义为：

```
return K.mean(K.abs(y_pred - y_true), axis=-1)
```

取预测值和真实值绝对误差的平均数作为损失函数。

4.3 MAPE

mape或mean_absolute_percentage_error。

公式为：

$$(|(y_{\text{true}} - y_{\text{pred}})| / \text{clip}(|y_{\text{true}}|, \text{epsilon}, \text{infinite})) . \text{mean}(\text{axis}=-1) * 100$$

和mae的区别就是，累加的是预测值与实际值的差，除以实际值（剔除不介于epsilon和infinite之间的实际值），然后求均值。

4.4 MSLE

msle或mean_squared_logarithmic_error。

公式为：

$$(\log(\text{clip}(y_{\text{pred}}, \text{epsilon}, \text{infinite})+1) - \log(\text{clip}(y_{\text{true}}, \text{epsilon}, \text{infinite})+1))^2.\text{mean}(\text{axis}=-1)$$

这个就是加入了log对数，剔除不介于epsilon和infinite之间的预测值与实际值之后，然后取对数，作差，平方，累加求均值。

4.5 squared_hinge

公式为: $(\max(1 - y_{\text{true}} * y_{\text{pred}}, 0))^2$.mean(axis=-1), 取1减去预测值与实际值乘积的结果与0比相对大的值的平方的累加均值。

4.6 hinge

公式为: $(\max(1 - y_{\text{true}} * y_{\text{pred}}, 0)).\text{mean}(\text{axis}=-1)$, 取1减去预测值与实际值乘积的结果与0比相对大的值的的累加均值。

4.7 binary_crossentropy

亦称对数损失，或logloss，常说的逻辑回归，就是常用的交叉熵函数。

交叉熵代价函数（下面的公式对应一个神经元，多输入单输出）：

$$C = -\frac{1}{n} \sum_x [y \ln a + (1 - y) \ln(1 - a)]$$

其中y为期望的输出，a为神经元实际输出【 $a = \sigma(z)$, where $z = \sum W_j * X_j + b$ 】

与方差代价函数一样，交叉熵代价函数同样有两个性质：

- 非负性。（所以我们的目标就是最小化代价函数）。
- 当真实输出a与期望输出y接近的时候，代价函数接近于0.(比如y=0, $a \sim 0$; y=1, $a \sim 1$ 时，代价函数都接近0)。

另外，它可以克服方差代价函数更新权重过慢的问题。我们同样看看它的导数：

$$\frac{\partial C}{\partial w_j} = \frac{1}{n} \sum_x x_j (\sigma(z) - y).$$

$$\frac{\partial C}{\partial b} = \frac{1}{n} \sum_x (\sigma(z) - y).$$

可以看到，导数中没有 $\sigma'(z)$ 这一项，权重的更新是受 $\sigma(z)-y$ 这一项的影响，即受误差的影响。所以当误差大的时候，权重更新就快，当误差小的时候，权重的更新就慢。这是一个很好的性质。

4.8 categorical_crossentropy

亦称多类的对数损失，注意使用该目标函数时，需要将标签转化为形如(nb_samples, nb_classes)的二值序列。

注意：当使用"category_crossentropy"作为目标函数时,标签应该为多类模式，即one-hot编码的向量，而不是单个数值。可以使用工具中的to_categorical函数完成该转换，示例如下：

```
from keras.utils.np_utils import to_categorical
categorical_labels = to_categorical(int_labels, num_classes=None)
```

4.9 `sparse_categorical_crossentropy`

如上，但接受稀疏标签。注意，使用该函数时仍然需要你的标签与输出值的维度相同，你可能需要在标签数据上增加一个维度：`np.expand_dims(y,-1)`。

5 优化器

优化问题指的是，给定目标函数 $f(x)$ ，我们需要找到一组参数 x ，使得 $f(x)$ 的值最小。

本文以下内容假设读者已经了解机器学习基本知识和梯度下降的原理。

优化器optimizers是编译Keras模型必要的两个参数之一，例如：

```
from keras import optimizers
model = Sequential()
model.add(Dense(64, init='uniform', input_shape=(10,)))
model.add(Activation('tanh'))
model.add(Activation('softmax'))
sgd = optimizers.SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='mean_squared_error', optimizer=sgd)
```

可以在调用`model.compile()`之前初始化一个优化器对象，然后传入该函数（如上所示），也可以在调用`model.compile()`时传递一个预定义优化器名。在后者的情形下，优化器的参数将使用默认值。

5.1 公共参数

参数clipnorm和clipvalue是所有优化器都可以使用的参数，用于对梯度进行裁剪。clipnorm取值范围在0~1.0之间，clipvalue取值范围在-0.5~0.5之间，示例如下：

```
from keras import optimizers
# All parameter gradients will be clipped to
# a maximum norm of 1.
sgd = optimizers.SGD(lr=0.01, clipnorm=1.)
```

5.2 SGD

SGD指stochastic gradient descent，即随机梯度下降，随机的意思是随机选取部分数据集参与计算，是梯度下降的batch版本。SGD支持动量参数，支持学习衰减率。

用法

```
keras.optimizers.SGD(lr=0.01, momentum=0.0, decay=0.0, nesterov=False)
```

参数

- **lr**: 大于0的浮点数，学习率
- **momentum**: 大于0的浮点数，动量参数
- **decay**: 大于0的浮点数，每次更新后的学习率衰减值
- **nesterov**: 布尔值，确定是否使用Nesterov动量

对于训练数据集，我们首先将其分成n个batch，每个batch包含m个样本。我们每次更新都利用一个batch的数据，而非整个训练集，即：

$$x_{t+1} = x_t + \Delta x_t$$

$$\Delta x_t = -\eta g_t$$

其中， η 为学习率， g_t 为 x 在 t 时刻的梯度。

这么做的好处在于：

- 当训练数据太多时，利用整个数据集更新往往时间上不显示。batch的方法可以减少机器的压力，并且可以更快地收敛。
- 当训练集有很多冗余时（类似的样本出现多次），batch方法收敛更快。以一个极端情况为例，若训练集前一半和后一半梯度相同。那么如果前一半作为一个batch，后一半作为另一个batch，那么在一次遍历训练集时，batch的方法向最优解前进两个step，而整体的方法只前进一个step。

5.3 RMSprop

RMSProp通过引入一个衰减系数，让r每回合都衰减一定比例，类似于Momentum中的做法，该优化器通常是面对递归神经网络时的一个良好选择。

具体实现：

需要：全局学习速率 ϵ ，初始参数 θ ，数值稳定量 δ ，衰减速率 ρ

中间变量：梯度累计量 r (初始化为0)

每步迭代过程：

01. 从训练集中的随机抽取一批容量为 m 的样本 $\{x_1, \dots, x_m\}$ 以及相关的输出 y_i 。

02. 计算梯度和误差，更新 r ，再根据 r 和梯度计算参数更新量。

$$\begin{aligned}\hat{g} &\leftarrow + \frac{1}{m} \nabla_{\theta} \sum_i L(f(x_i; \theta), y_i) \\ r &\leftarrow \rho r + (1 - \rho) \hat{g} \odot \hat{g} \\ \Delta \theta &= - \frac{\epsilon}{\delta + \sqrt{r}} \odot \hat{g} \\ \theta &\leftarrow \theta + \Delta \theta\end{aligned}$$

用法

```
keras.optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=1e-06)
```

参数

- lr: 大于0的浮点数，学习率。
- rho: 大于0的浮点数。
- epsilon: 大于0的小浮点数，防止除0错误。

5.4 Adagrad

AdaGrad可以自动变更学习速率，只需要设定一个全局的学习速率 ϵ ，但是这并非是实现学习速率，实际的速率是与以往参数的模之和的开方成反比的。也许说起来有点绕口，不过用公式来表示就直白得多；

$$\epsilon_n = \frac{\epsilon}{\delta + \sqrt{\sum_{i=1}^{n-1} g_i \odot g_i}}$$

其中 δ 是一个很小的常量,大概在 10^{-7} ,防止出现除以0的情况。

具体实现：

需要:全局学习速率 ϵ ，初始参数 θ ，数值稳定量 δ 。

中间变量：梯度累计量 r (初始化为0)。

每步迭代过程：

01. 从训练集中的随机抽取一批容量为 m 的样本 $\{x_1, \dots, x_m\}$ 以及相关的输出 y_i 。

02. 计算梯度和误差，更新 r ，再根据 r 和梯度计算参数更新量。

$$\begin{aligned}\hat{g} &\leftarrow + \frac{1}{m} \nabla_{\theta} \sum_i L(f(x_i; \theta), y_i) \\ r &\leftarrow r + \hat{g} \odot \hat{g} \\ \Delta\theta &= - \frac{\epsilon}{\delta + \sqrt{r}} \odot \hat{g} \\ \theta &\leftarrow \theta + \Delta\theta\end{aligned}$$

优点

能够实现学习率的自动更改。如果这次梯度大，那么学习速率衰减的就快一些；如果这次梯度小，那么学习速率衰减的就慢一些。

缺点

任然要设置一个变量 ϵ 。

经验表明，在普通算法中也许效果不错，但在深度学习中，深度过深时会造成训练提前结束。

用法：

```
keras.optimizers.Adagrad(lr=0.01, epsilon=1e-06)
```


参数

- lr: 大于0的浮点数，学习率。
- epsilon: 大于0的小浮点数，防止除0错误。

5.5 Adadelta

Adagrad算法存在三个问题：

- 其学习率是单调递减的，训练后期学习率非常小。
- 其需要手工设置一个全局的初始学习率。
- 更新 x_t 时，左右两边的单位不同一。

Adadelta针对上述三个问题提出了比较漂亮的解决方案。

首先，针对第一个问题，我们可以只使用adagrad的分母中的累计项离当前时间点比较近的项，如下式：

$$E[g^2]_t = \rho E[g^2]_{t-1} + (1 - \rho)g_t^2$$

$$\Delta x_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

这里 ρ 是衰减系数，通过这个衰减系数，我们令每一个时刻的 g_t 随之时间按照 ρ 指数衰减，这样就相当于我们仅使用离当前时刻比较近的 g_t 信息，从而使得还很长时间之后，参数仍然可以得到更新。

针对第三个问题，其实sgd跟momentum系列的方法也有单位不统一的问题。sgd、momentum系列方法中：

$$\Delta x \text{ 的单位} \propto g \text{ 的单位} \propto \frac{\partial f}{\partial x} \propto \frac{1}{x \text{ 的单位}}$$

类似的，adagrad中，用于更新 Δx 的单位也不是 x 的单位，而是1。

而对于牛顿迭代法：

$$\Delta x = H_t^{-1} g_t$$

其中 H 为Hessian矩阵，由于其计算量巨大，因而实际中不常使用。其单位为：

$$\Delta x \propto H^{-1} g \propto \frac{\frac{\partial f}{\partial x}}{\frac{\partial^2 f}{\partial^2 x}} \propto x \text{ 的单位}$$

注意，这里 f 无单位。因而，牛顿迭代法的单位是正确的。

所以，我们可以模拟牛顿迭代法来得到正确的单位。注意到：

$$\Delta x = \frac{\frac{\partial f}{\partial x}}{\frac{\partial^2 f}{\partial^2 x}} \Rightarrow \frac{1}{\frac{\partial^2 f}{\partial^2 x}} = \frac{\Delta x}{\frac{\partial f}{\partial x}}$$

这里，在解决学习率单调递减的问题的方案中，分母已经是 $\partial f / \partial x$ 的一个近似了。这里我们可以构造 Δx 的近似，来模拟得到 H^{-1} 的近似，从而得到近似的牛顿迭代法。具体做法如下：

$$\Delta x_t = - \frac{\sqrt{E[\Delta x^2]_{t-1}}}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

可以看到，如此一来adagrad中分子部分需要人工设置的初始学习率也消失了，从而顺带解决了上述的第二个问题。

用法

```
keras.optimizers.Adadelta(lr=1.0, rho=0.95, epsilon=1e-06)
```

建议保持优化器的默认参数不变。

参数

- **lr**: 大于0的浮点数，学习率。
- **rho**: 大于0的浮点数。
- **epsilon**: 大于0的小浮点数，防止除0错误。

5.6 Adam

Adam是一种基于一阶梯度来优化随机目标函数的算法。

Adam这个名字来源于adaptive moment estimation，自适应矩估计。概率论中矩的含义是：如果一个随机变量 X 服从某个分布， X 的一阶矩是 $E(X)$ ，也就是样本平均值， X 的二阶矩就是 $E(X^2)$ ，也就是样本平方的平均值。Adam 算法根据损失函数对每个参数的梯度的一阶矩估计和二阶矩估计动态调整针对于每个参数的学习速率。Adam 也是基于梯度下降的方法，但是每次迭代参数的学习步长都有一个确定的范围，不会因为很大的梯度导致很大的学习步长，参数的值比较稳定。

Adam(Adaptive Moment Estimation)本质上是带有动量项的RMSprop，它利用梯度的一阶矩估计和二阶矩估计动态调整每个参数的学习率。Adam的优点主要在于经过偏置校正后，每一次迭代学习率都有个确定范围，使得参数比较平稳。

具体实现：

需要：步进值 ϵ ，初始参数 θ ，数值稳定量 δ ，一阶动量衰减系数 ρ_1 ，二阶动量衰减系数 ρ_2 。

其中几个取值一般为： $\delta=10^{-8}, \rho_1=0.9, \rho_2=0.999$ 。

中间变量：一阶动量 s ，二阶动量 r ，都初始化为0。

每步迭代过程：

01. 从训练集中的随机抽取一批容量为 m 的样本 $\{x_1, \dots, x_m\}$ ，以及相关的输出 y_i 。
02. 计算梯度和误差，更新 r 和 s ，再根据 r 和 s 以及梯度计算参数更新量。

$$\begin{aligned} g &\leftarrow +\frac{1}{m} \nabla_{\theta} \sum_i L(f(x_i; \theta), y_i) \\ s &\leftarrow \rho_1 s + (1 - \rho_1) g \\ r &\leftarrow \rho_2 r + (1 - \rho_2) g \odot g \\ \hat{s} &\leftarrow \frac{s}{1 - \rho_1} \\ \hat{r} &\leftarrow \frac{r}{1 - \rho_2} \\ \Delta \theta &= -\epsilon \frac{\hat{s}}{\sqrt{\hat{r}} + \delta} \\ \theta &\leftarrow \theta + \Delta \theta \end{aligned}$$

用法

```
keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-08)
```

该优化器的默认值来源于参考文献。

参数

- lr: 大于0的浮点数, 学习率。
- beta_1/beta_2: 浮点数, $0 < \text{beta} < 1$, 通常很接近1。
- epsilon: 大于0的小浮点数, 防止除0错误。

5.7 Adamax

```
keras.optimizers.Adamax(lr=0.002, beta_1=0.9, beta_2=0.999, epsilon=1e-08)
```

Adamax优化器来自于Adam的论文的Section7，该方法是基于无穷范数的Adam方法的变体。

默认参数由论文提供。

参数

- lr: 大于0的浮点数，学习率。
- beta_1/beta_2: 浮点数， $0 < \text{beta} < 1$ ，通常很接近1。
- epsilon: 大于0的小浮点数，防止除0错误。

5.8 Nadam

```
keras.optimizers.Nadam(lr=0.002,beta_1=0.9,beta_2=0.999,epsilon=1e-08,schedule_decay=0.004)
```

Nesterov Adam optimizer: Adam本质上像是带有动量项的RMSprop，Nadam就是带有Nesterov 动量的Adam RMSprop。

默认参数来自于论文，推荐不要对默认参数进行更改。

参数

- lr: 大于0的浮点数，学习率。
- beta_1/beta_2: 浮点数， $0 < \text{beta} < 1$ ，通常很接近1。
- epsilon: 大于0的小浮点数，防止除0错误。

6 训练模型的注意事项

6.1 参数初始化

6.1.1 零初始化

在使用神经网络的时候，注意不要将参数全部初始化为零。

几乎所有的CNN网络都是对称结构，将参数零初始化会导致流过网络的数据也是对称的（都是零），并且没有办法在不受扰动的情況下打破这种数据对称，从而导致网络无法学习。

6.1.2 随机初始化

打破对称性的思路很简单，给每个参数随机赋予一个接近零的值就好了：

```
W = 0.01 * numpy.random.randn(D,H)
```

`randn` 方法生成一个均值为零，方差为1的服从正态分布的随机数。

把 W 应用到之前的表达式 S ， S 就是多个随机变量的加权和。独立随机变量和的方差为：

$$\text{Var}(A+B+C) = \text{Var}(A) + \text{Var}(B) + \text{Var}(C)$$

假设 W 各元素之间相互独立，随着数据维度的增长， S 的方差将会线性累积。根据任务的不同，数据的维度从小到大跨度非常大，是不可控的，所以我们希望将 S 的方差做归一化，这只需要对 W 动动手脚就可以了：

$$W = \text{numpy.random.randn}(n) / \text{sqrt}(n)$$

其中 n 就是数据的维度。

推导过程如下：

$$\begin{aligned}
\text{Var}(s) &= \text{Var}\left(\sum_i^n w_i x_i\right) \\
&= \sum_i^n \text{Var}(w_i x_i) \\
&= \sum_i^n [E(w_i)]^2 \text{Var}(x_i) + E[(x_i)]^2 \text{Var}(w_i) + \text{Var}(x_i) \text{Var}(w_i) \\
&= \sum_i^n \text{Var}(x_i) \text{Var}(w_i) \\
&= (n \text{Var}(w)) \text{Var}(x)
\end{aligned}$$

令 $n \cdot \text{Var}(W) = 1$ ，就得到 $\text{std}(W) = 1 / \sqrt{n}$ 。

在实际使用中，如果结合 ReLU，这篇文章推荐：

```
w = numpy.random.randn(n) * sqrt(2.0/n)
```

6.2 数据预处理（Data Preprocessing）

6.2.1 零均值化（Mean subtraction）

为什么要零均值化？

- 人们对图像信息的摄取通常不是来自于像素色值的高低，而是来自于像素之间的相对色差。零均值化并没有消除像素之间的相对差异（交流信息），仅仅是去掉了直流信息的影响。
- 数据有过大的均值也可能导致参数的梯度过大。
- 如果有后续的处理，可能要求数据零均值，比如PCA。

