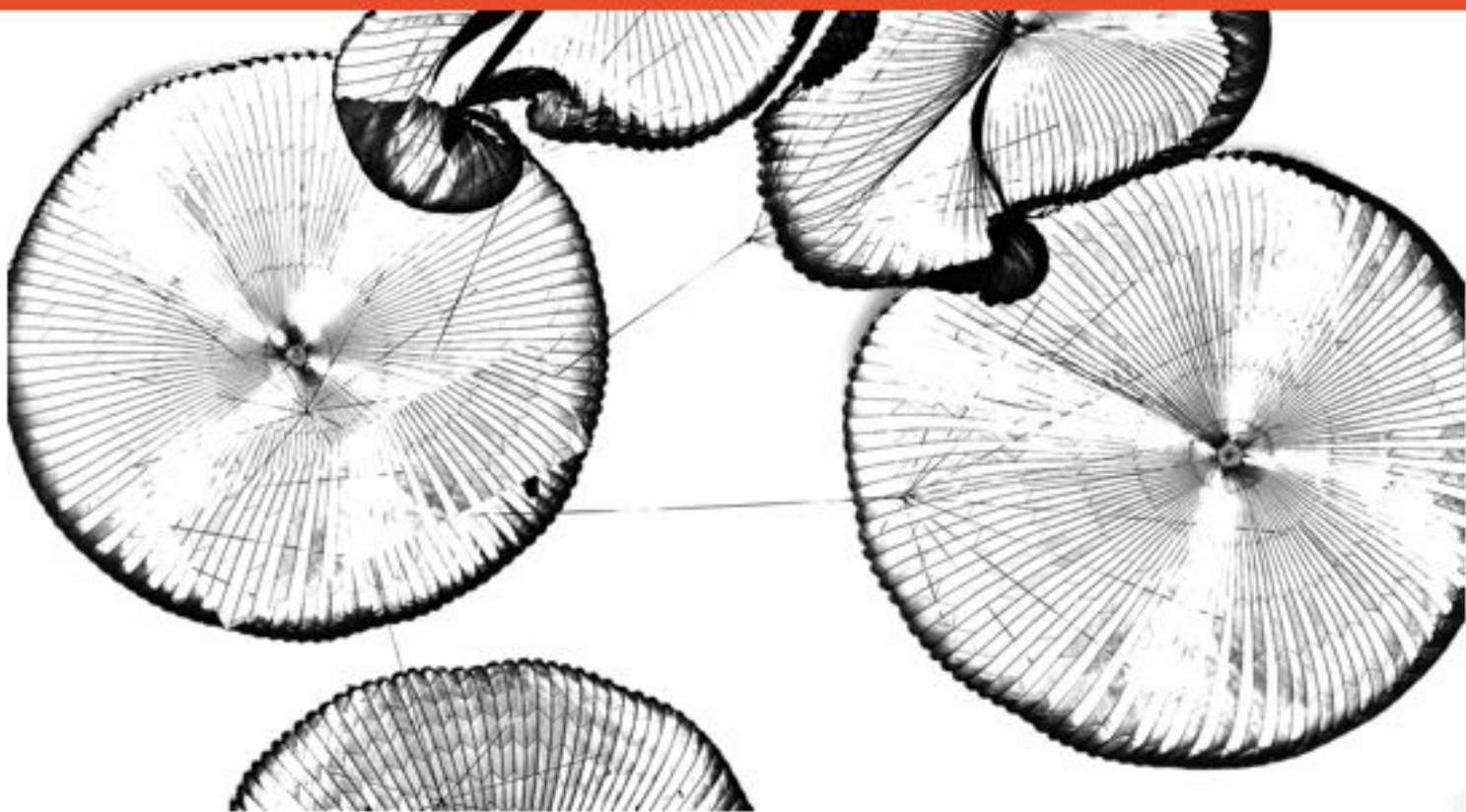
图灵电子书

PyTorch 深度学习实战

侯宜军 著



版权信息

书名：PyTorch深度学习实战

作者：侯宜军

内容摘要

PyTorch是Facebook发布的一款非常具有个性的深度学习框架，它和Tensorflow，Keras，Theano等其他深度学习框架都不同，它是动态计算图模式，其应用模型支持在运行过程中根据运行参数动态改变，而其他几种框架都是静态计算图模式，其模型在运行之前就已经确定。

本书共分成16个章节，第1章是Pytorch简介和环境搭建；第2～8章是Keras的软件框架说明，包含了层的说明，优化器和损失函数等；第9～10章重点介绍了深度学习中的卷积和池化的概念；第11～12章是介绍用神经网络模型搭建图像识别系统的实战经验；第13～16章介绍如何使用LSTM模型来处理自然语言。本书从原理到实战、深入浅出的介绍了Facebook人工智能利器Pytorch的卓越表现，只要认真读完本书，你就能掌握Pytorch的使用技巧了。本书具有很强的实战性。

本书的目标人群主要定位为具有一定Python编程基础，对机器学习和神经网络有一定了解的程序员们。

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。

具有大型电商网站实战经验，多年分布式系统源码研究，深度学习框架研究。对Keras，Pytorch，ZooKeeper，Spark，Kafka等大数据技术框架较熟悉。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 Facebook深度学习框架PyTorch