假设数据存放在一个矩阵 X 中， X 的形状为 (N, D) ， N 是样本个数， D 是样本维度，零均值化操作可用 python 的 `numpy` 来实现：

```
X -= numpy.mean(X, axis=0)
```

即 X 的每一列都减去该列的均值。

对于灰度图像，也可以减去整张图片的均值：

```
X -= numpy.mean(X)
```

对于彩色图像，将以上操作在3个颜色通道内分别进行即可。

6.2.2 归一化（Normalization）

为什么要归一化？

归一化是为了让不同纬度的数据具有相同的分布规模。

假如二维数据数据 (x_1, x_2) 两个维度都服从均值为零的正态分布，但是 x_1 方差为100， x_2 方差为1。可以想像对 (x_1, x_2) 进行随机采样并在而为坐标系中标记后的图像，应该是一个非常狭长的椭圆形。

对这些数据做特征提取会用到以下形式的表达式：

$$S = w_1 * x_1 + w_2 * x_2 + b$$

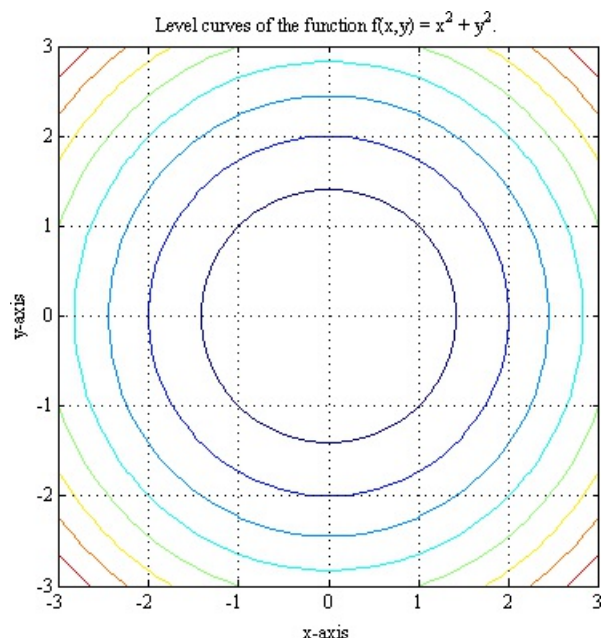
那么：

$$dS / dw_1 = x_1$$

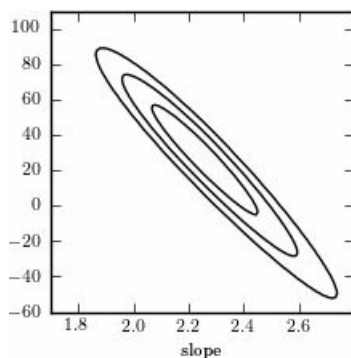
$$dS / dw_2 = x_2$$

由于 x_1 与 x_2 在分布规模上的巨大差异， w_1 与 w_2 的导数也会差异巨大。此时绘制目标函数（不是 S ）的曲面图，就像一个深邃的峡谷，沿着峡谷方向变化的是 w_2 ，坡度很小；在峡谷垂直方向变化的是 w_1 ，坡度非常陡峭。

因为我们期望的目标函数是这样的：



而现在的目标函数可能是这样的：



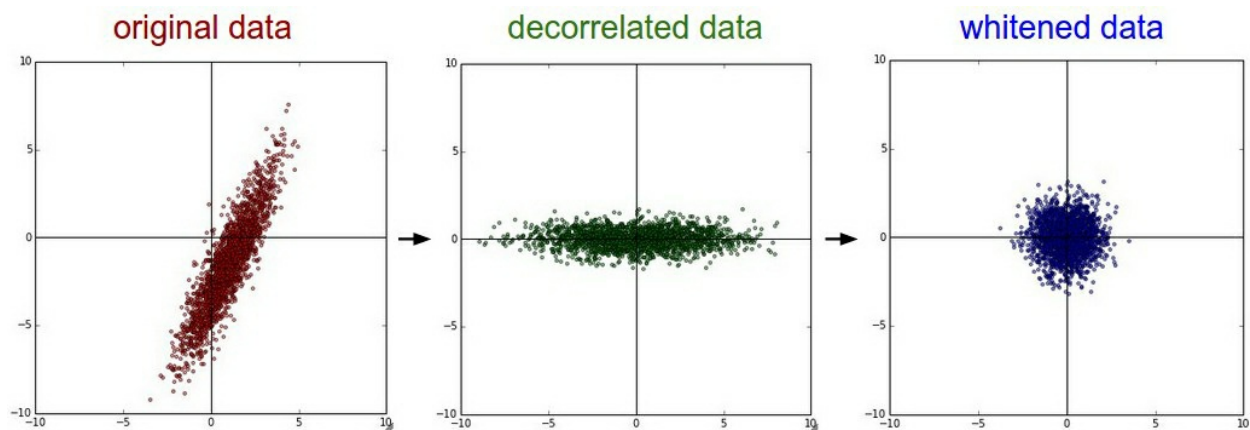
我们知道这样的目标函数是非常难以优化的。因为 w_1 与 w_2 的梯度差异太大，在两个维度上需要不同的迭代方案。但是在实际操作中，为了简便，我们通常为所有维度设置相同的步长，随着迭代的进行，步长的缩减在不同维度间也是同步的。这就要求 W 不同维度的分布规模大致相同，而这一切都始于数据的归一化。

一个典型的归一化实现：

```
X /= numpy.std(X, axis = 0)
```

在自然图像上进行训练时，可以不进行归一化操作，因为（理论上）图像任一部分的统计性质都应该和其它部分相同，图像的这种特性被称作平稳性（stationarity）。

6.2.3 PCA & Whitening

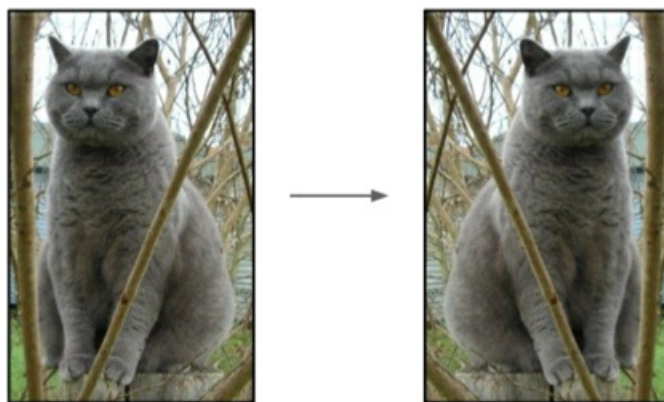


白化相当于在零均值化与归一化操作之间插入一个旋转操作，将数据投影在主轴上。一张图片在经过白化后，可以认为各个像素之间是统计独立的。

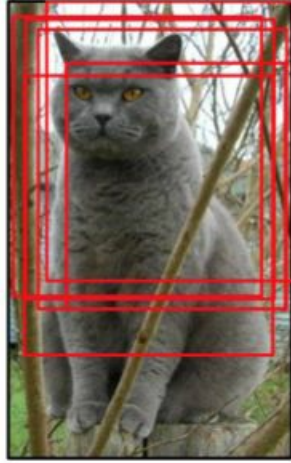
然而白化很少在卷积神经网络中使用。我猜测是因为图像信息本来就是依靠像素之间的相对差异来体现的，白化让像素间去相关，让这种差异变得不确定，抹掉了很多信息。

6.2.4 数据扩充（Data Augmentation）

大型模型往往需要大量数据来训练。一些数据扩充方法因此被发明出来，以自然图像为例：



Flip horizontally



Random crops/scales



Color jittering

可以对图片做水平翻转、进行一定程度的位移或者剪裁，还可以对它的颜色做一定程度的调整。在不改变图像类别的情况下，增加数据量，还能提高模型的泛化能力。

7 图片预处理

7.1 利用小数据量

为了尽量利用我们有限的训练数据，我们将通过一系列随机变换对数据进行提升，这样我们的模型将看不到任何两张完全相同的图片，这有利于我们抑制过拟合，使得模型的泛化能力更好。

在Keras中，这个步骤可以通过`keras.preprocessing.image.ImageDataGenerator`来实现，这个类使你可以：

- 在训练过程中，设置要实行的随机变换
- 通过`.flow`或`.flow_from_directory(directory)`方法实例化一个针对图像batch的生成器，这些生成器可以被用作keras模型相关方法的输入，如`fit_generator`，`evaluate_generator`和`predict_generator`

现在我们看个例子：

```
from keras.preprocessing.image import ImageDataGenerator
datagen = ImageDataGenerator(
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    rescale=1./255,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest')
```

上面显示的只是一部分选项，请阅读文档的相关部分来查看全部可用的选项。我们来快速的浏览一下这些选项的含义：

- `rotation_range`是一个0~180的度数，用来指定随机选择图片的角度。
- `width_shift`和`height_shift`用来指定水平和竖直方向随机移动的程度，这是两个0~1之间的比例。
- `rescale`值将在执行其他处理前乘到整个图像上，我们的图像在RGB通道都是0~255的整数，这样的操作可能使图像的值过高或过低，所以我们将这个值定为0~1之间的数。
- `shear_range`是用来进行剪切变换的程度，参考剪切变换。
- `zoom_range`用来进行随机的放大。

- `horizontal_flip`随机的对图片进行水平翻转，这个参数适用于水平翻转不影响图片语义的时候。
- `fill_mode`用来指定当需要进行像素填充，如旋转，水平和竖直位移时，如何填充新出现的像素。

下面我们使用这个工具来生成图片，并将它们保存在一个临时文件夹中，这样我们可以感受一下数据提升究竟做了什么事。为了使图片能够展示出来，这里没有使用`rescaling`。

```
from keras.preprocessing.image import ImageDataGenerator, array_to_img, img_to_array, load_img

datagen = ImageDataGenerator(
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest')

img = load_img('data/image.0.jpg')
x = img_to_array(img)
x = x.reshape((1,) + x.shape)
i = 0
for batch in datagen.flow(x, batch_size=1,
                          save_to_dir='preview', save_prefix='cat', save_format='jpeg'):
    i += 1
    if i > 20:
        break
```

下面是一张图片被提升以后得到的多个结果：



在小数据集上训练神经网络：40行代码达到80%的准确率

然后我们开始准备数据，使用`.flow_from_directory()`来从我们的jpgs图片中直接产生数据和标签。

```
# this is the augmentation configuration we will use for training
train_datagen = ImageDataGenerator(
    rescale=1./255,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True)
```

```
test_datagen = ImageDataGenerator(rescale=1./255)
train_generator = train_datagen.flow_from_directory(
    'data/train',
    target_size=(150, 150),
    batch_size=32,
    class_mode='binary')
validation_generator = test_datagen.flow_from_directory(
    'data/validation',
    target_size=(150, 150),
    batch_size=32,
    class_mode='binary')
```

然后我们可以用这个生成器来训练网络了，在GPU上每个epoch耗时20~30秒，在CPU上耗时300~400秒，所以如果你不是很着急，在CPU上跑这个模型也是完全可以的。

```
model.fit_generator(
    train_generator,
    samples_per_epoch=2000,
    nb_epoch=50,
    validation_data=validation_generator,
    nb_val_samples=800)
model.save_weights('first_try.h5')
```

这个模型在50个epoch后的准确率为79%~81%，别忘了我们只用了8%的数据，也没有花时间来模型和超参数的优化。

注意这个准确率的变化可能会比较大，因为准确率本来就是一个变化较高的评估参数，而且我们只有800个样本用来测试。比较好的验证方法是使用K折交叉验证，但每轮验证中我们都要训练一个模型。

- 为了进行fine-tune,所有的层都应该以训练好的权重为初始值，例如，你不能将随机初始的全连接放在预训练的卷积层之上，这是因为由随机权重产生的大地图将会破坏卷积层预训练的权重。在我们的情形中，这就是为什么我们首先训练顶层分类器，然后再基于它进行fine-tune的原因。
- 我们选择只fine-tune最后的卷积块，而不是整个网络，这是为了防止过拟合。整个网络具有巨大的熵容量，因此具有很高的过拟合倾向。由底层卷积模块学习到的特征更加一般，更加不具有抽象性，因此我们要保持前两个卷积块（学习一般特征）不动，只fine-tune后面的卷积块（学习特别的特征）。
- fine-tune应该在很低的学习率下进行，通常使用SGD优化而不是其他自适应学习率的优化算法，如RMSProp。这是为了保证更新的幅度保持在较低的程度，以免毁坏预训练的特征。

代码如下，首先在初始化好的vgg网络上添加我们预训练好的模型：

```
# build a classifier model to put on top of the convolutional model
top_model = Sequential()
top_model.add(Flatten(input_shape=model.output_shape[1:]))
top_model.add(Dense(256, activation='relu'))
```



```
top_model.add(Dropout(0.5))
top_model.add(Dense(1, activation='sigmoid'))

# note that it is necessary to start with a fully-trained
# classifier, including the top classifier,
# in order to successfully do fine-tuning
top_model.load_weights(top_model_weights_path)
model.add(top_model)
```

然后将最后一个卷积块前的卷积层参数冻结，不让它参加训练（将trainable属性设置为false）：

```
for layer in model.layers[:25]:
    layer.trainable = False

model.compile(loss='binary_crossentropy',
              optimizer=optimizers.SGD(lr=1e-4, momentum=0.9),
              metrics=['accuracy'])
```

然后以很低的学习率进行训练：

```
# prepare data augmentation configuration
train_datagen = ImageDataGenerator(
    rescale=1./255,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True)
test_datagen = ImageDataGenerator(rescale=1./255)
train_generator = train_datagen.flow_from_directory(
    train_data_dir,
    target_size=(img_height, img_width),
    batch_size=32,
    class_mode='binary')
validation_generator = test_datagen.flow_from_directory(
    validation_data_dir,
    target_size=(img_height, img_width),
    batch_size=32,
    class_mode='binary')
# fine-tune the model
model.fit_generator(
    train_generator,
    samples_per_epoch=nb_train_samples,
    nb_epoch=nb_epoch,
    validation_data=validation_generator,
    nb_val_samples=nb_validation_samples)
```

在50个epoch之后该方法的准确率为94%，非常成功。

通过下面的方法你可以达到95%以上的正确率：

- 更加强烈的数据提升。

- 更加强烈的dropout。
- 使用L1和L2正则项（也称为权重衰减）。
- fine-tune更多的卷积块（配合更大的正则）。

7.2 图像处理 `scipy.ndimage`

`scipy.ndimage`是一个处理多维图像的函数库，它其中又包括以下几个模块：

- `filters`：图像滤波器。
- `fourier`：傅立叶变换。
- `interpolation`：图像的插值、旋转以及仿射变换等。
- `measurements`：图像相关信息的测量。
- `morphology`：形态学图像处理。

更强大的图像处理库。

`scipy.ndimage`只提供了一些基础的图像处理功能，下面是一些更强大的图像处理库：

- OpenCV
- SimpleCV
- scikit-image
- Pillow

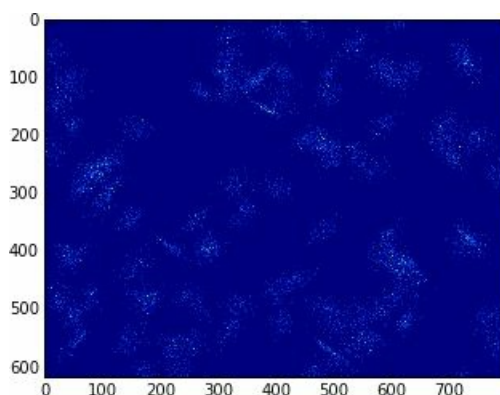
7.3 热点图

首先载入地图图片，并创建一些随机分布的散列点，这些散列点以某些坐标为中心正态分布，构成一些热点。使用`numpy.histogram2d()`可以在地图图片的网格中统计二维散列点的频度。由于散列点数量较少，`histogram2d()`的结果并不能形成足够的热点信息：

```
img = plt.imread("images/china010.png")
h, w, _ = img.shape

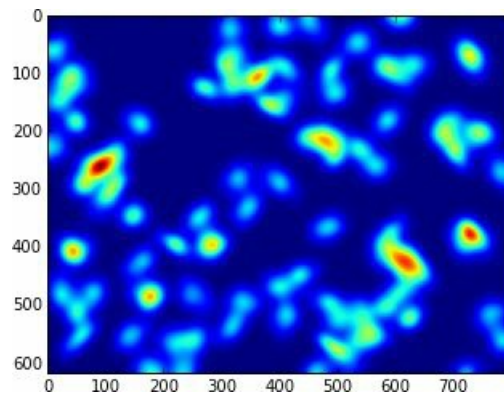
xs, ys = [], []
for i in range(100):
    mean = w*np.random.rand(), h*np.random.rand()
    a = 50 + np.random.randint(50, 200)
    b = 50 + np.random.randint(50, 200)
    c = (a + b)*np.random.normal()*0.2
    cov = [[a, c], [c, b]]
    count = 200
    x, y = np.random.multivariate_normal(mean, cov, size=count).T
    xs.append(x)
    ys.append(y)
x = np.concatenate(xs)
y = np.concatenate(ys)

hist, _, _ = np.histogram2d(x, y, bins=(np.arange(0, w), np.arange(0, h)))
hist = hist.T
plt.imshow(hist);
```



调用`scipy.ndimage.filters.gaussian_filter`对频度图进行高斯模糊处理，则相当与在上图中每个亮点处描绘一个高斯曲面，让每个亮点增加其周围的像素的亮度。其第二个参数为高斯曲面的宽度，即高斯分布的标准差。这个值越大，曲面的影响范围越大，最终的热点图也越平滑。

```
from scipy.ndimage import filters
heat = filters.gaussian_filter(hist, 10.0)
plt.imshow(heat);
```



下面通过修改热点图的alpha通道，将热点图与地图叠加显示。

7.4 高斯滤波

高斯滤波在图像处理概念下，将图像频域处理和时域处理相联系，作为低通滤波器使用，可以将低频能量（比如噪声）滤去，起到图像平滑作用。

高斯滤波是一种线性平滑滤波，适用于消除高斯噪声，广泛应用于图像处理的减噪过程。通俗的讲，高斯滤波就是对整幅图像进行加权平均的过程，每一个像素点的值，都由其本身和邻域内的其他像素值经过加权平均后得到。高斯滤波的具体操作是：用一个模板（或称卷积、掩模）扫描图像中的每一个像素，用模板确定的邻域内像素的加权平均灰度值去替代模板中心像素点的值。高斯平滑滤波器对于抑制服从正态分布的噪声非常有效。

我们常说的高斯模糊就是使用高斯滤波器完成的，高斯模糊是低通滤波的一种，也就是滤波函数是低通高斯函数，但是高斯滤波是指用高斯函数作为滤波函数，至于是不是模糊，要看是高斯低通还是高斯高通，低通就是模糊，高通就是锐化。

在图像处理中，高斯滤波一般有两种实现方式，一是用离散化窗口滑窗卷积，另一种通过傅里叶变换。最常见的就是第一种滑窗实现，只有当离散化的窗口非常大，用滑窗计算量非常大（即使用可分离滤波器的实现）的情况下，可能会考虑基于傅里叶变化的实现方法。

由于高斯函数可以写成可分离的形式，因此可以采用可分离滤波器实现来加速。所谓的可分离滤波器，就是可以把多维的卷积化成多个一维卷积。具体到二维的高斯滤波，就是指先对行做一维卷积，再对列做一维卷积。这样就可以将计算复杂度从 $O(M*M*N*N)$ 降到 $O(2*M*M*N)$ ， M ， N 分别是图像和滤波器的窗口大小。

高斯模糊是一个非常典型的图像卷积例子，本质上，高斯模糊就是将(灰度)图像和一个高斯核进行卷积操作：

$$I_{\sigma} = I * G_{\sigma}$$

其中 $*$ 表示卷积操作； G_{σ} 是标准差为 σ 的二维高斯核,定义为：

$$G_{\sigma} = \frac{1}{2\pi\sigma} e^{-(x^2+y^2)/2\sigma^2}$$

这里补充以下卷积的知识：

卷积是分析数学中一种重要的运算。

设： $f(x)$ ， $g(x)$ 是 R^1 上的两个可积函数，作积分：

$$\int_{-\infty}^{\infty} f(\tau) g(x-\tau) d\tau$$

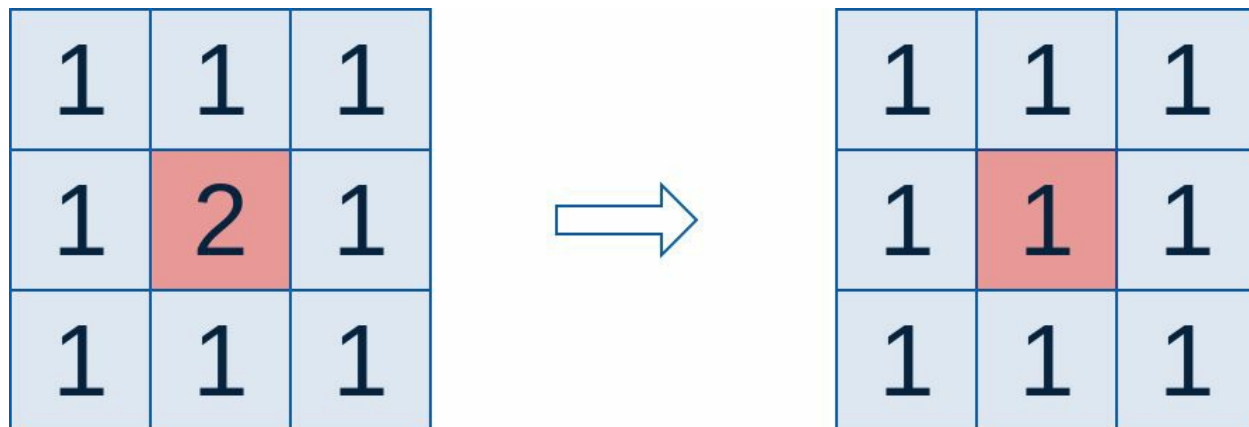
可以证明，关于几乎所有的实数 x ，上述积分是存在的。这样，随着 x 的不同取值，这个积

分就定义了一个新函数 $h(x)$ ，称为函数 f 与 g 的卷积，记为 $h(x)=(f*g)(x)$ 。

卷积是一个单纯的定义，本身没有什么意义可言，但是其在各个领域的应用是十分广泛的，在滤波中可以理解为一个加权平均过程，每一个像素点的值，都由其本身和邻域内的其他像素值经过加权平均后得到,而如何加权则是依据核函数高斯函数。

平均的过程：

对于图像来说，进行平滑和模糊，就是利用周边像素的平均值。



“中间点”取”周围点”的平均值，就会变成1。在数值上，这是一种”平滑化”。在图形上，就相当于产生”模糊”效果，”中间点”失去细节。

显然，计算平均值时，取值范围越大，”模糊效果”越强烈。

使用opencv2进行高斯滤波很方便，参考下面代码：

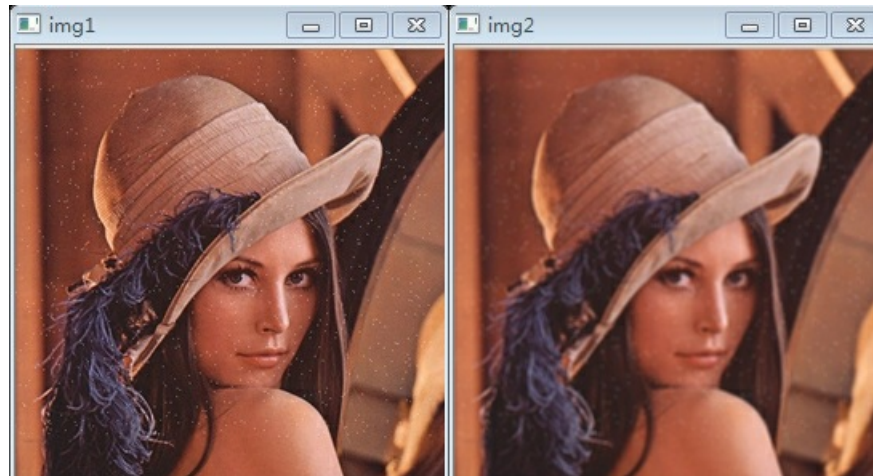
```
import cv2
#两个回调函数
def GaussianBlurSize(GaussianBlur_size):
    global KSIZE
    KSIZE = GaussianBlur_size * 2 +3
    print KSIZE, SIGMA
    dst = cv2.GaussianBlur(src, (KSIZE,KSIZE), SIGMA, KSIZE)
    cv2.imshow(window_name,dst)
```

我们对图片lena.png添加噪音作为输入图片，进行高斯滤波后看看结果如何。这个例子中我们采用的核大小是3，代码如下：

```
from __future__ import print_function
import os
import struct
import numpy as np
import cv2
KSIZE = 3
SIGMA = 3
image = cv2.imread("d:/ai/lena.png")
```

```
print("image shape:",image.shape)
dst = cv2.GaussianBlur(image, (KSIZE,KSIZE), SIGMA, KSIZE)
cv2.imshow("img1",image)
cv2.imshow("img2",dst)
cv2.waitKey()
```

最终生成的结果如下图所示（左边是噪音图像，右边是处理后的结果）：



7.5 图片翻转

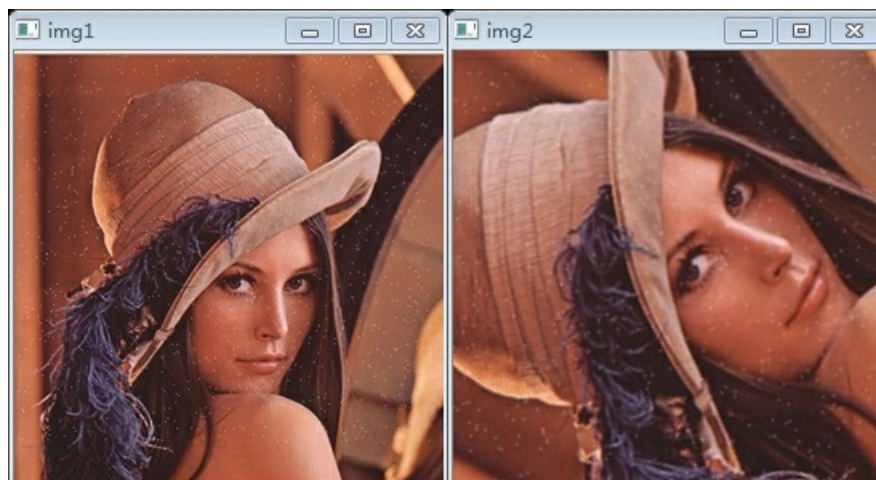
openCv2提供图像的翻转函数getRotationMatrix2D和warpAffine，详细用法可查看相关API文档，这里给出一个简单的例子。

```
from __future__ import print_function
import os
import struct
import math
import numpy as np
import cv2

def rotate(
    img, #image matrix
    angle #angle of rotation
):
    height = img.shape[0]
    width = img.shape[1]
    if angle%180 == 0:
        scale = 1
    elif angle%90 == 0:
        scale = float(max(height, width))/min(height, width)
    else:
        scale = math.sqrt(pow(height,2)+pow(width,2))/min(height, width)
    #print 'scale %f\n' %scale
    rotateMat = cv2.getRotationMatrix2D((width/2, height/2), angle, scale)
    rotateImg = cv2.warpAffine(img, rotateMat, (width, height))
    #cv2.imshow('rotateImg',rotateImg)
    #cv2.waitKey(0)
    return rotateImg #rotated image

image = cv2.imread("d:/ai/lena.png")
dst = rotate(image,60)
cv2.imshow("img1",image)
cv2.imshow("img2",dst)
cv2.waitKey()
```

运行之后得出结果如下图所示：



7.6 轮廓检测

轮廓检测也是图像处理中经常用到的。OpenCV2使用findContours()函数来查找检测物体的轮廓。

```
import cv2
img = cv2.imread('D:\\test\\contour.jpg')
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
ret, binary = cv2.threshold(gray,127,255,cv2.THRESH_BINARY)

contours, hierarchy = cv2.findContours(binary,cv2.RETR_TREE,cv2.CHAIN_APPROX_SIMPLE)
cv2.drawContours(img,contours,-1,(0,0,255),3)

cv2.imshow("img", img)
cv2.waitKey(0)
```

需要注意的是cv2.findContours()函数接受的参数为二值图，即黑白的（不是灰度图），所以读取的图像要先转成灰度的，再转成二值图。

原图如下：



检测结果如下：



注意，findcontours函数会“原地”修改输入的图像。这一点可通过下面的语句验证：

```
cv2.imshow("binary", binary)
contours, hierarchy = cv2.findContours(binary,cv2.RETR_TREE,cv2.CHAIN_APPROX_SIMPLE)
cv2.imshow("binary2", binary)
```

执行这些语句后会发现原图被修改了。

cv2.findContours()函数

函数的原型为

```
cv2.findContours(image, mode, method[, contours[, hierarchy[, offset ]]])
```

返回两个值：contours： hierarchy。

参数

第一个参数是寻找轮廓的图像；

第二个参数表示轮廓的检索模式，有四种（本文介绍的都是新的cv2接口）：

cv2.RETR_EXTERNAL表示只检测外轮廓。

cv2.RETR_LIST检测的轮廓不建立等级关系。

cv2.RETR_CCOMP建立两个等级的轮廓，上面的一层为外边界，里面的一层为内孔的边界信息。如果内孔内还有一个连通物体，这个物体的边界也在顶层。

cv2.RETR_TREE建立一个等级树结构的轮廓。

第三个参数method为轮廓的近似办法。

cv2.CHAIN_APPROX_NONE存储所有的轮廓点，相邻的两个点的像素位置差不超过1，即 $\max(\text{abs}(x_1-x_2), \text{abs}(y_2-y_1)) \leq 1$ 。

cv2.CHAIN_APPROX_SIMPLE压缩水平方向，垂直方向，对角线方向的元素，只保留该方向的终点坐标，例如一个矩形轮廓只需4个点来保存轮廓信息。

cv2.CHAIN_APPROX_TC89_L1，CV_CHAIN_APPROX_TC89_KCOS使用teh-Chinl chain近似算法。

返回值

cv2.findContours()函数返回两个值，一个是轮廓本身，还有一个是每条轮廓对应的属性。

contour返回值

cv2.findContours()函数首先返回一个list，list中每个元素都是图像中的一个轮廓，用numpy中的ndarray表示。这个概念非常重要。在下面drawContours中会看见。

```
print(type(contours))
print(type(contours[0]))
print(len(contours))
```

可以验证上述信息。会看到本例中有两条轮廓，一个是五角星的，一个是矩形的。每个轮廓是一个ndarray，每个ndarray是轮廓上的点的集合。

由于我们知道返回的轮廓有两个，因此可通过

```
cv2.drawContours(img, contours, 0, (0, 0, 255), 3)
```

和

```
cv2.drawContours(img, contours, 1, (0, 255, 0), 3)
```

分别绘制两个轮廓，关于该参数可参见下面一节的内容。同时通过

```
print(len(contours[0]))  
print(len(contours[1]))
```

输出两个轮廓中存储的点的个数，可以看到，第一个轮廓中只有4个元素，这是因为轮廓中并不是存储轮廓上所有的点，而是只存储可以用直线描述轮廓的点的个数，比如一个“正立”的矩形，只需4个顶点就能描述轮廓了。

7.7 角点

角点的定义和特性：

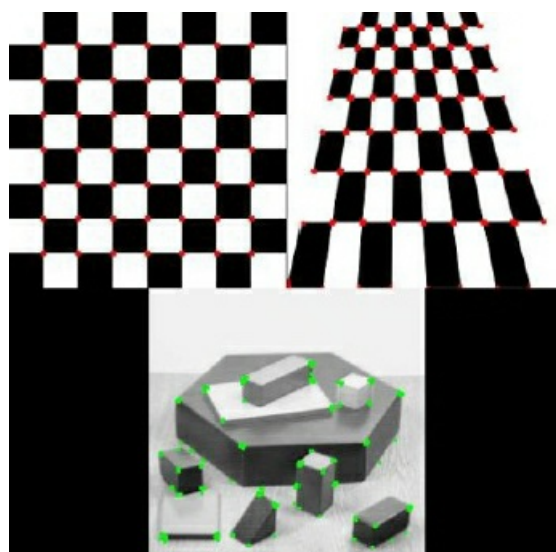
- 角点是一类含有足够信息且能从当前帧和下一帧中都能提取出来的点。
- 最普遍使用的角点的定义是由Harris提出的。
- 典型的角点检测算法：Harris角点检测、CSS角点检测。
- 好的角点检测算法的特点：1、检测出图像中“真实的”角点；2、准确的定位性能；3、很高的重复检测率（稳定性好）；4、具有对噪声的鲁棒性；5、具有较高的计算效率。

Open 中的函数`cv2.cornerHarris(src, blockSize, ksize, k[, dst[, borderType]])` → `dst` 可以用来进行角点检测。参数如下：

- `src` –数据类型为float32 的输入图像。
- `dst` – 存储角点数组的输出图像，和输入图像大小相等。
- `blockSize` –角点检测中要考虑的领域大小。
- `ksize` –Sobel 求导中使用的窗口大小。
- `k` –Harris 角点检测方程中的自由参数，取值参数为[0.04, 0.06]。
- `borderType` –边界类型。

```
# coding=utf-8
import cv2
import numpy as np
'''Harris算法角点特征提取'''
img = cv2.imread('chess_board.png')
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
gray = np.float32(gray)
# {标记点大小, 敏感度 (3~31, 越小越敏感)}
# OpenCV函数cv2.cornerHarris() 有四个参数 其作用分别为 :
dst = cv2.cornerHarris(gray,2,23,0.04)
img[dst>0.01 * dst.max()] = [0,0,255]
cv2.imshow('corners',img)
cv2.waitKey()
cv2.destroyAllWindows()
```

最后检测出角点并在图像上标注出来，示例如下：



7.8 直方图

图像的构成是有像素点构成的，每个像素点的值代表着该点的颜色（灰度图或者彩色图）。所谓直方图就是对图像中的这些像素点的值进行统计，得到一个统一的整体灰度概念。直方图的好处就在于可以清晰了解图像的整体灰度分布，这对于后面依据直方图处理图像来说至关重要。

一般情况下直方图都是灰度图像，直方图x轴是灰度值（一般0~255），y轴就是图像中每一个灰度级对应的像素点的个数。

那么如何获得图像的直方图？首先来了解绘制直方图需要的一些量：灰度级，正常情况下就是0-255共256个灰度级，从最黑一直到最亮（白）（也有可能统计其中的某部分灰度范围），那么每一个灰度级对应一个数来储存该灰度对应的点数目。也就是说直方图其实就是一个 $1 \times m$ （灰度级）的一个数组而已。但是有的时候我们不希望一个一个灰度的递增，比如现在我想15个灰度一起作为一个灰度级来画直方图，这个时候我们可能只需要 $1 \times (m/15)$ 这样一个数组就够了。那么这里的15就是直方图的间隔宽度了。

OpenCV给我们提供的函数是`cv2.calcHist()`，该函数有5个参数：

- **image**: 输入图像，传入时应该用中括号[]括起来。
- **channels**: 传入图像的通道，如果是灰度图像，那就不用说了，只有一个通道，值为0，如果是彩色图像（有3个通道），那么值为0,1,2,中选择一个，对应着BGR各个通道。这个值也得用[]传入。
- **mask**: 掩膜图像。如果统计整幅图，那么为none。主要是如果要统计部分图的直方图，就得构造相应的掩膜来计算。
- **histSize**: 灰度级的个数，需要中括号，比如[256]。
- **ranges**: 像素值的范围，通常[0,256]，有的图像如果不是0-256，比如说你来回各种变换导致像素值负值、很大，则需要调整后才可以。