1.1 Pytorch是什么

PyTorch 是 Facebook 发布的一款非常具有个性的深度学习框架，它和 Tensorflow，Keras，Theano 等其他深度学习框架都不同，它是动态计算图模式，其应用模型支持在运行过程中根据运行参数动态改变，而其他几种框架都是静态计算图模式，其模型在运行之前就已经确定。

Python 模块可以通过 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。

（之所以要安装Keras是因为文中某些地方使用了Keras的数据预处理类包）

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。

1.2 Tensor

在 PyTorch 中 Tensor 代表多维数组，类似 TensorFlow 中的 matrix 或 ndarrays。

例如，生成一个 (5,3) 的 tensor：

```
x = torch.rand(5, 3)
print(x)
```

输出：

```
0.1290  0.8748  0.1096
0.7204  0.9472  0.2891
0.3106  0.1791  0.0739
0.2701  0.5434  0.9968
0.2098  0.9916  0.1561
```

获取它的大小：

```
print(x.size())
```

输出:

```
torch.Size([5, 3])
```

1.3 安装Python

安装 Python2.7 版本，笔者用的是阿里云服务器，现在申请的版本应该是 2.7 版本以上，笔者当时的版本是 2.6 版本，安装 numpy 等 python 库需要 2.7 版本以上，所以先安装 python2.7。

看当前版本：

```
[root@iZ25ix41uc3Z ~]# python --version
Python 2.6.6
```

安装 2.7 版本步骤：

```
1.  #wget http://python.org/ftp/python/2.7.3/Python-2.7.3.tar.bz2
2.  tar zxvf Python-2.7.3.tar.bz2
3.  cd Python-2.7.4
4.  ./configure
5.  make all
6.  make install
7.  make clean
8.  make distclean
```

```
cd /usr/local/bin

ls
```

看到 python2.7 已经安装到 /usr/local/bin 目录下了。

```
[root@iZ25ix41uc3Z bin]# /usr/local/bin/python2.7 --version
Python 2.7.4
```

新的 python 版本是 2.7.4

最后一步还要将系统默认 python 指向该新的 python2.7.4

```
[root@iZ25ix41uc3Z bin]# rm /usr/bin/python
rm：是否删除普通文件 "/usr/bin/python"？y
[root@iZ25ix41uc3Z bin]# ln -s /usr/local/bin/python2.7
python2.7         python2.7-config
[root@iZ25ix41uc3Z bin]# ln -s /usr/local/bin/python2.7 /usr/bin/python
[root@iZ25ix41uc3Z bin]# ls -l /usr/bin/python
lrwxrwxrwx 1 root root 24 6月  25 12:56 /usr/bin/python -> /usr/local/bin/p
```

最后看看默认 python 版本：

```
[root@iZ25ix41uc3Z bin]# python --version
Python 2.7.4
```

OK，python2.7 安装成功！

安装之后发现 yum 用不了了，找到 /usr/bin/yum。编辑一下：

```
vi /usr/bin/yum
```

然后将第一行的 #!/usr/bin/python 改成 #!/usr/bin/python2.6 即可。

下面我们再安装个 pip，pip 安装 python 第三方模块非常方便。

先下载 setuptools：

```
wget --no-check-certificate  http://pypi.python.org/packages/source/s/setup
解压之后进入目录setuptools-0.6c11
安装python setup.py install
```

然后安装 pip，和 setuptools 过程类似：

```
wget --no-check-certificate https://github.com/pypa/pip/archive/1.5.5.tar.g
解压之后进入目录pip-1.5.5
安装python setup.py install
```

看看 pip 安装是否成功，执行：

```
pip list
```

如果提示 HTTPERROR：

则先安装 openssl：

```
yum install openssl openssl-devel -y
```

然后再重新安装 python，别的不用重新安装了。

```
[root@iZ25ix41uc3Z Python-2.7.4]# pip list
pip (1.5.4)
setuptools (27.3.0)
wsgiref (0.1.2)
```

最后我们就可以安装 numpy，scipy 等科学计算库了。

```
pip install numpy
pip install scipy
```

最后验证一下 numpy：

```
[root@iZ25ix41uc3Z ~]# python
Python 2.7.4 (default, Jun 25 2017, 13:13:33)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-17)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
>>> np.array([1,2,3])
array([1, 2, 3])
>>>
```