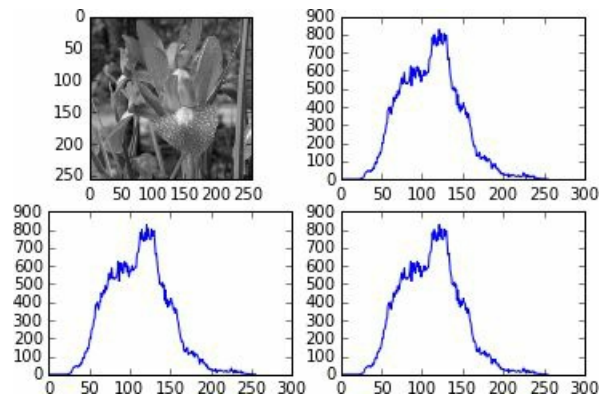
除此之外，强大的numpy也有函数用于统计直方图的，通用的一个函数`np.histogram`，还有一个函数是`np.bincount()`（用于统计直方图，速度更快）。这三个方式的传入参数基本上差不多，不同的是opencv自带的需要中括号括起来。

对于直方图的显示也是比较简单的，直接`plt.plot()`就可以。一个实例如下：

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
img = cv2.imread('flower.jpg',0) #直接读为灰度图像
#opencv方法读取-cv2.calcHist（速度最快）
#图像，通道[0]-灰度图，掩膜-无，灰度级，像素范围
hist_cv = cv2.calcHist([img],[0],None,[256],[0,256])
#numpy方法读取-np.histogram()
hist_np,bins = np.histogram(img.ravel(),256,[0,256])
```



```
#numpy的另一种方法读取-np.bincount()（速度=10倍法2）
hist_np2 = np.bincount(img.ravel(),minlength=256)
plt.subplot(221),plt.imshow(img,'gray')
plt.subplot(222),plt.plot(hist_cv)
plt.subplot(223),plt.plot(hist_np)
plt.subplot(224),plt.plot(hist_np2)
```

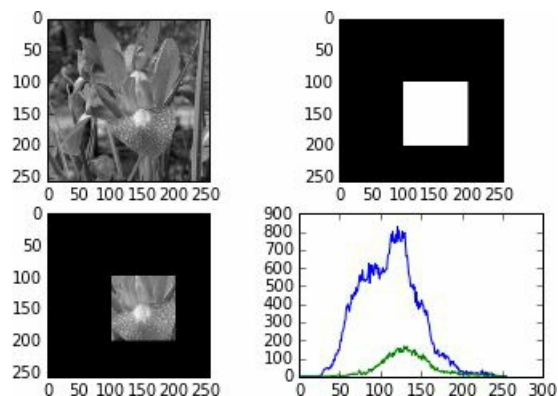


现在来考虑opencv的直方图函数中掩膜的使用，这个掩膜就是一个区域大小，表示你接下来的直方图统计就是这个区域的像素统计。一个例子如下：

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
img = cv2.imread('flower.jpg',0) #直接读为灰度图像
mask = np.zeros(img.shape[:2],np.uint8)
mask[100:200,100:200] = 255
masked_img = cv2.bitwise_and(img,img,mask=mask)

#opencv方法读取-cv2.calcHist（速度最快）
#图像，通道[0]-灰度图，掩膜-无，灰度级，像素范围
hist_full = cv2.calcHist([img],[0],None,[256],[0,256])
hist_mask = cv2.calcHist([img],[0],mask,[256],[0,256])

plt.subplot(221),plt.imshow(img,'gray')
plt.subplot(222),plt.imshow(mask,'gray')
plt.subplot(223),plt.imshow(masked_img,'gray')
plt.subplot(224),plt.plot(hist_full),plt.plot(hist_mask)
```



7.9 形态学图像处理

本节介绍如何使用morphology模块实现二值图像处理。二值图像中的每个像素的颜色只有两种：黑色和白色，在NumPy中可以用二维布尔数组表示：False表示黑色，True表示白色。也可以用无符号单字节整型(uint8)数组表示：0表示黑色，非0表示白色。

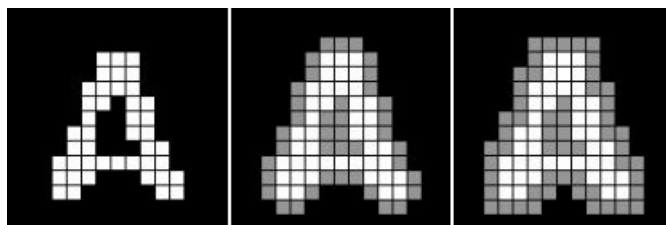
下面的两个函数用于显示形态学图像处理的结果。

```
import numpy as np
def expand_image(img, value, out=None, size = 10):
    if out is None:
        w, h = img.shape
        out = np.zeros((w*size, h*size), dtype=np.uint8)
        tmp = np.repeat(np.repeat(img, size, 0), size, 1)
        out[:, :] = np.where(tmp, value, out)
        out[:, :size, :] = 0
        out[:, :, :size] = 0
    return out
def show_image(*imgs):
    for idx, img in enumerate(imgs, 1):
        ax = plt.subplot(1, len(imgs), idx)
        plt.imshow(img, cmap="gray")
        ax.set_axis_off()
    plt.subplots_adjust(0.02, 0, 0.98, 1, 0.02, 0)
```

7.9.1 膨胀和腐蚀

二值图像最基本的形态学运算是膨胀和腐蚀。膨胀运算是将与某物体(白色区域)接触的所有背景像素(黑色区域)合并到该物体中的过程。简单地说，就是对于原始图像中的每个白色像素进行处理，将其周围的黑色像素都设置为白色像素。这里的“周围”是一个模糊概念，在实际运算时，需要明确给出“周围”的定义。下图是膨胀运算的一个例子，其中左图是原始图像，中间的图是四连通定义的“周围”的膨胀效果，右图是八连通定义的“周围”的膨胀效果。图中用灰色方块表示由膨胀处理添加进物体的像素。

```
from scipy.ndimage import morphology
def dilation_demo(a, structure=None):
    b = morphology.binary_dilation(a, structure)
    img = expand_image(a, 255)
    return expand_image(np.logical_xor(a, b), 150, out=img)
a = plt.imread("images/scipy_morphology_demo.png")[:, :, 0].astype(np.uint8)
img1 = expand_image(a, 255)
img2 = dilation_demo(a)
img3 = dilation_demo(a, [[1, 1, 1], [1, 1, 1], [1, 1, 1]])
show_image(img1, img2, img3)
```



四连通包括上下左右四个像素，而八连通则还包括四个斜线方向上的邻接像素。它们都可以使用下面的正方形矩阵定义，其中正中心的元素表示当前要进行运算的像素，而其周围的1和0表示对应位置的像素是否算作其“周围”像素。这种矩阵描述了周围像素和当前像素之间的关系，被称作结构元素(structuring element)。

四连通	八连通
0 1 0	1 1 1
1 1 1	1 1 1
0 1 0	1 1 1

假设数组a是一个表示二值图像的数组，可以用如下语句对其进行膨胀运算：

```
binary_dilation(a)
```

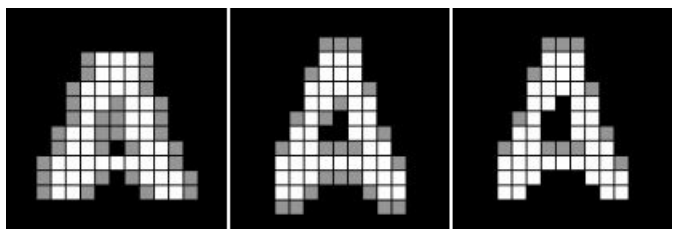
`binary_dilation()`缺省使用四连通进行膨胀运算，通过`structure`参数可以指定其它的结构元素，下面是进行八连通膨胀运算的语句：

```
binary_dilation(a, structure=\[\[1,1,1],\[1,1,1],\[1,1,1]]])
```

通过设置不同的结构元素，能够制作出各种不同的效果，下面显示了三个种不同结构元素的膨胀效果。图中的结构元素分别为：

左	中	右
0 0 0	0 1 0	0 1 0
1 1 1	0 1 0	0 1 0
0 0 0	0 1 0	0 0 0

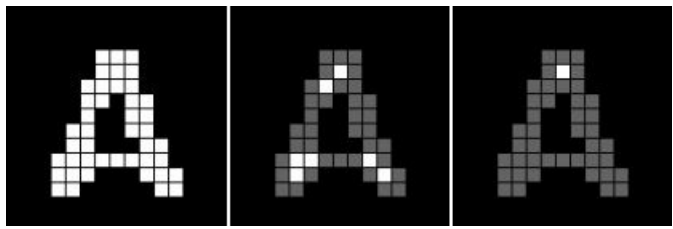
```
img4 = dilation_demo(a, [[0,0,0],[1,1,1],[0,0,0]])
img5 = dilation_demo(a, [[0,1,0],[0,1,0],[0,1,0]])
img6 = dilation_demo(a, [[0,1,0],[0,1,0],[0,0,0]])
show_image(img4, img5, img6)
```



`binary_erosion()`的腐蚀运算正好和膨胀相反，它将“周围”有黑色像素的白色像素设置为黑色。下面是四连通和八连通腐蚀的效果，图中用灰色方块表示被腐蚀的像素。

```
def erosion_demo(a, structure=None):
    b = morphology.binary_erosion(a, structure)
    img = expand_image(a, 255)
    return expand_image(np.logical_xor(a,b), 100, out=img)

img1 = expand_image(a, 255)
img2 = erosion_demo(a)
img3 = erosion_demo(a, [[1,1,1],[1,1,1],[1,1,1]])
show_image(img1, img2, img3)
```



7.9.2 Hit和Miss

Hit和Miss是二值形态学图像处理中最基本的运算，因为几乎所有的其它的运算都可以用Hit和Miss的组合推演出来。它对图像中的每个像素周围的像素进行模式判断，如果周围像素的黑白模式符合指定的模式，则将此像素设为白色，否则设置为黑色。因为它需要同时对白色和黑色像素进行判断，因此需要指定两个结构元素。进行Hit和Miss运算的`binary_hit_or_miss()`的调用形式如下：

```
binary_hit_or_miss(input, structure1=None, structure2=None, ...)
```

其中`structure1`参数指定白色像素的结构元素，而`structure2`参数则指定黑色像素的结构元素。下图是`binary_hit_or_miss()`的运算结果。其中左图为原始图像，中图为使用下面两个结构元素进行运算的结果：

白色结构元素	黑色结构元素
0 0 0	1 0 0
0 1 0	0 0 0
1 1 1	0 0 0

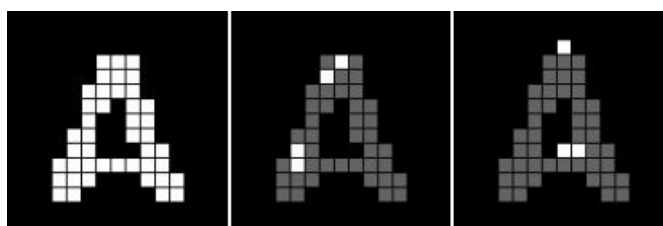
在这两个结构元素中，0表示不关心其对应位置的像素的颜色，1表示其对应位置的像素必须为结构元素所表示的颜色。因此通过这两个结构元素可以找到“下方三个像素为白色，并且左上像素为黑色的白色像素”。

与右图对应的结构元素如下。通过它可以找到“下方三个像素为白色、左上像素为黑色的黑色像素”。

白色结构元素	黑色结构元素
0 0 0	1 0 0
0 0 0	0 1 0
1 1 1	0 0 0

```
def hitmiss_demo(a, structure1, structure2):
    b = morphology.binary_hit_or_miss(a, structure1, structure2)
    img = expand_image(a, 100)
    return expand_image(b, 255, out=img)
img1 = expand_image(a, 255)
img2 = hitmiss_demo(a, [[0,0,0],[0,1,0],[1,1,1]], [[1,0,0],[0,0,0],[0,0,0]])
img3 = hitmiss_demo(a, [[0,0,0],[0,0,0],[1,1,1]], [[1,0,0],[0,1,0],[0,0,0]])

show_image(img1, img2, img3)
```



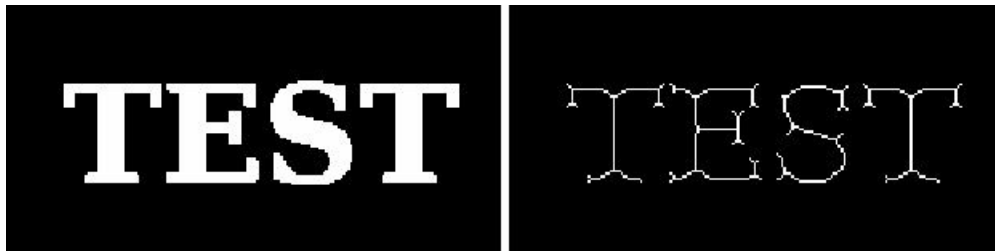
使用Hit和Miss运算的组合，可以实现很复杂的图像处理。例如文字识别中常用的细线化运算就可以用一系列的Hit和Miss运算实现。下图显示了细线化处理的效果。

```
def skeletonize(img):
    h1 = np.array([[0, 0, 0],[0, 1, 0],[1, 1, 1]])
    m1 = np.array([[1, 1, 1],[0, 0, 0],[0, 0, 0]])
    h2 = np.array([[0, 0, 0],[1, 1, 0],[0, 1, 0]])
    m2 = np.array([[0, 1, 1],[0, 0, 1],[0, 0, 0]])
    hit_list = []
    miss_list = []
    for k in range(4):
        hit_list.append(np.rot90(h1, k))
        hit_list.append(np.rot90(h2, k))
        miss_list.append(np.rot90(m1, k))
        miss_list.append(np.rot90(m2, k))
    img = img.copy()
    while True:
        last = img
        for hit, miss in zip(hit_list, miss_list):
            hm = morphology.binary_hit_or_miss(img, hit, miss)
```

```

        # 从图像中删除hit_or_miss所得到的白色点
        img = np.logical_and(img, np.logical_not(hm))
    # 如果处理之后的图像和处理前的图像相同，则结束处理
    if np.all(img == last):
        break
    return img
a = plt.imread("images/scipy_morphology_demo2.png")[:, :, 0].astype(np.uint8)
b = skeletonize(a)
_, (ax1, ax2) = plt.subplots(1, 2, figsize=(9, 3))
ax1.imshow(a, cmap="gray", interpolation="nearest")
ax2.imshow(b, cmap="gray", interpolation="nearest")
ax1.set_axis_off()
ax2.set_axis_off()
plt.subplots_adjust(0.02, 0, 0.98, 1, 0.02, 0)

```



细线化算法的实现程序如下，这里只列出其中真正进行细线化算法的函数skeletonize()：

根据上图所示的两个结构元素为基础，构造四个3*3的二维数组：h1、m1、h2、m2。其中h1和m1对应图中左边的结构元素，而h2和m2对应图中右边的结构元素，h1和h2对应白色结构元素，m1和m2对应黑色结构元素。将这些结构元素进行90、180、270度旋转之后一共得到8个结构元素。

依次使用这些结构元素进行Hit和Miss运算，并从图像中删除运算所得到的白色像素，其效果就是依次从8个方向删除图像的边缘上的像素。重复运算直到没有像素可删除为止。

8 CNN实战

本章重点介绍使用Keras的神经网络做图像识别的具体例子。通过这几个例子来掌握Keras神经网络的编程方法。

8.1 Mnist

MNIST是一个入门级的计算机视觉数据集，它包含各种手写数字图片：



它也包含每一张图片对应的标签，告诉我们这个是数字几。比如，上面这四张图片的标签分别是5，0，4，1。

用mnist数据集作例子，下载地址：<http://yann.lecun.com/exdb/mnist/>。一共四个文件：

train-images-idx3-ubyte.gz: training set images (9912422 bytes)

train-labels-idx1-ubyte.gz: training set labels (28881 bytes)

t10k-images-idx3-ubyte.gz: test set images (1648877 bytes)

t10k-labels-idx1-ubyte.gz: test set labels (4542 bytes)

这些文件中是已经处理过的数组数据，通过numpy的相关方法读取到训练数据集和测试数据集数组中。

8.1.1 代码

```
'''Trains a simple convnet on the MNIST dataset.
Gets to 99.25% test accuracy after 12 epochs
(there is still a lot of margin for parameter tuning).
16 seconds per epoch on a GRID K520 GPU.
'''
from __future__ import print_function
import os
import struct
import numpy as np
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras import backend as K

def load_mnist(path, kind='train'):
    """Load MNIST data from `path`"""
    labels_path = os.path.join(path, '%s-labels.idx1-ubyte' % kind)
    images_path = os.path.join(path, '%s-images.idx3-ubyte' % kind)
    with open(labels_path, 'rb') as lbpath:
```



```

        magic, n = struct.unpack('>II', lbp.read(8))
        labels = np.fromfile(lbp, dtype=np.uint8)
    with open(images_path, 'rb') as imgpath:
        magic, num, rows, cols = struct.unpack(">IIII", imgpath.read(16))
        images = np.fromfile(imgpath, dtype=np.uint8).reshape(len(labels), 784)
    return images, labels

X_train, y_train = load_mnist('./data', kind='train')
print('Rows: %d, columns: %d' % (X_train.shape[0], X_train.shape[1]))
X_test, y_test = load_mnist('./data', kind='t10k')
print('Rows: %d, columns: %d' % (X_test.shape[0], X_test.shape[1]))

batch_size = 128
num_classes = 10
epochs = 12

# input image dimensions
img_rows, img_cols = 28, 28

# the data, shuffled and split between train and test sets
x_train= X_train #mnist.load_data()
x_test=X_test

if K.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0], 1, img_rows, img_cols)
    x_test = x_test.reshape(x_test.shape[0], 1, img_rows, img_cols)
    input_shape = (1, img_rows, img_cols)
else:
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
    input_shape = (img_rows, img_cols, 1)

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(),
              metrics=['accuracy'])