成功安装 numpy。

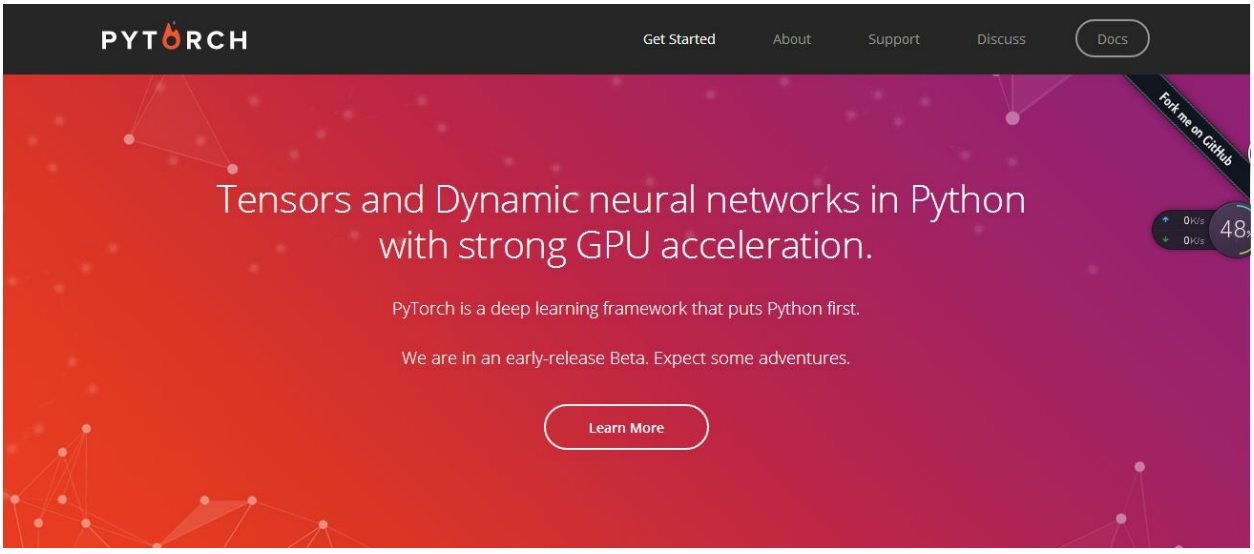
1.4 安装PyTorch

我们先试试看 pip 安装能不能成功？

输入命令 pip install pytorch，显示结果如下：

```
F:\ai>pip install pytorch
Collecting pytorch
  Downloading pytorch-0.1.2.tar.gz
Building wheels for collected packages: pytorch
  Running setup.py bdist_wheel for pytorch ... error
  Complete output from command e:\anaconda2\python.exe -u -c "import setuptools, tokenize;__file__='c:\\users\\houyijun\\appdata\\local\\temp\\pip-build-5nxyk5\\pytorch\\setup.py';f=getattr(tokenize, 'open', open)(__file__);code=f.read().replace('\r\n', '\n');f.close();exec(compile(code, __file__, 'exec'))" bdist_wheel -d c:\users\houyijun\appdata\local\temp\tmpfi_yzqpip-wheel- --python-tag cp27:
  Traceback (most recent call last):
    File "<string>", line 1, in <module>
    File "c:\users\houyijun\appdata\local\temp\pip-build-5nxyk5\pytorch\setup.py", line 17, in <module>
      raise Exception(message)
  Exception: You should install pytorch from http://pytorch.org

  ----------------------------------------

  Failed building wheel for pytorch
  Running setup.py clean for pytorch
Failed to build pytorch
Installing collected packages: pytorch
  Running setup.py install for pytorch ... error
    Complete output from command e:\anaconda2\python.exe -u -c "import setuptools, tokenize;__file__='c:\\users\\houyijun\\appdata\\local\\temp\\pip-build-5nxyk5\\pytorch\\setup.py';f=getattr(tokenize, 'open', open)(__file__);code=f.read().replace('\r\n', '\n');f.close();exec(compile(code, __file__, 'exec'))" install --record c:\users\houyijun\appdata\local\temp\pip-mw5h78-record\install-record.txt --single-version-externally-managed --compile:

    Traceback (most recent call last):
      File "<string>", line 1, in <module>
      File "c:\users\houyijun\appdata\local\temp\pip-build-5nxyk5\pytorch\setup.py", line 13, in <module>
        raise Exception(message)
    Exception: You should install pytorch from http://pytorch.org

    ----------------------------------------

Command "e:\anaconda2\python.exe -u -c "import setuptools, tokenize;__file__='c:\\users\\houyijun\\appdata\\local\\temp\\pip-build-5nxyk5\\pytorch\\setup.py';f=getattr(tokenize, 'open', open)(__file__);code=f.read().replace('\r\n', '\n');f.close();exec(compile(code, __file__, 'exec'))" install --record c:\users\houyijun\appdata\local\temp\pip-mw5h78-record\install-record.txt --single-version-externally-managed --compile" failed with error code 1 in c:\users\houyijun\appdata\local\temp\pip-build-5nxyk5\pytorch\
```

哦，原来不支持 pip 安装，并且提示到 pytorch.org 下载安装，同时，浏览器自动打开网址：http://pytorch.org/#pip-install-pytorch ，看看里面的安装说明是怎样的。



选择我们的环境对应的 whl 安装版本，我们用的是 Linux，Python2.7 环境。安装命令如下：

```
pip install  http://download.pytorch.org/whl/cu75/torch-0.1.12.post2-cp27-n
```

等待安装完成。

安装完成后，我们输入命令 python，进入 python 交互环境，写一段小小的 pytorch 程序验证一下是不是安装成功了，这段小代码调用 torch 的 ones 方法，看看能不能正常显示结果。

代码：

```
>>> import torch
>>> a=torch.ones(2,2)
>>> a
 1  1
 1  1
[torch.FloatTensor of size 2x2]
>>>
```

OK，安装成功了，下面我们来一步步学习 pytorch 吧。

2 变量

先看看 Tensor，pytorch 中的数据都是封装成 Tensor 来引用的，Tensor 实际上就类似于 numpy 中的数组，两者可以自由转换。

生成一个 (3,4) 维度的数组

```
import torch
x = torch.Tensor(3,4)
print("x Tensor: ",x)
```