```

```
model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          validation_data=(x_test, y_test))
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

该例子的输入是28×28像素的手写数字图片，这里使用其灰度数据进行卷积，因此输入张量为（1，28，28）。

第一个卷积层已（3，3）卷积核对原始数据进行卷积，该层特征数取32，意味着图像的灰度通道映射到长度32的权重向量。该层输出向量（32，26，26），如果通道在后方式则为（26，26，32）。

后面的层配置这里就不展开论述了，最后我们看看执行结果如果。由于笔者是在笔记本上跑，速度慢，将epoch改成3，最后结果如下：

```
58752/60000 [=====>.] - ETA: 12s - loss: 0.0828 - acc: 0.
58880/60000 [=====>.] - ETA: 11s - loss: 0.0829 - acc: 0.
59008/60000 [=====>.] - ETA: 10s - loss: 0.0829 - acc: 0.
59136/60000 [=====>.] - ETA: 8s - loss: 0.0828 - acc: 0.9
59264/60000 [=====>.] - ETA: 7s - loss: 0.0827 - acc: 0.9
59392/60000 [=====>.] - ETA: 6s - loss: 0.0827 - acc: 0.9
59520/60000 [=====>.] - ETA: 4s - loss: 0.0827 - acc: 0.9
59648/60000 [=====>.] - ETA: 3s - loss: 0.0827 - acc: 0.9
59776/60000 [=====>.] - ETA: 2s - loss: 0.0826 - acc: 0.9
59904/60000 [=====>.] - ETA: 0s - loss: 0.0825 - acc: 0.9
60000/60000 [=====] - 630s - loss: 0.0826 - acc: 0.9755
- val_loss: 0.0429 - val_acc: 0.9862
Test loss: 0.0429289447919
Test accuracy: 0.9862
```

8.1.2 导入导出

每次训练模型都需要花费大量的时间，为此Keras提供了模型导入导出功能，这样下次用同一个模型的时候直接从保存的模型导入即可，不需要每次都花费大量时间重新训练。

- 模型的保存

```
Model.save('mnist.model')
```

将模型保存到当前目录mnist.model文件中，文件中包括模型定义和各层的权重参数。

- 模型的导入

```
from keras.models import load_model
model = load_model('mnist.model')
```

从mnist.model文件导入模型。导入的模型可以直接用于预测。

8.1.3 预测

```
model.summary()
```

计算模型各层的参数数量和张量大小。

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 26, 26, 32)	320
conv2d_2 (Conv2D)	(None, 24, 24, 64)	18496
max_pooling2d_1 (MaxPooling2D)	(None, 12, 12, 64)	0
dropout_1 (Dropout)	(None, 12, 12, 64)	0
flatten_1 (Flatten)	(None, 9216)	0
dense_1 (Dense)	(None, 128)	1179776
dropout_2 (Dropout)	(None, 128)	0
dense_2 (Dense)	(None, 10)	1290
Total params: 1,199,882		
Trainable params: 1,199,882		
Non-trainable params: 0		

```
result = model.predict_classes(x_train\[10001:10003,:,:,])  
print(result)
```

result是分类的结果，在mnist场景中就是0-9的数字，如果是用predict方法预测，则返回的result是各个分类结果的可能性值，比如向量模式，比如

```
\[\[ 3.30833864e-05  6.27903442e-04  1.01537415e-04  9.22122912e-04  
      8.59064767e-06  2.64745613e-05  2.08549463e-05  4.13584530e-07  
      9.98244345e-01  1.47016517e-05]  
  
\[ 4.91593006e-08  2.06154118e-06  4.62675816e-05  8.84949695e-05  
     6.84448764e-09  1.48443196e-08  6.24425234e-10  9.99852717e-01  
     5.75028162e-06  4.63267634e-06]]
```

如果给定图片文件a.jpg，则我们读取图像通道1的数据进行预测。

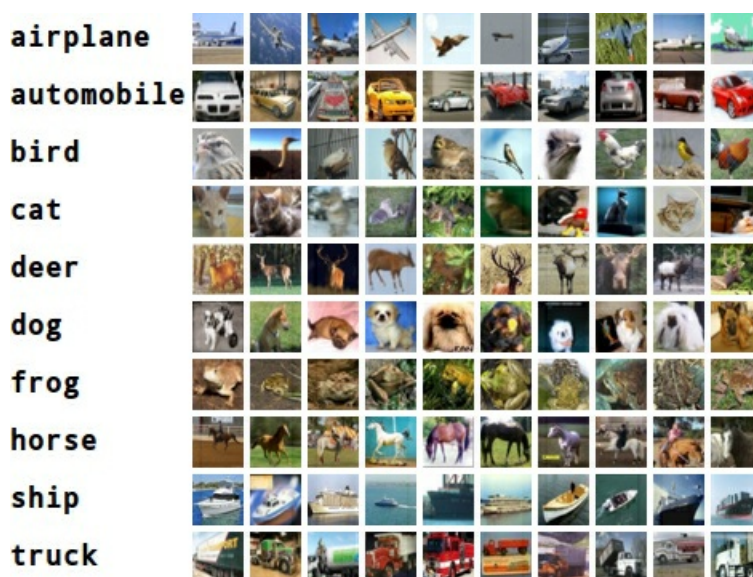
```
Import cv2  
image = cv2.imread("a.jpg",0)  
#image.shape=[28,28,1]  
data = np.empty((1,28,28,1),dtype="float32")  
result = model.predict(data)
```

`result`返回的就是预测结果。

8.2 Cifar16

我们以cifar10_cnn.py作为神经网络图像识别的另一个例子。

Cifar-10是由Hinton的两个大弟子Alex Krizhevsky、Ilya Sutskever收集的一个用于普适物体识别的数据集。Cifar-10由60000张32*32的RGB彩色图片构成，共10个分类。50000张训练，10000张测试（交叉验证）。这个数据集最大的特点在于将识别迁移到了普适物体，而且应用于多分类（姐妹数据集Cifar-100达到100类，ILSVRC比赛则是1000类）。



运行python cifar10_cnn.py，首先下载测试图像集数据，保存到目前用户目录的.keras目录下，在windows系统就在c:\用户\xx.keras\dataset下：

此电脑 > 本地磁盘 (C:) > 用户 > houyijun > .keras > datasets				
名称	修改日期	类型	大小	
cifar-10-batches-py.tar.gz	2017/5/21 0:06	WinRAR 压缩文件	2,336 KB	
imdb.pkl	2016/12/8 20:37	PKL 文件	16 KB	
imdb_full.pkl	2016/12/8 23:35	PKL 文件	32 KB	

其中cifar-10-batches-py.tar.gz就是下载的图像集压缩文件，一共50000张训练图片，10000张测试图片。

然后等待程序训练数据集。

```

F:\Keras\keras-master\examples>python cifar10_cnn.py
Using Theano backend.
x_train shape: (50000L, 32L, 32L, 3L)
50000 train samples
10000 test samples
Using real-time data augmentation.
Epoch 1/200
1562/1562 [=====] - 510s - loss: 1.8772 - acc: 0.3098 - val_loss: 1.5649 - val_acc: 0.4336
Epoch 2/200
174/1562 [=>.....] - ETA: 427s - loss: 1.6682 - acc: 0.3894
搜狗拼音输入法 全：

```

根据机器的性能要等待几个小时才会出结果。

8.2.1 代码

Keras的源码如下所示。

```

'''Train a simple deep CNN on the CIFAR10 small images dataset.

GPU run command with Theano backend (with TensorFlow, the GPU is automatically used):
    THEANO_FLAGS=mode=FAST_RUN,device=gpu,floatx=float32 python cifar10_cnn.py

It gets down to 0.65 test logloss in 25 epochs, and down to 0.55 after 50 epochs.
(it's still underfitting at that point, though).
'''

from __future__ import print_function
import keras
from keras.datasets import cifar10
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Conv2D, MaxPooling2D

batch_size = 32
num_classes = 10
epochs = 200
data_augmentation = True

# The data, shuffled and split between train and test sets:
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# Convert class vectors to binary class matrices.
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

model = Sequential()
model.add(Conv2D(32, (3, 3), padding='same',
                 input_shape=x_train.shape[1:]))
model.add(Activation('relu'))
model.add(Conv2D(32, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Conv2D(64, (3, 3), padding='same'))

```

```

model.add(Activation('relu'))
model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes))
model.add(Activation('softmax'))

# initiate RMSprop optimizer
opt = keras.optimizers.rmsprop(lr=0.0001, decay=1e-6)

# Let's train the model using RMSprop
model.compile(loss='categorical_crossentropy',
              optimizer=opt,
              metrics=['accuracy'])

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
if not data_augmentation:
    print('Not using data augmentation.')
    model.fit(x_train, y_train,
              batch_size=batch_size,
              epochs=epochs,
              validation_data=(x_test, y_test),
              shuffle=True)
else:
    print('Using real-time data augmentation.')
    # This will do preprocessing and realtime data augmentation:
    datagen = ImageDataGenerator(
        featurewise_center=False, # set input mean to 0 over the dataset
        samplewise_center=False, # set each sample mean to 0
        featurewise_std_normalization=False, # divide inputs by std of the dataset
        samplewise_std_normalization=False, # divide each input by its std
        zca_whitening=False, # apply ZCA whitening
        rotation_range=0, # randomly rotate images in the range (degrees, 0 to 180)
        width_shift_range=0.1, # randomly shift images horizontally (fraction of total width)
        height_shift_range=0.1, # randomly shift images vertically (fraction of total height)
        horizontal_flip=True, # randomly flip images
        vertical_flip=False) # randomly flip images
    # Compute quantities required for feature-wise normalization
    # (std, mean, and principal components if ZCA whitening is applied).
    datagen.fit(x_train)
    # Fit the model on the batches generated by datagen.flow().
    model.fit_generator(datagen.flow(x_train, y_train, batch_size=batch_size),
                        steps_per_epoch=x_train.shape[0] // batch_size,
                        epochs=epochs,
                        validation_data=(x_test, y_test))

```

8.2.2 分析

一般流程：

- 准备数据集。
- 创建神经网络模型，一般是Sequenal模型。
- 编译模型，调用model.compile方法。
- 训练模型，调用model.fit方法。
- 评估模型，调用model.evaluate方法。

```
#编译模型
model.compile(loss='categorical_crossentropy',
optimizer='adadelta',
metrics=['accuracy'])
#训练模型
model.fit(X_train, Y_train, batch_size=batch_size, epochs=epochs,
          verbose=1, validation_data=(X_test, Y_test))
#评估模型
score = model.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])
```


8.3 人脸识别

现在，我们开始尝试训练我们自己的卷积神经网络模型。CNN擅长图像处理，我们需要通过大量的训练数据训练我们的模型，因此首先要做的就是将训练数据准备好，并将其输入给CNN。我们先准备好2000张脸部图像，并对数据进行标注，将这2000张图片保存在目录中，以文件名作为类别标注。

首先我们建立一个空白的python文件，文件名为：load_face_dataset.py，示例代码如下：

```
# -*- coding: utf-8 -*-
import os
import sys
import numpy as np
import cv2
IMAGE_SIZE = 64
#按照指定图像大小调整尺寸
def resize_image(image, height = IMAGE_SIZE, width = IMAGE_SIZE):
    top, bottom, left, right = (0, 0, 0, 0)

    h, w, _ = image.shape
    #对于长宽不相等的图片，找到最长的一边
    longest_edge = max(h, w)
    #计算短边需要增加多少像素宽度使其与长边等长
    if h < longest_edge:
        dh = longest_edge - h
        top = dh // 2
        bottom = dh - top
    elif w < longest_edge:
        dw = longest_edge - w
        left = dw // 2
        right = dw - left
    else:
        pass
    #RGB颜色
    BLACK = [0, 0, 0]
    #给图像增加边界，是图片长、宽等长，cv2.BORDER_CONSTANT指定边界颜色由value指定
    constant = cv2.copyMakeBorder(image, top, bottom, left, right, cv2.BORDER_CONSTANT, value=BLACK)
    #调整图像大小并返回
    return cv2.resize(constant, (height, width))

#读取训练数据
images = []
labels = []
def read_path(path_name):
    for dir_item in os.listdir(path_name):
        #从初始路径开始叠加，合并成可识别的操作路径
        full_path = os.path.abspath(os.path.join(path_name, dir_item))

        if os.path.isdir(full_path):
            read_path(full_path)
        else:
            if dir_item.endswith('.jpg'):
                image = cv2.imread(full_path)
                image = resize_image(image, IMAGE_SIZE, IMAGE_SIZE)
                #cv2.imwrite('1.jpg', image)
```

```

        images.append(image)
        labels.append(path_name)
    return images, labels
#从指定路径读取训练数据
def load_dataset(path_name):
    images, labels = read_path(path_name)
    #将输入的所有图片转成四维数组，尺寸为(图片数量*IMAGE_SIZE*IMAGE_SIZE*3)
    #两个人共1200张图片，IMAGE_SIZE为64，故对我来说尺寸为1200 * 64 * 64 * 3
    #图片为64 * 64像素，一个像素3个颜色值(RGB)
    images = np.array(images)
    print(images.shape)
    #标注数据，'me'文件夹下都是我的脸部图像，全部指定为0，另外一个文件夹下是女儿的，全部指定为1
    labels = np.array([0 if label.endswith('me') else 1 for label in labels])
    return images, labels
if __name__ == '__main__':
    if len(sys.argv) != 2:
        print("Usage:%s path_name\r\n" % (sys.argv[0]))
    else:
        images, labels = load_dataset(sys.argv[1])

```

上面给出的代码主函数就是load_dataset()，它将图片数据进行标注并以多维数组的形式加载到内存中。实际用于训练的脸部数据取**1200**张，同时去掉一些模糊的或者表情基本一致的头像，留下了清晰、脸部表情有些区别的图像。上述代码注释很清楚，不多讲，唯一一个理解起来稍微有点难度的就是resize_image()函数。这个函数其实就做了一件事情，判断图片是不是四边等长，也就是图片是不是正方形。如果不是，则短的那两边增加两条黑色的边框，使图像变成正方形，这样再调用cv2.resize()函数就可以实现等比例缩放了。因为我们指定缩放的比例就是64 x 64，只有缩放之前图像为正方形才能确保图像不失真。resize_image()函数的执行结果如下所示：

上图为200 x 300的图片，宽度小于高度，因此，需要增加宽度，正常应该是两边各增加宽50像素的黑边。变成一个300 x 300的正方形图片，这时我们再缩放到64 x 64就可以了。

这些工作做完之后，我们就可以开始构建训练代码了。

同样，在load_face_dataset.py所在文件夹下新建一个python空白文件face_train_use_keras.py，然后我们先把需要的库文件添加到代码中：

```

#-*- coding: utf-8 -*-
import random
import numpy as np
from sklearn.cross_validation import train_test_split
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Convolution2D, MaxPooling2D
from keras.optimizers import SGD
from keras.utils import np_utils
from keras.models import load_model
from keras import backend as K

from load_face_dataset import load_dataset, resize_image, IMAGE_SIZE

```

我们先不管导入的这些库是干啥的，你只要知道接下来的代码要用到这些库就够了，用到了我们再讲。到目前为止，数据加载的工作已经完成，我们只需调用这个接口即可。关于训练集的使用，我们需要拿出一部分用于训练网络，建立识别模型；另一部分用于验证模型。同时我们还有一些其它的比如数据归一化等预处理的工作要做，因此，我们把这些工作封装成一个dataset类来完成：

```
class Dataset:
    def __init__(self, path_name):
        #训练集
        self.train_images = None
        self.train_labels = None
        #验证集
        self.valid_images = None
        self.valid_labels = None
        #测试集
        self.test_images = None
        self.test_labels = None
        #数据集加载路径
        self.path_name = path_name
        #当前库采用的维度顺序
        self.input_shape = None
    #加载数据集并按照交叉验证的原则划分数据集并进行相关预处理工作
    def load(self, img_rows = IMAGE_SIZE, img_cols = IMAGE_SIZE,
            img_channels = 3, nb_classes = 2):
        #加载数据集到内存
        images, labels = load_dataset(self.path_name)

        train_images, valid_images, train_labels, valid_labels = train_test_split(images, labels,
            test_size = 0.5, random_state = 1)
        #当前的维度顺序如果为'th'，则输入图片数据时的顺序为：channels,rows,cols，否则:rows,cols,channels
        #这部分代码就是根据keras库要求的维度顺序重组训练数据集
        if K.image_dim_ordering() == 'th':
            train_images = train_images.reshape(train_images.shape[0], img_channels, img_rows, img_cols)
            valid_images = valid_images.reshape(valid_images.shape[0], img_channels, img_rows, img_cols)
            test_images = test_images.reshape(test_images.shape[0], img_channels, img_rows, img_cols)
            self.input_shape = (img_channels, img_rows, img_cols)
        else:
            train_images = train_images.reshape(train_images.shape[0], img_rows, img_cols, img_channels)
            valid_images = valid_images.reshape(valid_images.shape[0], img_rows, img_cols, img_channels)
            test_images = test_images.reshape(test_images.shape[0], img_rows, img_cols, img_channels)
            self.input_shape = (img_rows, img_cols, img_channels)

        #输出训练集、验证集、测试集的数量
        print(train_images.shape[0], 'train samples')
        print(valid_images.shape[0], 'valid samples')
        print(test_images.shape[0], 'test samples')

        #我们的模型使用categorical_crossentropy作为损失函数，因此需要根据类别数量nb_classes将类别标签进行one-hot编码使其向量化，在这里我们的类别只有两种，经过转化后标签数据变为二维
        train_labels = np_utils.to_categorical(train_labels, nb_classes)
        valid_labels = np_utils.to_categorical(valid_labels, nb_classes)
        test_labels = np_utils.to_categorical(test_labels, nb_classes)

        #像素数据浮点化以便归一化
        train_images = train_images.astype('float32')
        valid_images = valid_images.astype('float32')
        test_images = test_images.astype('float32')
```

```
#将其归一化,图像的各像素值归一化到0~1区间
train_images /= 255.0
valid_images /= 255.0
test_images /= 255.0

self.train_images = train_images
self.valid_images = valid_images
self.test_images = test_images
self.train_labels = train_labels
self.valid_labels = valid_labels
self.test_labels = test_labels
```

我们构建了一个Dataset类，用于数据加载及预处理。其中，__init__()为类的初始化函数，load()则完成实际的数据加载及预处理工作。关于预处理，我们主要做了以下几项准备工作：

- 1) 按照交叉验证的原则将数据集划分成三部分：训练集、验证集、测试集。
- 2) 按照keras库运行的后端系统要求改变图像数据的维度顺序。
- 3) 将数据标签进行one-hot编码，使其向量化。
- 4) 归一化图像数据。

关于第一项工作，先简单说说什么是交叉验证？交叉验证属于机器学习中常用的精度测试方法，它的目的是提升模型的可靠和稳定性。我们会拿出大部分数据用于模型训练，小部分数据用于对训练后的模型验证，验证结果会与验证集真实值（即标签值）比较并计算出差平方和，此项工作重复进行，直至所有验证结果与真实值相同，交叉验证结束，模型交付使用。在这里我们导入了sklearn库的交叉验证模块，利用函数train_test_split()来划分训练集和验证集，具体语句如下：

```
train_images, valid_images, train_labels, valid_labels = train_test_split(images, labels, test
```

train_test_split()会根据test_size参数按比例划分数据集（不要被test_size的外表所迷惑，它只是用来指定数据集划分比例的，本质上与测试无关，划分完了你爱咋用就咋用），在这里我们划分出了30%的数据用于验证，70%用于训练模型。参数random_state用于指定一个随机数种子，从全部数据中随机选取数据建立训练集和验证集，所以你将会看到每次训练的结果都会稍有不同。当然，为了省事，测试集我也调用了这个函数：

```
_, test_images, _, test_labels = train_test_split(images, labels, test_size = 0.5, random_stat
```

在这里，测试集我选择的比例为0.5，所以前面的“_, test_images, _, test_labels”语句你调个顺序也成，即“test_images, _, test_labels, _”，但是如果你改成其它数值，就必须严格按照代码给出的顺序才能得到你想要的结果。train_test_split()函数会按照训练集特征数据（这里就是图像数据）、测试集特征数据、训练集标签、测试集标签的顺序返回各数据集。所以，看你的选择了。

关于第二项工作，我们前面不止一次说过keras建立在tensorflow或theano基础上，换句话说，keras的后端系统可以是tensorflow也可以是theano。后端系统决定了图像数据输入CNN网络时的维度顺序，tensorflow的维度顺序为行数(rows)、列数(cols)、通道数(颜色通道, channels); theano则是通道数、行数、列数。所以，我们通过调用image_dim_ordering()函数来确定后端系统的类型（‘th’代表theano, ‘tf’代表tensorflow），然后再通过numpy提供的reshape()函数重新调整数组维度。

关于第三项工作，对标签集进行one-hot编码的原因是我们的训练模型采用categorical_crossentropy作为损失函数（多分类问题的常用函数，后面会详解），这个函数要求标签集必须采用one-hot编码形式。所以，我们对训练集、验证集和测试集标签均做了编码转换。那么什么是one-hot编码呢？one-hot有的翻译成独热，有的翻译成一位有效，个人感觉一位有效更直白一些。因为one-hot编码采用状态寄存器的组织方式对状态进行编码，每个状态值对应一个寄存器位，且任意时刻，只有一位有效。对于我们的程序来说，我们类别状态只有两种（nb_classes = 2）：0和1，0代表我，1代表闺女。one-hot编码会提供两个寄存器位保存这两个状态，如果标签值为0，则编码后值为[1 0]，代表第一位有效；如果为1，则编码后值为[0 1]，代表第2为有效。换句话说，one-hot编码将数值变成了位置信息，使其向量化，这样更方便CNN操作。

关于第四项工作，数据集先浮点后归一化的目的是提升网络收敛速度，减少训练时间，同时适应值域在（0,1）之间的激活函数，增大区分度。其实归一化有一个特别重要的原因是确保特征值权重一致。举个例子，我们使用mse这样的均方误差函数时，大的特征数值比如(5000-1000)²与小的特征值(3-1)²相加再求平均得到的误差值，显然大值对误差值的影响最大，但大部分情况下，特征值的权重应该是一样的，只是因为单位不同才导致数值相差甚大。因此，我们提前对特征数据做归一化处理，以解决此类问题。

8.4 视频识别

最后我们举一个Keras神经网络的高级应用示例——建立我们自己的视频识别卷积神经网络模型，激动吧？同前面的例子一样，我们将模型构建成一个类来使用，新建的这个模型类命名为Model。

```
#CNN网络模型类
class Model:
    def __init__(self):
        self.model = None

    #建立模型
    def build_model(self, dataset, nb_classes = 2):
        #构建一个空的网络模型，它是一个线性堆叠模型，各神经网络层会被顺序添加，专业名称为序贯模型或线
        self.model = Sequential()

        #以下代码将顺序添加CNN网络需要的各层，一个add就是一个网络层
        self.model.add(Convolution2D(32, 3, 3, border_mode='same',
            input_shape = dataset.input_shape))    #1 2维卷积层
        self.model.add(Activation('relu'))        #2 激活函数层

        self.model.add(Convolution2D(32, 3, 3))    #3 2维卷积层
        self.model.add(Activation('relu'))        #4 激活函数层

        self.model.add(MaxPooling2D(pool_size=(2, 2)))    #5 池化层
        self.model.add(Dropout(0.25))                #6 Dropout层

        self.model.add(Convolution2D(64, 3, 3, border_mode='same')) #7 卷积
        self.model.add(Activation('relu'))            #8 激活函数层

        self.model.add(Convolution2D(64, 3, 3))    #9 2维卷积层
        self.model.add(Activation('relu'))        #10 激活函数层

        self.model.add(MaxPooling2D(pool_size=(2, 2)))    #11 池化层
        self.model.add(Dropout(0.25))                #12 Dropout层

        self.model.add(Flatten())                    #13 Flatten层
        self.model.add(Dense(512))                    #14 Dense层,又被称作全连接层
        self.model.add(Activation('relu'))            #15 激活函数层
        self.model.add(Dropout(0.5))                #16 Dropout层
        self.model.add(Dense(nb_classes))            #17 Dense层
        self.model.add(Activation('softmax'))        #18 分类层,输出最终结果

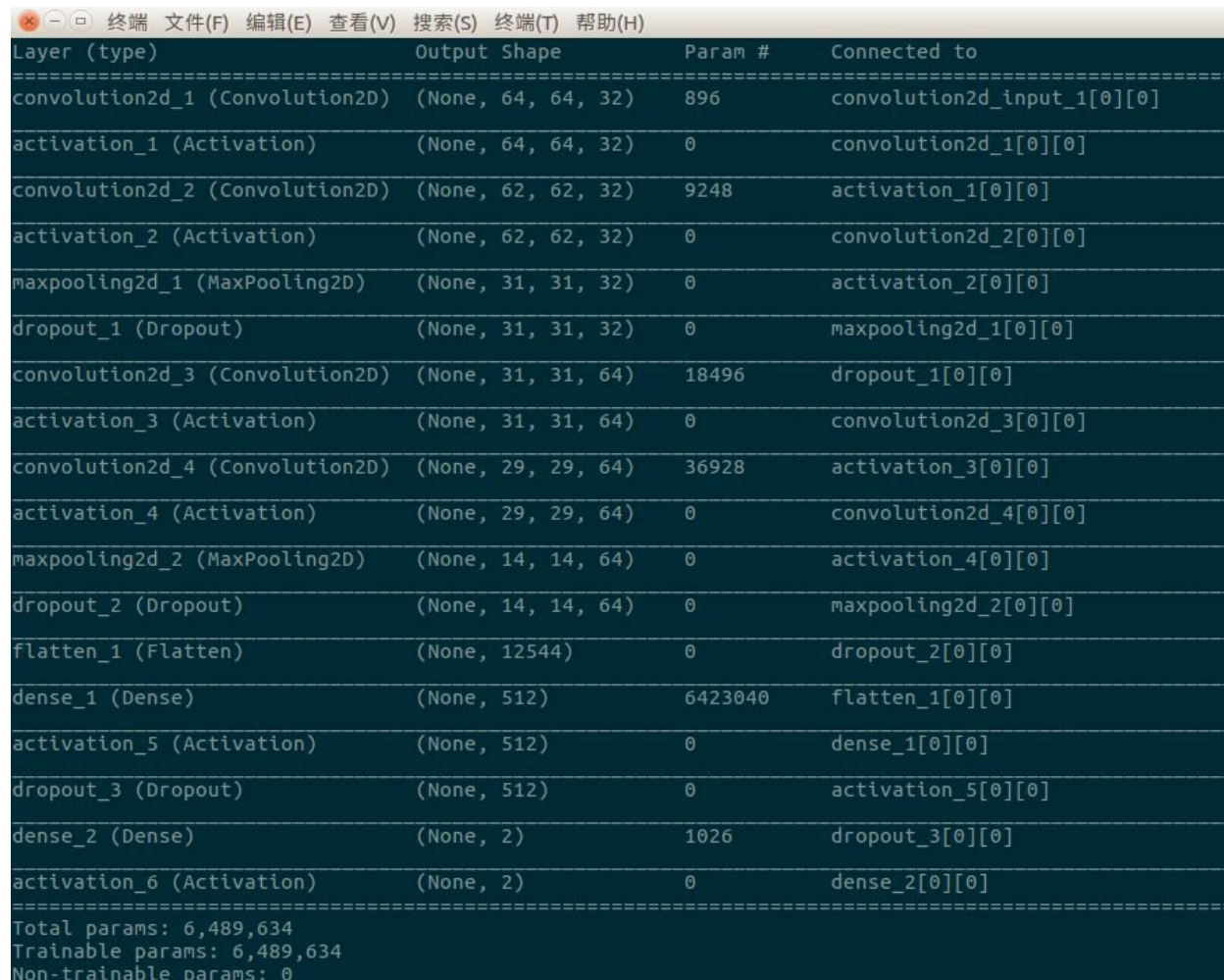
    #输出模型概况
    self.model.summary()

if __name__ == '__main__':
    dataset = Dataset('./data/')
    dataset.load()
    model = Model()
    model.build_model(dataset)
```

然后在控制台输入：

```
python face_train_use_keras.py
```

如果你没敲错代码，一切顺利的话，你应该看到类似下面这样的输出内容：

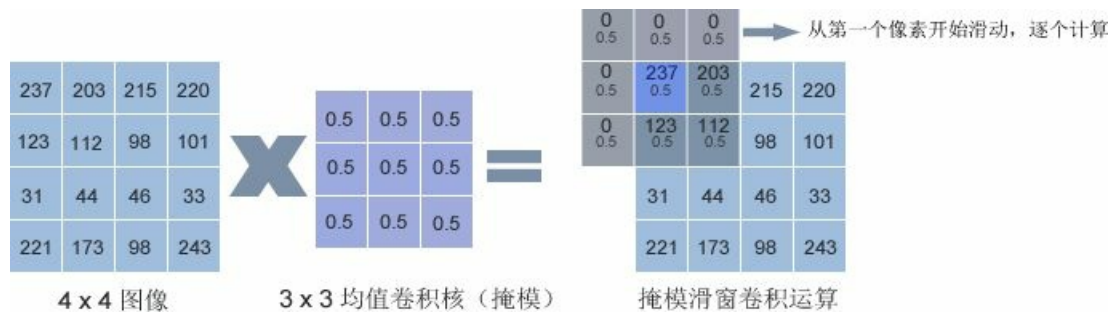


Layer (type)	Output Shape	Param #	Connected to
convolution2d_1 (Convolution2D)	(None, 64, 64, 32)	896	convolution2d_input_1[0][0]
activation_1 (Activation)	(None, 64, 64, 32)	0	convolution2d_1[0][0]
convolution2d_2 (Convolution2D)	(None, 62, 62, 32)	9248	activation_1[0][0]
activation_2 (Activation)	(None, 62, 62, 32)	0	convolution2d_2[0][0]
maxpooling2d_1 (MaxPooling2D)	(None, 31, 31, 32)	0	activation_2[0][0]
dropout_1 (Dropout)	(None, 31, 31, 32)	0	maxpooling2d_1[0][0]
convolution2d_3 (Convolution2D)	(None, 31, 31, 64)	18496	dropout_1[0][0]
activation_3 (Activation)	(None, 31, 31, 64)	0	convolution2d_3[0][0]
convolution2d_4 (Convolution2D)	(None, 29, 29, 64)	36928	activation_3[0][0]
activation_4 (Activation)	(None, 29, 29, 64)	0	convolution2d_4[0][0]
maxpooling2d_2 (MaxPooling2D)	(None, 14, 14, 64)	0	activation_4[0][0]
dropout_2 (Dropout)	(None, 14, 14, 64)	0	maxpooling2d_2[0][0]
flatten_1 (Flatten)	(None, 12544)	0	dropout_2[0][0]
dense_1 (Dense)	(None, 512)	6423040	flatten_1[0][0]
activation_5 (Activation)	(None, 512)	0	dense_1[0][0]
dropout_3 (Dropout)	(None, 512)	0	activation_5[0][0]
dense_2 (Dense)	(None, 2)	1026	dropout_3[0][0]
activation_6 (Activation)	(None, 2)	0	dense_2[0][0]
Total params: 6,489,634			
Trainable params: 6,489,634			
Non-trainable params: 0			

我们通过调用`self.model.summary()`函数将网络模型基本结构信息展示在我们面前，包括层类型、维度、参数个数、层连接等信息，一目了然，简洁、清晰。通过上图我们可以看出，这个网络模型共18层，包括4个卷积层、5个激活函数层、2个池化层（pooling layer）、3个Dropout层、2个全连接层、1个Flatten层、1个分类层，训练参数为6,489,634个，还是很可观的。

你看，这个实际运作的网络比我们上次给出的那个3层卷积的网络复杂多了，多了池化、Dropout、Dense、Flatten以及最终的分层，这些都是些什么鬼东西，需要我们逐个理一理：

卷积层（convolution layer）：这一层前面讲了太多，这里重点讲讲`Convolution2D()`函数。根据keras官方文档描述，2D代表这是一个2维卷积，其功能为对2维输入进行滑窗卷积计算。我们的脸部图像尺寸为64*64，拥有长、宽两维，所以在这里我们使用2维卷积函数计算卷积。所谓的滑窗计算，其实就是利用卷积核逐个像素、顺序进行计算，如下图：



上图选择了最简单的均值卷积核，3x3大小，我们用这个卷积核作为掩模对前面4x4大小的图像逐个像素作卷积运算。首先我们将卷积核中心对准图像第一个像素，在这里就是像素值为237的那个像素。卷积核覆盖的区域（掩模之称即由此来），其下所有像素取均值然后相加：

$$C(1) = 0 * 0.5 + 0 * 0.5 + 0 * 0.5 + 0 * 0.5 + 237 * 0.5 + 203 * 0.5 + 0 * 0.5 + 123 * 0.5 + 112 * 0.5$$

结果直接替换卷积核中心覆盖的像素值，接着是第二个像素、然后第三个，从左至右，由上到下.....以此类推，卷积核逐个覆盖所有像素。整个操作过程就像一个滑动的窗口逐个滑过所有像素，最终生成一副尺寸相同但已经过卷积处理的图像。上图我们采用的是均值卷积核，实际效果就是将图像变模糊了。显然，卷积核覆盖图像边界像素时，会有部分区域越界，越界的部分我们以0填充，如上图。对于此种情况，还有一种处理方法，就是丢掉边界像素，从覆盖区域不越界的像素开始计算。像上图，如果采用丢掉边界像素的方法，3x3的卷积核就应该从第2行第2列的像素（值为112）开始，到第3行第3列结束，最终我们会得到一个2x2的图像。这种处理方式会丢掉图像的边界特征；而第一种方式则保留了图像的边界特征。在我们建立的模型中，卷积层采用哪种方式处理图像边界，卷积核尺寸有多大等参数都可以通过Convolution2D()函数来指定：

```
self.model.add(Convolution2D(32, 3, 3, border_mode='same', input_shape = dataset.input_shape))
```

第一个卷积层包含32个卷积核，每个卷积核大小为3x3，border_mode值为“same”意味着我们采用保留边界特征的方式滑窗，而值“valid”则指定丢掉边界像素。根据keras开发文档的说明，当我们将卷积层作为网络的第一层时，我们还应指定input_shape参数，显式地告知输入数据的形状，对我们的程序来说，input_shape的值为(64,64,3)，来自Dataset类，代表64x64的彩色RGB图像。

激活函数层：它的作用前面已经说了，这里讲一下代码中采用的relu（Rectified Linear Units，修正线性单元）函数，它的数学形式如下：

$$f(x) = \max(0, x)$$

这个函数非常简单，其输出一目了然，小于0的输入，输出全部为0，大于0的则输入与输出相等。该函数的优点是收敛速度快，除了它，keras库还支持其它几种激活函数，如下：

- softplus

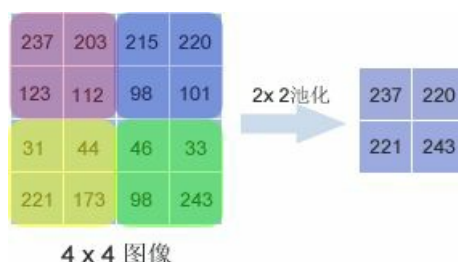
- softsign
- tanh
- sigmoid
- hard_sigmoid
- linear

它们的函数式、优缺点度娘会告诉你，不多说。对于不同的需求，我们可以选择不同的激活函数，这也是模型训练可调整的一部分，运用之妙，存乎一心，请自忖之。另外再交代一句，其实激活函数层按照我们前文所讲，其属于人工神经元的一部分，所以我们亦可以在构造层对象时通过传递activation参数设置，如下：