可以看到 torch.Tensor() 方法生成制定维度的随机数。

下面看看 Variable 的基本操作，引用 Variable：

```
import torch
from torch.autograd import Variable
x=Variable(torch.Tensor(2,2))
print("x variable: ",x)
```

我们看到 Variable 比 Tensor 多出来第一行提示符”Variable containing:”，说明 Variable 不光包含了数据，还包含了其他东西，那么还包含什么东西呢？

默认 Variable 是有导数 grad 的，x.data 是数据，这里 x.data 就是 Tensor。x.grad 是计算过程中动态变化的导数。

```
print ("x.data: ",x.data, ",x.grad: ",x.grad)
```

此时 Variable 还未进行计算，因此 x.grad 为 None。

示例代码 chart1.py 中的 example1 方法。完整代码的执行结果截图如下：

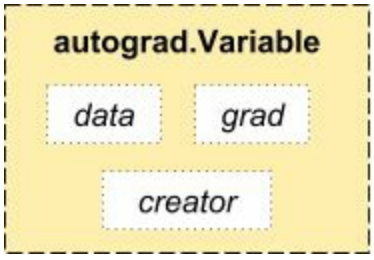
3 求导

理解 pytorch，首先要理解求导的概念。

数学上求导简单来说就是求取方程式相对于输入参数的变化率，也就是加速度。这部分理论基础参考高等数学的内容，上过大学的都学过，可能现在都忘了吧：）

求导的作用是用导数对神经网络的权重参数进行调整，注意这里提到了权重参数的概念，这是神经网络的范畴定义了，关于神经网络的基础知识本书不做介绍，读者最好先了解有关神经网络的基本概念再读此书。

Pytorch 中为求导提供了专门的包，包名叫 autograd。如果用 autograd.Variable 来定义参数，则 Variable 自动定义了两个变量，data 代表原始权重数据；而 grad 代表求导后的数据，也就是梯度。每次迭代过程就用这个 grad 对权重数据进行修正。



实践：

```
import torch
from torch.autograd import Variable
x = Variable(torch.ones(2, 2), requires_grad=True)
print(x)
```

输出：

```
1  1
```

```
 1  1
[torch.FloatTensor of size 2x2]
```

```
y=x+2
print(y)
```

输出：

```
3  3
 3  3
[torch.FloatTensor of size 2x2]
```

```
z = y * y * 3
out = z.mean()
print(z, out)
```

输出：

```
(Variable containing:
 27  27
 27  27
[torch.FloatTensor of size 2x2]
, Variable containing:
 27
[torch.FloatTensor of size 1]
) [torch.FloatTensor of size 1]
```

```
out.backward()
```

反向传播，也就是求导数的意思。输出 out 对 x 求导：

```
print(x.grad)
```

输出结果：

```
Variable containing:
 4.5000  4.5000
 4.5000  4.5000
[torch.FloatTensor of size 2x2]
```

4.5 是怎么算出来的呢，从前面的公式可以看出 z=(x+2)*(x+2)*3，它的导数是 3*(x+2)/2，当 x=1 时导数的值就是 3*(1+2)/2=4.5，和 pytorch 计算得出的结果是一致的。

权值更新方法：

```
weight = weight + learning_rate * gradient
```

```
learning_rate = 0.01
```

```
for f in model.parameters():
    f.data.sub_(f.grad.data * learning_rate)
```

learning_rate 是学习速率，多数时候就叫做 lr，是学习步长，用步长 * 导数就是每次权重修正的 delta 值，lr 越大表示学习的速度越快，相应的精度就会降低。

4 损失函数

损失函数，又叫目标函数，是编译一个神经网络模型必须的两个参数之一。另一个必不可少的参数是优化器。

损失函数是指用于计算标签值和预测值之间差异的函数，在机器学习过程中，有多种损失函数可供选择，典型的有距离向量，绝对值向量等。

上图是一个用来模拟线性方程自动学习的示意图。粗线是真实的线性方程，虚线是迭代过程的示意，w1 是第一次迭代的权重，w2 是第二次迭代的权重，w3 是第三次迭代的权重。随着迭代次数的增加，我们的目标是使得 wn 无限接近真实值。

那么怎么让 w 无限接近真实值呢？其实这就是损失函数和优化器的作用了。图中 1/2/3 这三个标签分别是 3 次迭代过程中预测 Y 值和真实 Y 值之间的差值（这里差值就是损失函数的意思了，当然了，实际应用中存在多种差值计算的公式），这里的差值示意图上是用绝对差来表示的，那么在多维空间时还有平方差，均方差等多种不同的距离计算公式，也就是损失函数了，这么一说是不是容易理解了呢？