```
self.model.add(Convolution2D(32, 3, 3, border_mode='same', input_shape = dataset.input_shape))
self.model.add(Activation('relu'))      #设置为单独的激活层

#通过传递activation参数设置，与上两行代码的作用相同
self.model.add(Convolution2D(32, 3, 3, border_mode='same', input_shape = dataset.input_shape,
```

池化层（pooling layer）：池化层存在的目的是缩小输入的特征图，简化网络计算复杂度；同时进行特征压缩，突出主要特征。我们通过调用MaxPooling2D()函数建立了池化层，这个函数采用了最大值池化法，这个方法选取覆盖区域的最大值作为区域主要特征组成新的缩小后的特征图：



显然，池化层与卷积层覆盖区域的方法不同，前者按照池化尺寸逐块覆盖特征图，卷积层则是逐个像素滑动覆盖。对于我们输入的64x64的脸部特征图来说，经过2x2池化后，图像变为32x32大小。

Dropout层：随机断开一定百分比的输入神经元链接，以防止过拟合。那么什么是过拟合呢？一句话解释就是训练数据预测准确率很高，测试数据预测准确率很低，用图形表示就是拟合曲线较尖，不平滑。导致这种现象的原因是模型的参数很多，但训练样本太少，导致模型拟合过度。为了解决这个问题，Dropout层将有意识的随机减少模型参数，让模型变得简单，而越简单的模型越不容易产生过拟合。代码中Dropout()函数只有一个输入参数——指定抛弃比率，范围为0~1之间的浮点数，其实就是百分比。这个参数亦是一个可调参数，我们可以根据训练结果调整它以达到更好的模型成熟度。

Flatten层：截止到Flatten层之前，在网络中流动的数据还是多维的（对于我们的程序就是2维的），经过多次的卷积、池化、Dropout之后，到了这里就可以进入全连接层做最后的

处理了。全连接层要求输入的数据必须是一维的，因此，我们必须把输入数据“压扁”成一维后才能进入全连接层，Flatten层的作用即在于此。该层的作用如此纯粹，因此反映到代码上我们看到它不需要任何输入参数。

全连接层（dense layer）：全连接层的作用就是用于分类或回归，对于我们来说就是分类。keras将全连接层定义为Dense层，其含义就是这里的神经元连接非常“稠密”。我们通过Dense()函数定义全连接层。这个函数的一个必填参数就是神经元个数，其实就是指该层有多少个输出。在我们的代码中，第一个全连接层（#14 Dense层）指定了512个神经元，也就是保留了512个特征输出到下一层。这个参数可以根据实际训练情况进行调整，依然是没有可参考的调整标准，自调之。

分类层：全连接层最终的目的就是完成我们的分类要求：0或者1，模型构建代码的最后两行完成此项工作：

```
self.model.add(Dense(nb_classes))          #17 Dense层
self.model.add(Activation('softmax'))      #18 分类层，输出最终结果
```

第17层我们按照实际的分类要求指定神经元个数，对于我们来说就是2，18层我们通过softmax函数完成最终分类。关于softmax函数，其函数式如下：

$$a_j^L = \frac{e^{z_j^L}}{\sum_{k=1}^N e^{z_k^L}}$$

a_j^L 代表第L层第j个神经元的输出， z_j^L 代表第L层第j个神经元的输入，我们用单个神经元的输入结合自然常数e做指数运算，运算结果除以所有L层神经元输入的指数运算之和，就得到了一个介于0~1之间的浮点值 a_j^L 。显然，从上述公式很容易看出，所有神经元输出之和肯定为1：

$$\sum_{j=1}^N a_j^L = 1$$

这个值其实就是第j个神经元在所有神经元输出中所占的百分比。从分类的角度来说，该神经元的输出值越大，其对应的类别为真实类别的可能性就越大。因此，经过softmax函数，上层的N个输入被映射成N个概率分布，概率之和为1，概率值最大者即为模型预测的类别。

好了，模型构建完毕，接下来构建训练代码，在build_model()函数下面继续添加如下代码：

```
#训练模型
def train(self, dataset, batch_size = 20, nb_epoch = 10, data_augmentation = True):
    sgd = SGD(lr = 0.01, decay = 1e-6,
              momentum = 0.9, nesterov = True)
    self.model.compile(loss='categorical_crossentropy',
                      optimizer=sgd,
```

```

        metrics=['accuracy'])

#不使用数据提升，所谓的提升就是从我们提供的训练数据中利用旋转、翻转、加噪声等方法创造新的
#训练数据，有意识的提升训练数据规模，增加模型训练量
if not data_augmentation:
    self.model.fit(dataset.train_images,
                    dataset.train_labels,
                    batch_size = batch_size,
                    nb_epoch = nb_epoch,
                    validation_data = (dataset.valid_images, dataset.valid_labels),
                    shuffle = True)
else:
    datagen = ImageDataGenerator(
        featurewise_center = False,
        samplewise_center = False,
        featurewise_std_normalization = False,
        samplewise_std_normalization = False,
        zca_whitening = False,
        rotation_range = 20,
        width_shift_range = 0.2,
        height_shift_range = 0.2,
        horizontal_flip = True,
        vertical_flip = False)

    #计算整个训练样本集的数量以用于特征值归一化、ZCA白化等处理
    datagen.fit(dataset.train_images)
    #利用生成器开始训练模型
    self.model.fit_generator(datagen.flow(dataset.train_images, dataset.train_labels, batch_size=batch_size),
                             nb_epoch=nb_epoch, validation_data=(dataset.valid_images, dataset.valid_labels),
                             metrics=['accuracy'])

```

按照习惯，这里先不解释代码，先看执行结果，程序执行前添加如下一行代码：

#先前添加的测试build_model()函数的代码

```
model.build_model(dataset)
```

#测试训练函数的代码

```
model.train(dataset)
```

保存，控制台输入：

```
python face_train_use_keras.py
```

训练结果如下：

```

I tensorflow/core/common_runtime/gpu/gpu_device.cc:885] Found device 0 with properties:
name: GeForce GT 750M
major: 3 minor: 0 memoryClockRate (GHz) 0.967
pciBusID 0000:08:00:00
Total memory: 1.95GiB
Free memory: 1.78GiB
I tensorflow/core/common_runtime/gpu/gpu_device.cc:906] DMA: 0
I tensorflow/core/common_runtime/gpu/gpu_device.cc:916] 0: Y
I tensorflow/core/common_runtime/gpu/gpu_device.cc:975] Creating TensorFlow device (/gpu:0) -> (device: 0, name:
00.0)
840/840 [=====] - 7s - loss: 0.5476 - acc: 0.6833 - val_loss: 0.0423 - val_acc: 0.9944
Epoch 2/10
840/840 [=====] - 5s - loss: 0.1838 - acc: 0.9369 - val_loss: 0.0171 - val_acc: 0.9944
Epoch 3/10
840/840 [=====] - 5s - loss: 0.0919 - acc: 0.9690 - val_loss: 0.0335 - val_acc: 0.9944
Epoch 4/10
840/840 [=====] - 5s - loss: 0.1615 - acc: 0.9488 - val_loss: 0.0164 - val_acc: 0.9972
Epoch 5/10
840/840 [=====] - 5s - loss: 0.0660 - acc: 0.9786 - val_loss: 0.0220 - val_acc: 0.9944
Epoch 6/10
840/840 [=====] - 5s - loss: 0.0991 - acc: 0.9679 - val_loss: 0.0158 - val_acc: 0.9972
Epoch 7/10
840/840 [=====] - 5s - loss: 0.0297 - acc: 0.9881 - val_loss: 0.0226 - val_acc: 0.9944
Epoch 8/10
840/840 [=====] - 5s - loss: 0.0515 - acc: 0.9833 - val_loss: 0.0192 - val_acc: 0.9944
Epoch 9/10
840/840 [=====] - 5s - loss: 0.0505 - acc: 0.9869 - val_loss: 0.0180 - val_acc: 0.9972
Epoch 10/10
840/840 [=====] - 5s - loss: 0.0529 - acc: 0.9893 - val_loss: 0.0377 - val_acc: 0.9917

```

我们共进行了10轮次训练（nb_epoch = 10），每轮42次迭代（840 / 20，训练集1200 x（1-0.3）= 840），每次迭代训练使用20个样本（batch_size = 20），得到的训练结果还不错（以第10轮次训练结果为例）：

训练误差（**loss**）：0.0529

训练准确率（**acc**）：0.9893

验证误差（**val_loss**）：0.0377

验证准确率（**val_acc**）：0.9917

验证集准确率高达99%，至少从验证结果上看模型已达实用化要求，下一步可以用测试数据集对其进行测试了。添加测试代码之前，我们需要对训练代码中几个关键函数交代一下。首先是优化器函数，优化器用于训练模型，它的作用就是调整训练参数（权重和偏置值）使其最优，确保e值最小。keras提供了很多优化器，我们在这里采用的SGD就是其中一种，它就是机器学习领域最著名的随机梯度下降法。函数第一个参数lr用于指定学习效率（lr，Learning Rate，参见系列4），其值为大于0的浮点数。decay指定每次更新后学习效率的衰减值，这个值一定很小（1e-6，0.000 001），否则速率会衰减很快。momentum指定动量值。SGD方法有一个明显的缺点就是，它的下降方向完全依赖当前的训练样本（batch），因此其优化十分不稳定。为了解决这个问题，大牛们引进了动量

（momentum），用它来模拟物体运动时的惯性，让优化器在一定程度上保留之前的优化方向，同时利用当前样本微调最终的优化方向，这样即能增加稳定性，提高学习速度，又在一定程度上避免了陷入局部最优陷阱。参数momentum即用于指定在多大程度上保留原有方向，其值为0~1之间的浮点数。一般来说，选择一个在0.5~0.9之间的数即可。代码中SGD函数的最后一个参数nesterov则用于指定是否采用nesterov动量方法，nesterov momentum是对传统动量法的一个改进方法，其效率更高，关于它的详细介绍可参考如下链接：

http://www.360doc.com/content/16/1010/08/36492363_597225745.shtml

对于`compile()`函数，其作用就是编译模型以完成实际的配置工作，为接下来的模型训练做好准备。换句话说，`compile`之后模型就可以开始训练了。这个函数有一个很重要的参数：`loss`，它用于指定一个损失函数。所谓损失函数，通俗地说，它是统计学中衡量损失和错误程度的函数，显然，其值越小，模型就越好，这个函数其实就是我们的优化对象。代码中`loss`的值为“`categorical_crossentropy`”，常用于多分类问题，其与激活函数`softmax`配对使用（我们的类别只有两种，也可采用“`binary_crossentropy`”二值分类函数，该函数与`sigmoid`配对使用，注意如果采用它就不需要`one-hot`编码）。参数`metrics`用于指定模型评价指标，参数值“`accuracy`”表示用准确率来评价。

接着就是数据提升，我们可以选择不提升，也就是采用原始训练集和验证集，这时我们直接调用`model.fit()`函数即可开始模型训练。该函数`shuffle`参数用于指定是否随机打乱数据集。一般来说选择数据提升要比不提升好，这样可以让我们利用有限数量的图片获得无限数量的训练图片。因为我们一旦选择数据提升，`ImageDataGenerator()`函数返回的生成器会在模型训练时无限生成训练数据，直至所有训练轮次（`epoch`）结束（对我们的代码来说就是`840 x 10`，生成了8400张图片）。`model.fit_generator()`函数使用生成器开始模型训练。

在这里需要重点交代一下`batch_size`和`nb_epoch`两个参数。`nb_epoch`指定模型需要训练多少轮次，使用训练集全部样本训练一次为一个训练轮次。根据模型成熟度，我们可以适当调整该值以增加或减少训练次数。`batch_size`则是一个影响模型训练结果的重要参数。我们知道，一个训练轮次要经过多次迭代训练才能让模型逐渐趋向本轮最优，这是因为理论上每次迭代训练结束后，模型都应该朝着梯度下降的方向前进一步，直至全部样本训练完毕，模型梯度到达本轮最小点。之所以说理论上，是因为决定梯度方向的是训练样本，每次迭代训练选取的样本——其决定的下降方向能否很好的代表样本全体，直接决定了模型能否到达正确的极值点。对于小的训练集，我们完全可以采用全数据集的方式进行训练，因为，全数据集确定的方向肯定能代表正确方向。但这样做对大的训练集就很不现实，因为内存有限，无法一次载入全部数据。于是，批梯度下降法（`Mini-batches Learning`）应运而生。我们一次选取适当数量的训练样本（视内存大小，可多可少），逐批次迭代，直至本轮全部样本训练完毕。参数`batch_size`的作用即在于此，其指定每次迭代训练样本的数量。该值的选取非常讲究，不能盲目的增大或减小，因为`batch_size`太大或太小都会让模型训练效率变慢。显然，`batch_size`肯定存在一个局部最优值，这需要我们慢慢调试，调试时可从一个小值开始，慢慢加大，直至到达一个合理值。

现在模型训练的工作已经完成，接下来我们就要考虑模型使用的问题了。要想使用模型，我们必须能够把模型保存下来，因此，我们继续为`Model`类添加两个函数：

```
MODEL_PATH = './me.face.model.h5'
def save_model(self, file_path = MODEL_PATH):
    self.model.save(file_path)
def load_model(self, file_path = MODEL_PATH):
    self.model = load_model(file_path)
```

一个函数用于保存模型，一个函数用于加载模型。`keras`库利用了压缩效率更高的HDF5保存模型，所以我们用“`.h5`”作为文件后缀。上述代码添加完毕后，我们接着在文件尾部添加测试代码，把模型训练好并把模型保存下来：

```
if __name__ == '__main__':
```

```
dataset = Dataset('./data/')
dataset.load()
model = Model()
model.build_model(dataset)
model.train(dataset)
model.save_model(file_path = './model/me.face.model.h5')
```

执行上述代码，顺利的话，我们应当看到模型保存文件出现在model文件夹下了：

好了，接下来我们就要用前面Dataset类提供的测试集测试模型了。首先，我们为Model类添加一个模型评估函数：

```
def evaluate(self, dataset):
    score = self.model.evaluate(dataset.test_images, dataset.test_labels, verbose = 1)
    print("%s: %.2f%%" % (self.model.metrics_names[1], score[1] * 100))
```

然后，继续添加测试代码：

```
if __name__ == '__main__':
    dataset = Dataset('./data/')
    dataset.load()
    #评估模型
    model = Model()
    model.load_model(file_path = './model/me.face.model.h5')
    model.evaluate(dataset)
```

执行结果如下：

```
600/600 [=====] - 5s
acc: 99.50%
```

准确率99.5%，相当高的评估结果了。

至此，我们完成了模型建立工作，下一篇博文讨论如何用它识别出“我”了。

从实时视频流识别出“我”

终于到了最后一步，激动时刻就要来临了，先平复一下心情，把剩下的代码加上，首先是为Model类增加一个预测函数：

```
#识别人脸
def face_predict(self, image):
    #依然是根据后端系统确定维度顺序
    if K.image_dim_ordering() == 'th' and image.shape != (1, 3, IMAGE_SIZE, IMAGE_SIZE):
        image = resize_image(image)
        image = image.reshape((1, 3, IMAGE_SIZE, IMAGE_SIZE))
    elif K.image_dim_ordering() == 'tf' and image.shape != (1, IMAGE_SIZE, IMAGE_SIZE, 3):
        image = resize_image(image)
        image = image.reshape((1, IMAGE_SIZE, IMAGE_SIZE, 3))
```

```

#浮点并归一化
image = image.astype('float32')
image /= 255.0
#给出输入属于各个类别的概率，这里是二值类别，输出属于0和1的概率各为多少
result = self.model.predict_proba(image)
print('result:', result)
#给出类别预测：0或者1
result = self.model.predict_classes(image)
#返回类别预测结果
return result[0]

```

这个函数是提供给外部模块使用的，外部模块用它来预测哪个是“我”，哪个不是“我”。代码很简单，注释也很详细，就不多解释了。接下来我们新建一个python文件：face_predict_use_keras.py，然后为这个文件添加如下代码：

```

#-*- coding: utf-8 -*-
import cv2
import sys
import gc
from face_train_use_keras import Model
if __name__ == '__main__':
    if len(sys.argv) != 2:
        print("Usage:%s camera_id\r\n" % (sys.argv[0]))
        sys.exit(0)
    #加载模型
    model = Model()
    model.load_model(file_path = './model/me.face.model.h5')
    #框住人脸的矩形边框颜色
    color = (0, 255, 0)
    #捕获指定摄像头的实时视频流
    cap = cv2.VideoCapture(int(sys.argv[1]))
    #人脸识别分类器本地存储路径
    cascade_path = "/OpenCV/haarcascades/haarcascade_frontalface_alt2.xml"
    #循环检测识别人脸
    while True:
        _, frame = cap.read() #读取一帧视频
        #图像灰化，降低计算复杂度
        frame_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        #使用人脸识别分类器，读入分类器
        cascade = cv2.CascadeClassifier(cascade_path)
        #利用分类器识别出哪个区域为人脸
        faceRects = cascade.detectMultiScale(frame_gray, scaleFactor = 1.2, minNeighbors = 3,
        if len(faceRects) > 0:
            for faceRect in faceRects:
                x, y, w, h = faceRect
                #截取脸部图像提交给模型识别这是谁
                image = frame[y - 10: y + h + 10, x - 10: x + w + 10]
                faceID = model.face_predict(image)
                #如果是“我”
                if faceID == 0:
                    cv2.rectangle(frame, (x - 10, y - 10), (x + w + 10, y + h + 10), color, th
                    #文字提示是谁
                    cv2.putText(frame, 'Dady',
                                (x + 30, y + 30),
                                cv2.FONT_HERSHEY_SIMPLEX,
                                1,
                                (255,0,255),
                                #坐标
                                #字体
                                #字号
                                #颜色

```


```
                2)                                #字的线宽
            else:
                pass
            cv2.imshow("识别朕", frame)
            #等待10毫秒看是否有按键输入
            k = cv2.waitKey(10)
            #如果输入q则退出循环
            if k & 0xFF == ord('q'):
                break
        #释放摄像头并销毁所有窗口
        cap.release()
        cv2.destroyAllWindows()
```

这个就是我们的最终程序，它能够从USB拍摄的实时视频流中找出哪一个是我。


8.5 用神经网络去噪

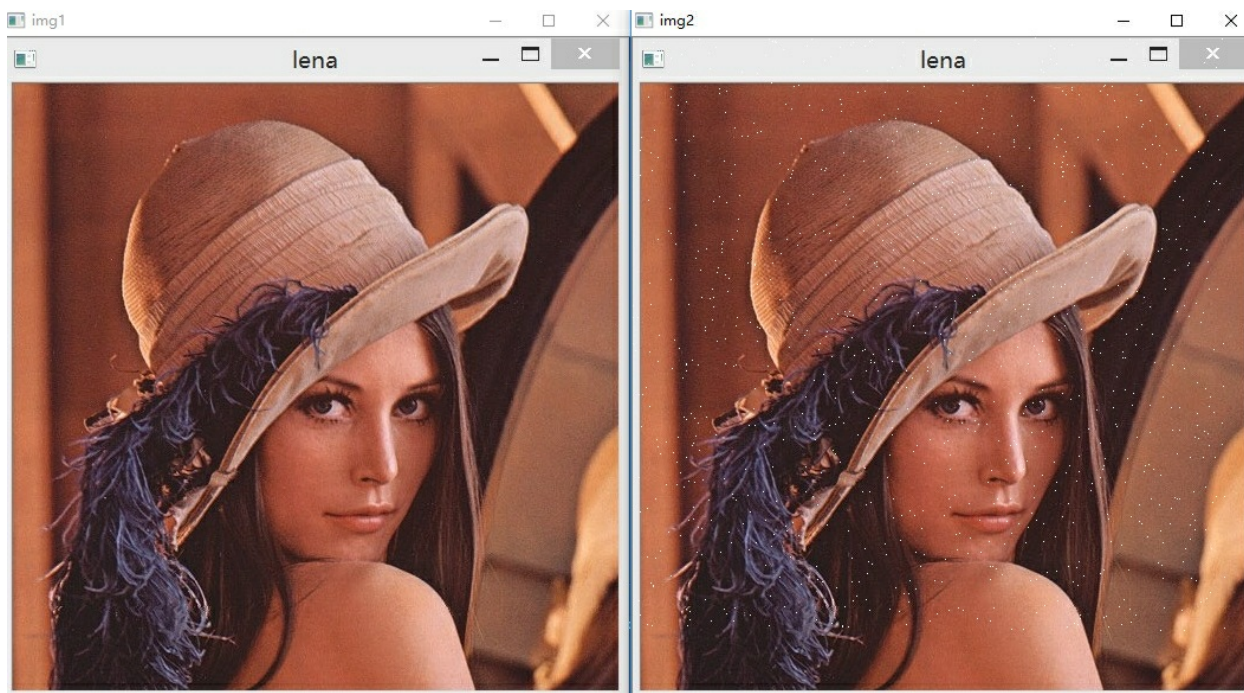
首先我们需要生成噪音数据，所谓的图像噪音实际就是修改某些像素点的值，使之偏离实际图像，噪音像素值和真实像素值相差越远说明噪音越大。我们先简单准备些噪音数据。

8.5.1 对图像添加噪音

导入lena.jpg图像，并读取到中，然后随即修改图像中的部分像素，这个例子中随即挑选1000个像素设置为白色（255），程序代码如下：

```
import cv2
import numpy as np
import random
file='f:\zookeeper\im1.jpg'
img = cv2.imread(file)
cv2.imshow("img1",img)
img2=img
for i in range(1000):
    randx = random.randint(0,500)
    randy = random.randint(0,500)
    img2[randx,randy,:]=255
cv2.imshow("img2",img2)
cv2.waitKey()
```

其中[randx,randy,:]=255的含义是这随机选取点的图像值设置为255，也就是白色（黑色是0），我们从处理过的结果中可以看到图像被添加了很多白点，这就是这段代码的作用。如下图所示。



8.5.2 去噪神经网络

Opencv提供了高斯滤波函数进行图像去噪处理，它纯粹是用像素的连续性来达到去噪的效果，无法体现像素特征的作用。下面我们换个思路，用神经网络来训练一个图像去噪模型，并看看它的实际运行效果。

同样，我们也采用mnist手写图像库来训练该模型，选择这个图像库的原因一是其图像是灰度一通道格式，数据量较小，二是因为mnist是通用的神经网络图像库之一，用它做训练数据有助于在不同模型之间进行比较。

我们选择训练轮次为10次，由于笔者用的笔记本是8G内存的CPU，性能一般，而每次训练都要花费几十分钟时间，为防止中途程序意外终止造成不必要的浪费，我们的训练代码在每轮迭代后选择将模型数据保存到磁盘，这样发生意外时可以从保存的模型数据直接开始新一轮的训练（前面的训练结果已保存在模型中）。

完整的代码如下所示：

```
from __future__ import print_function
import os
import struct
import numpy as np
import keras
from keras.datasets import mnist
from keras.models import Sequential, Model
from keras.layers import Input, Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D, UpSampling2D
from keras import backend as K
import cv2

def load_mnist(path, kind='train'):
```

```

"""Load MNIST data from `path`"""
labels_path = os.path.join(path, '%s-labels.idx1-ubyte' % kind)
images_path = os.path.join(path, '%s-images.idx3-ubyte' % kind)
with open(labels_path, 'rb') as lbpath:
    magic, n = struct.unpack('>II', lbpath.read(8))
    labels = np.fromfile(lbpath, dtype=np.uint8)
with open(images_path, 'rb') as imgpath:
    magic, num, rows, cols = struct.unpack(">IIII", imgpath.read(16))
    images = np.fromfile(imgpath, dtype=np.uint8).reshape(len(labels), 784)
return images, labels

X_train, y_train = load_mnist('./data', kind='train')
print("shape:", X_train.shape)
print('Rows: %d, columns: %d' % (X_train.shape[0], X_train.shape[1]))
X_test, y_test = load_mnist('./data', kind='t10k')
print('Rows: %d, columns: %d' % (X_test.shape[0], X_test.shape[1]))
print("x_train(1):", X_train[1].shape)

x_train = X_train.astype('float32') / 255.
x_test = X_test.astype('float32') / 255.
x_train = np.reshape(x_train, (len(x_train), 28, 28, 1))
x_test = np.reshape(x_test, (len(x_test), 28, 28, 1))

noise_factor = 0.5
x_train_noisy = x_train + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=x_train.shape)
x_test_noisy = x_test + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=x_test.shape)

x_train_noisy = np.clip(x_train_noisy, 0., 1.)
x_test_noisy = np.clip(x_test_noisy, 0., 1.)
print("x_train_noisy shape:", x_train_noisy.shape)

batch_size = 128
num_classes = 10
epochs = 1

# input image dimensions
img_rows, img_cols = 28, 28
input_shape=(28,28,1)
model = Sequential()
x = Conv2D(32, (3, 3), activation='relu', padding='same', input_shape=input_shape)
model.add(x)
x = MaxPooling2D((2, 2), padding='same')
model.add(x)
x = Conv2D(32, (3, 3), activation='relu', padding='same')
model.add(x)
encoded = MaxPooling2D((2, 2), padding='same')
model.add(encoded)

# at this point the representation is (32, 7, 7)

x = Conv2D(32, (3, 3), activation='relu', padding='same')
model.add(x)
x = UpSampling2D((2, 2))
model.add(x)
x = Conv2D(32, (3, 3), activation='relu', padding='same')
model.add(x)
x = UpSampling2D((2, 2))
model.add(x)
decoded = Conv2D(1, (3, 3), activation='sigmoid', padding='same')

```

```

model.add(decoded)
model.compile(optimizer='adadelata', loss='binary_crossentropy')

print("x_test_noisy shape:",x_test_noisy.shape)
print("x_test shape:",x_test.shape)

model.fit(x_train_noisy, x_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          validation_data=(x_test_noisy, x_test))
model.save("encode.model")
model.save_weights("encode.weight")
score = model.evaluate(x_test_noisy, x_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
for currentEpoch in range(1,10):
    ii=currentEpoch+1
    dir = 'D:\\tmp\\'+str(ii)
    prevdir = 'D:\\tmp\\'+str(currentEpoch)
    if not os.path.isdir(dir) :
        print("mkdir:"+dir)
        os.mkdir(dir)
    model.fit(x_train_noisy, x_train,
              batch_size=batch_size,
              epochs=epochs,
              verbose=1,
              validation_data=(x_test_noisy, x_test))
    model.save(dir+'\\encode.model')
    model.save_weights(dir+'\\encode.weight')
    index = 0
    image1=x_test_noisy[index]
    image2=x_test[index]
    result = model.predict(image1.reshape(1,28,28,1))
    print("result shape:",result.shape)
    result = result.reshape(28,28,1)
    result *= 255
    cv2.imwrite(dir+'\\result001.png',result)
    image1 = image1.astype('float32')
    image2 = image2.astype('float32')
    image *= 255
    image2 *= 255
    image1=image1.reshape(28,28,1)
    image2=image2.reshape(28,28,1)
    noise = X_test[index] + 0.5 * np.random.normal(loc=0.0, scale=100.0, size=X_train[index]).s
    noise = noise.reshape(28,28,1)
    cv2.imwrite(dir+'\\noise001.png',noise)
    cv2.imwrite(dir+'\\clean001.png',image2)

```

该模型的解释，将（28，28，1）格式的图像数据作为输入参数。图像先经过卷积-》池化-》卷积-》池化的过程，到达中间的encode层。然后将之前的模型反转，经过卷积-》upsample-》卷积-》upsample-》卷积。最后生成和输入同样维度的数据，也就是生成的图像，最后输出的数据也是（28，28，1）格式的。

神经网络会自动抽取特征并计算各层的权重参数，当训练的损失函数达到收敛状态则表示模型基本训练完成，这时候可以开始进行预测了。

模型的描述如下图：

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 28, 28, 32)	320
max_pooling2d_1 (MaxPooling2D)	(None, 14, 14, 32)	0
conv2d_2 (Conv2D)	(None, 14, 14, 32)	9248
max_pooling2d_2 (MaxPooling2D)	(None, 7, 7, 32)	0
conv2d_3 (Conv2D)	(None, 7, 7, 32)	9248
up_sampling2d_1 (UpSampling2D)	(None, 14, 14, 32)	0
conv2d_4 (Conv2D)	(None, 14, 14, 32)	9248
up_sampling2d_2 (UpSampling2D)	(None, 28, 28, 32)	0
conv2d_5 (Conv2D)	(None, 28, 28, 1)	289
Total params: 28,353		
Trainable params: 28,353		
Non-trainable params: 0		

预测的代码：

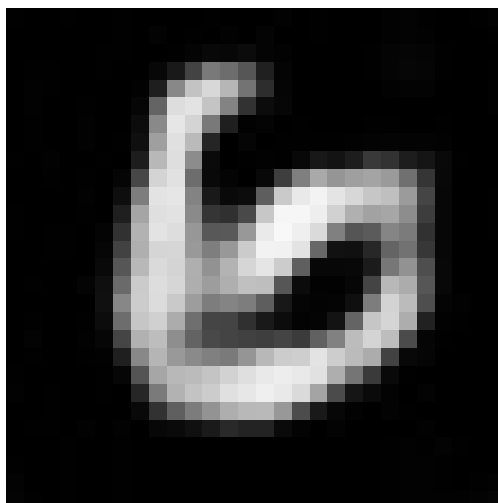
```
image1=x_test_noisy[index]
result = model.predict(image1.reshape(1,28,28,1))
print("result shape:",result.shape)
result = result.reshape(28,28,1)
result *= 255
cv2.imwrite(dir+'\\result001.png',result)
```

由于模型的输入参数在参加训练前已经进行了归一化处理，归一化的代码：

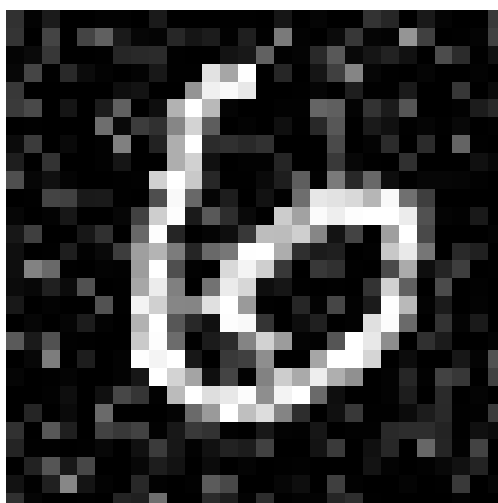
```
x_train = X_train.astype('float32') / 255.
```

因此模型的预测结果也是归一化的图像数据。

Predict的输出数据是（1，28，28，1）格式的，我们然后将输出数据重新规划成（28，28，1）维度大小，注意这两种格式元素数量是一样的。这是图像文件的维度格式，并且将归一化的像素值*255得到最终的像素值，保存到图像文件中，最终图像文件如图：



而预测的参数是一副噪音图像，图像为：



看到模型的实际效果还是不错的。

运行过程中会根据每个轮次自动创建目录，比如轮次1的目录为“1”，轮次2的目录为”“2”。

我们看生成的各个目录中的噪音图像和模型生成的结果，会发现后面轮次生成的图片清晰度比第一轮次的结果要好。同时观察训练结果发现到第10次时损失函数值越来越收敛了。说明再往下训练效果也会很有限。

8.5.3 模型的保存和恢复

如果发生意外，比如在训练到第5次时意外终止，此时第4次的训练好的模型和权重参数已经保存到“4”目录中，此时可通过以下代码恢复模型，而前面4轮的训练过程不需要再重复了，直接从第5轮开始继续训练。

```
dir='D:/tmp/4'
```

```
newdir='D:/tmp/5'
model = load_model(dir+'/encode.model')
model.summary()
model.fit(x_train_noisy, x_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          validation_data=(x_test_noisy, x_test))
model.save(newdir + '/encode.model')
model.save_weights(newdir + '/encode.weight')
```

通过load和save这两种操作，我们就不用担心训练时间过长而影响其他工作了，完全可以每次只运行一个轮次的训练过程，将及其耗时的神经网络训练分解成无数的轮次。当然如果有GPU设备就更理想了，通常情况下GPU训练时间比CPU要提高40倍左右。

8.6 视频抽取图片

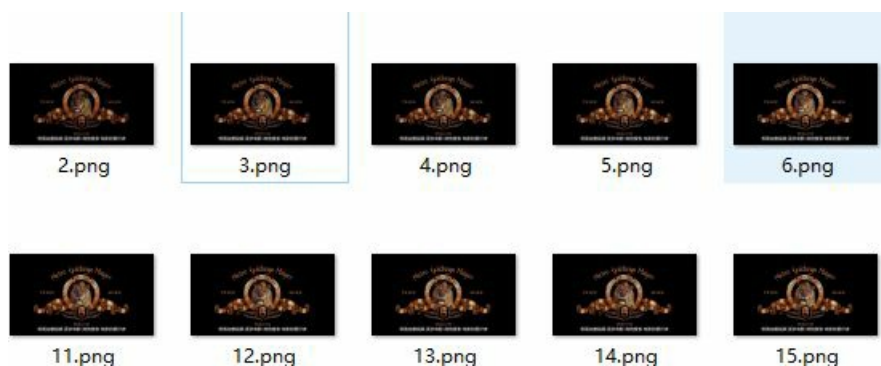
说完了图像识别，接下来看看如何给视频打标签。

视频其实就是一张张图片，利用opencv库可以很容易的从视频文件中抽离出图像文件来，下面我们就看一段示例代码是如何从视频文件中抽取出多张图片的。

例子：

```
import cv2
cap = cv2.VideoCapture("F:/ai/a_002.mp4")
success, frame = cap.read()
index = 1
while success :
    index = index+1
    cv2.imwrite(str(index)+".png",frame)
    if index > 20:
        break;
    success,frame = cap.read()
cap.release()
```

执行完上述代码后我们可以看到视频文件目录下生成了无数的图片文件，如图：



好了，下面来看看如何做视频文件的自动分类。这里面其实有两个步骤，一个是对每帧图片做智能识别，这可以用卷积图像神经网络来实现；另一个是对所有图片的识别结果做一个汇总，取数量最多的种类作为该视频的分类。

这是一个组合神经网络模型，具体的实现这里就不做具体的分析了，亲爱的读者，从中有没有想到一些很好玩的应用场景？比如通过神经网络对视频自动打标签；影视作品视频中通过对演员身份的自动识别来自动生成片段集锦等等，读者可自行补脑。感兴趣的读者可以等待笔者的下一本拙作中做进一步的讨论。

9 参考

- Keras中文文档

http://keras-cn.readthedocs.io/en/latest/for_beginners/concepts/

- 使用Numpy和Scipy处理图像

<http://reverland.org/python/2012/11/12/numpyscipy/#section-11>

- OPENCV图像处理提高（一）图像增强

http://blog.csdn.net/qq_25819827/article/details/52006841

- Multidimensional image processing (scipy.ndimage)官网

<https://docs.scipy.org/doc/scipy/reference/tutorial/ndimage.html#smoothing-filters>

- 深入理解CNN的细节

<http://www.thinksaas.cn/topics/0/491/491257.html>

- CNN眼中的世界

<http://blog.keras.io/how-convolutional-neural-networks-see-the-world.html>

- 理解LSTM网络

<http://www.jianshu.com/p/9dc9f41f0b29>

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们：

- 微博 @图灵教育：好书、活动每日播报
- 微博 @图灵社区：电子书和好文章的消息
- 微博 @图灵新知：图灵教育的科普小组
- 微信 图灵访谈：ituring_interview，讲述码农精彩人生
- 微信 图灵教育：turingbooks

图灵社区会员 llh_y（446181760@qq.com） 专享 尊重版权