这里示意的是一维度方程的情况，那么发挥一下想象力，扩展到多维度，是不是就是深度学习的本质了？

下面介绍几种常见的损失函数的计算方法，pytorch 中定义了很多类型的预定义损失函数，需要用到的时候再学习其公式也不迟。

我们先定义两个二维数组，然后用不同的损失函数计算其损失值。

```
import torch
from torch.autograd import Variable
import torch.nn as nn
import torch.nn.functional as F
sample = Variable(torch.ones(2,2))
a=torch.Tensor(2,2)
a[0,0]=0
```

```
a[0,1]=1
a[1,0]=2
a[1,1]=3
target = Variable (a)
```

sample 的值为：[[1,1],[1,1]]。

target 的值为：[[0,1],[2,3]]。

4.1 nn.L1Loss

L1Loss 计算方法很简单，取预测值和真实值的绝对误差的平均数即可。

```
criterion = nn.L1Loss()
loss = criterion(sample, target)
print(loss)
```

最后结果是：1。

它的计算逻辑是这样的：

先计算绝对差总和：|0-1|+|1-1|+|2-1|+|3-1|=4；

然后再平均：4/4=1。

4.2 nn.SmoothL1Loss

SmoothL1Loss 也叫作 Huber Loss，误差在 (-1,1) 上是平方损失，其他情况是 L1 损失。

```
criterion = nn.SmoothL1Loss()
loss = criterion(sample, target)
print(loss)
```

最后结果是：0.625。

4.3 nn.MSELoss

平方损失函数。其计算公式是预测值和真实值之间的平方和的平均数。

```
criterion = nn.MSELoss()
loss = criterion(sample, target)
print(loss)
```

最后结果是：1.5。

4.4 nn.BCELoss

二分类用的交叉熵，其计算公式较复杂，这里主要是有个概念即可，一般情况下不会用到。

```
criterion = nn.BCELoss()
loss = criterion(sample, target)
print(loss)
```

最后结果是：-13.8155。

4.5 nn.CrossEntropyLoss

交叉熵损失函数

该公式用的也较多，比如在图像分类神经网络模型中就常常用到该公式。

```
criterion = nn.CrossEntropyLoss()
loss = criterion(sample, target)
print(loss)
```

最后结果是：报错，看来不能直接这么用！

看文档我们知道 nn.CrossEntropyLoss 损失函数是用于图像识别验证的，对输入参数有各式要求，这里有这个概念就可以了，在图像识别一文中会有正确的使用方法。

4.6 nn.NLLLoss

负对数似然损失函数（Negative Log Likelihood）

在前面接上一个 LogSoftMax 层就等价于交叉熵损失了。注意这里的 xlabel 和上个交叉熵损失里的不一样，这里是经过 log 运算后的数值。这个损失函数一般也是用在图像识别模型上。

```
criterion = F.nll_loss()
```

```
loss = criterion(sample, target)
print(loss)
loss=F.nll_loss(sample,target)
```

最后结果是：报错，看来不能直接这么用！

Nn.NLLLoss 和 nn.CrossEntropyLoss 的功能是非常相似的！通常都是用在多分类模型中，实际应用中我们一般用 NLLLoss 比较多。

4.7 nn.NLLLoss2d

和上面类似，但是多了几个维度，一般用在图片上。

- input, (N, C, H, W)
- target, (N, H, W)

比如用全卷积网络做分类时，最后图片的每个点都会预测一个类别标签。

```
criterion = nn.NLLLoss2d()
loss = criterion(sample, target)
print(loss)
```

最后结果是：报错，看来不能直接这么用！

5 优化器Optim

优化器用通俗的话来说就是一种算法，是一种计算导数的算法。各种优化器的目的和发明它们的初衷其实就是能让用户选择一种适合自己场景的优化器。优化器的最主要的衡量指标就是优化曲线的平稳度，最好的优化器就是每一轮样本数据的优化都让权重参数匀速的接近目标值，而不是忽上忽下跳跃的变化。因此损失值的平稳下降对于一个深度学习模型来说是一个非常重要的衡量指标。

pytorch 的优化器都放在 torch.optim 包中。常见的优化器有：

SGD，Adam，Adadelta，Adagrad，Adamax 等。这几种优化器足够现实世界中使用了，如果需要定制特殊的优化器，pytorch 也提供了定制化的手段，不过这里我们就不去深究了，毕竟预留的优化器已经足够强大了。

这节就说说几种常见的优化器的用法。

5.1 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。SGD 优化器也是最常见的一种优化器，实现简单，容易理解。

用法：

```
optimizer = optim.SGD(model.parameters(), lr = 0.01, momentum=0.9)
```

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。
- parameters：Variable 参数，要优化的对象。

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，gt 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现实。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。

5.2 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 {x1,…,xm} 以及相关的输出 yi 。

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 。

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

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。

5.3 Adagrad

AdaGrad 可以自动变更学习速率，只是需要设定一个全局的学习速率 ϵ，但是这并非是实际学习速率，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的多：

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

其中 δ 是一个很小的常亮，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 {x1,…,xm} 以及相关的输出 yi 。

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 。

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

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 ：

仍然要设置一个变量 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。

5.4 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 xt 时，左右两边的单位不统一。

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 gt 随时间按照 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gt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 ∂f/∂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。

5.5 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 ρ1，二阶动量衰减系数 ρ2 。

其中几个取值一般为：δ=10-8,ρ1=0.9,ρ2=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 {x1,…,xm} 以及相关

的输出 yi 。

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 。

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

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<beta<1，通常很接近 1。
- epsilon：大于 0 的小浮点数，防止除0错误。

5.6 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

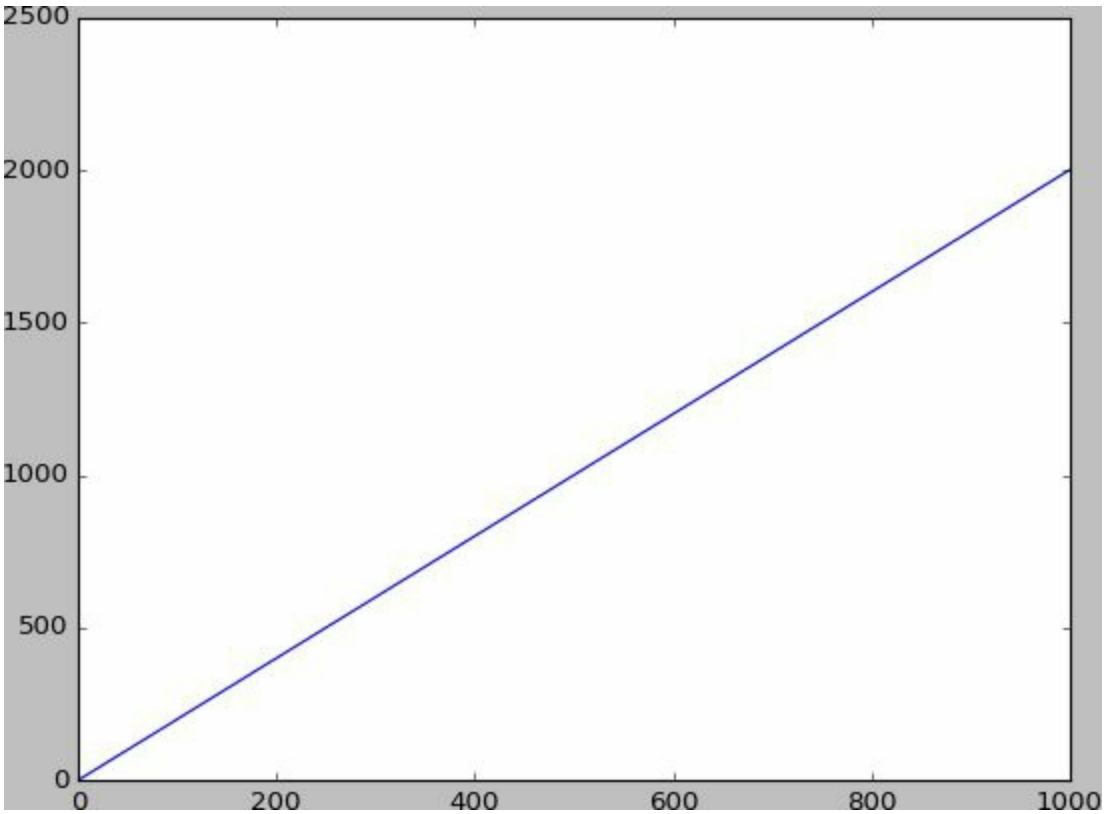
- lr：大于 0 的浮点数，学习率。
- beta_1/beta_2：浮点数， 0<beta<1，通常很接近 1。
- epsilon：大于 0 的小浮点数，防止除 0 错误。

6 线性回归

线性回归也叫 regression，它是一个比较简单的模拟线性方程式的模型。线性方程式我们应该都学过吧，就是类似这样：

Y=wX+b

其中 w 是系数，b 是位移，它是一条笔直的斜线。



那么我们假设给定一条模拟直线的点，每个点偏移这条直线很小的范围，我们要用到随机函数来模拟这个随机的偏移。

首先可以定义一个随机种子，随机种子基本不影响随机数的值，也可以不定义随机种子。随机数值在 0～1 之间。

例如：torch.manual_seed(1)

设置随机种子为 1。

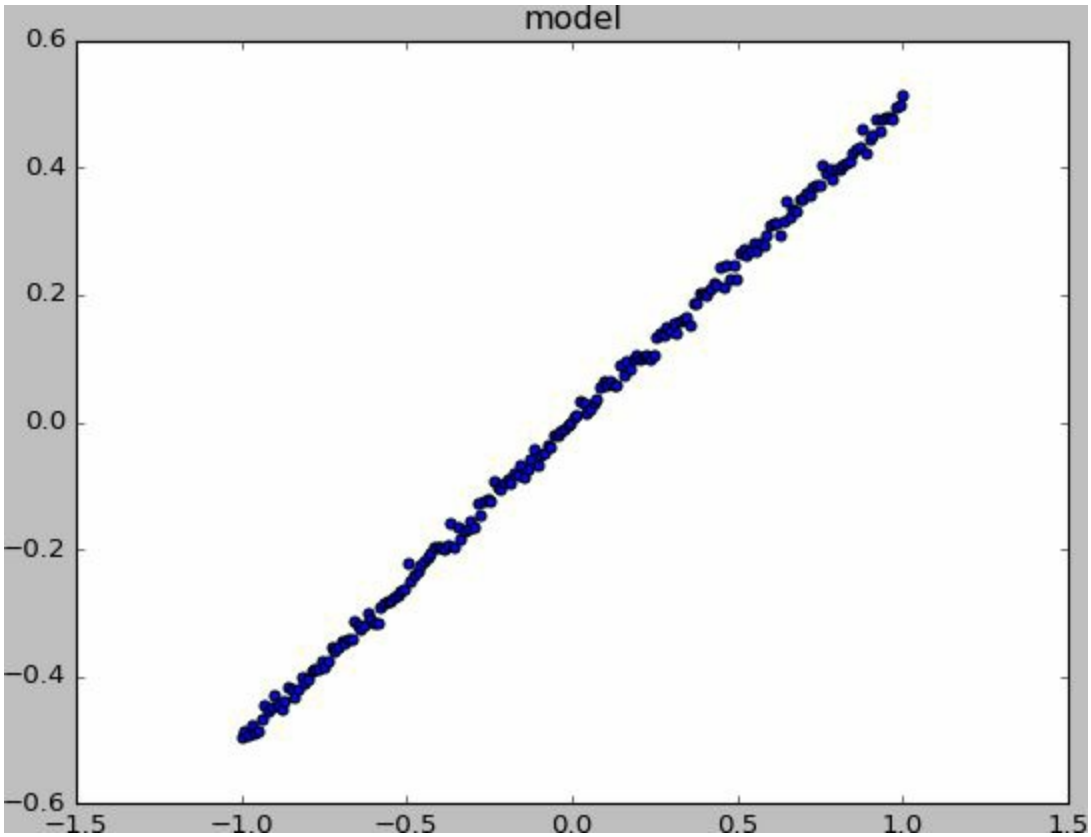
```
size=10
0.2*torch.rand(size)
```

这里我们不打算使用 pytorch 的随机函数，毕竟 numpy 中已经提供了随机函数，我们的数据是生成 200 个 X 和 Y，模拟参数 w 为 0.5。

代码：

```
import numpy as np
from numpy import random
import matplotlib.pyplot as plt
X = np.linspace(-1, 1, 200)
Y = 0.5 * X + 0.2* np.random.normal(0, 0.05, (200, ))
plt.scatter(X,Y)
plt.show()
#将X，Y转成200 batch大小，1维度的数据
X=Variable(torch.Tensor(X.reshape(200,1)))
Y=Variable(torch.Tensor(Y.reshape(200,1)))
```

图形：



注意：这里要将输入数据转换成 **(batch_size,dim)** 格式的数据，添加一个批次的维度。

现在的任务是给定这些散列点 (x,y) 对，模拟出这条直线来。这是一个简单的线性模型，我们先用一个简单的 1->1 的 Linear 层试试看。

示例代码：

```
# 神经网络结构
    model = torch.nn.Sequential(
        torch.nn.Linear(1, 1),
        )
    optimizer = torch.optim.SGD(model.parameters(), lr=0.5)
    loss_function = torch.nn.MSELoss()
```

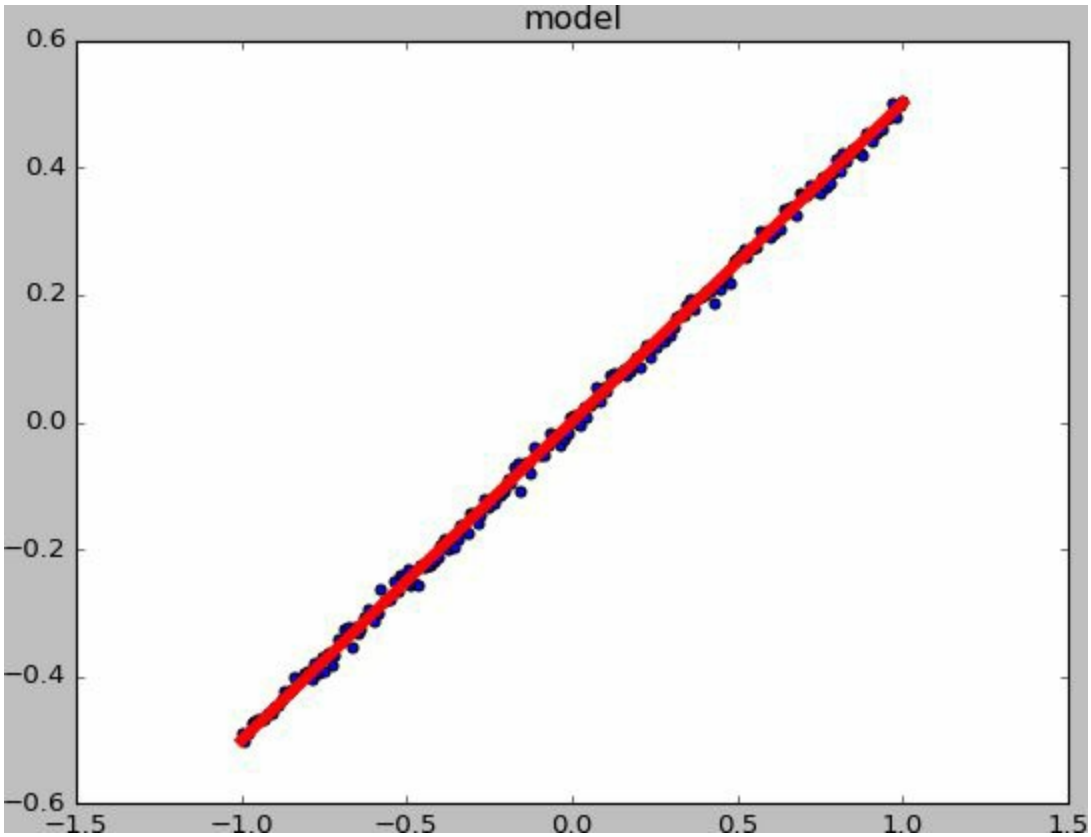
训练代码：

```
for i in range(300):
    prediction = model(X)
    loss = loss_function(prediction, Y)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

绘图部分代码：

```
plt.figure(1, figsize=(10, 3))
plt.subplot(131)
plt.title('model')
plt.scatter(X.data.numpy(), Y.data.numpy())
plt.plot(X.data.numpy(), prediction.data.numpy(), 'r-', lw=5)
plt.show()
```

最后的显示结果如图所示，红色是模拟出来的回归曲线：



笔者用的是阿里云主机，matplot 没能安装成功，因此将生成的 numpy 数组保存到文件，然后传到本地来显示图形的。numpy 数组的保存和导入代码：

```
np.save("pred.npy",prediction.data.numpy())
pred=  numpy.load("pred.npy")
```

最后附上完整代码：

```
import numpy as np
from numpy import random
import matplotlib.pyplot as plt
import torch
from torch.autograd import Variable
X = np.linspace(-1, 1, 200)
Y = 0.5 * X + 0.2* np.random.normal(0, 0.05, (200, ))
X=Variable(torch.Tensor(X.reshape(200,1)))
Y=Variable(torch.Tensor(Y.reshape(200,1)))
```

```
print(X)

model = torch.nn.Sequential(
    torch.nn.Linear(1, 1)
    )
optimizer = torch.optim.SGD(model.parameters(), lr=0.5)
loss_function = torch.nn.MSELoss()
for i in range(300):
       prediction = model(X)
       loss = loss_function(prediction, Y)
       print("loss:",loss)
       optimizer.zero_grad()
       loss.backward()
       optimizer.step()

print(prediction.data.numpy())
plt.figure(1, figsize=(10, 3))
plt.subplot(131)
plt.title('model')
plt.scatter(X.data.numpy(), Y.data.numpy())
plt.plot(X.data.numpy(), prediction.data.numpy(), 'r-', lw=5)
plt.show()
```

7 numpy矩阵的保存

```
import numpy as np
a=np.array(2)
np.save("nm.npy",a)
a = np.load("nm. npy ")
```

其中 np 是 import numpy as np 的 np。a 是对应的 numpy 数组，"nm. npy "是文件名称。

8 模型的保存和导入

每次定义和训练一个模型都要花费很长的时间，我们当然希望有一种方式可以将训练好的模型和参数保存下来，下一次使用的时候直接导入模型和参数，就跟一个已经训练好的神经网络模型一样。幸运的是，pytorch 提供了保存和导入方法。

保存模型：

```
# 保存整个神经网络的结构和模型参数
torch.save(mymodel, 'mymodel.pkl')
# 只保存神经网络的模型参数
torch.save(mymodel.state_dict(), 'mymodel_params.pkl')
```

导入模型：

```
mymodel = torch.load('mymodel.pkl')
```

9 卷积层

卷积层是用一个固定大小的矩形区去席卷原始数据，将原始数据分成一个个和卷积核大小相同的小块，然后将这些小块和卷积核相乘输出一个卷积值（注意：这里是一个单独的值，不再是矩阵了）。

卷积的本质就是用卷积核的参数来提取原始数据的特征，通过矩阵点乘的运算，提取出和卷积核特征一致的值，如果卷积层有多个卷积核，则神经网络会自动学习卷积核的参数值，使得每个卷积核代表一个特征。

这里我们拿最常用的 conv2d 和 conv1d 举例说明卷积过程的计算。

9.1 Conv2d

conv2d 是二维度卷积，对数据在宽度和高度两个维度上进行卷积。

函数定义：

```
torch.nn.functional.conv2d(input, weight, bias=None, stride=1, padding=0, d
```

参数说明：

- input：输入的Tensor数据，格式为 (batch,channels,H，W)，四维数组，第一维度是样本数量，第二维度是通道数或者记录数，三、四维度是高度和宽度。
- weight：卷积核权重，也就是卷积核本身，是一个四维度数组，(out_channels, in_channels/groups, kH, kW)。 Out_channels 是卷积核输出层的神经元个数，也就是这层有多少个卷积核；in_channels 是输入通道数，kH 和 kW 是卷积核的高度和宽度。
- bias：位移参数，可选项，一般不用管。

- stride：滑动窗口，默认为 1，指每次卷积对原数据滑动 1 个单元格。
- padding：是否对输入数据填充 0。Padding 可以将输入数据的区域改造成卷积核大小的整数倍，这样对不满足卷积核大小的部分数据就不会忽略了。通过 padding 参数指定填充区域的高度和宽度，默认 0（就是填充区域为0，不填充的意思）。
- dilation：卷积核之间的空格，默认1。
- groups：将输入数据分组，通常不用管这个参数，没有太大意义。

测试代码

```
import torch
from torch.autograd import Variable
import torch.nn as nn
import torch.nn.functional as F
print("conv2d sample")
a=torch.ones(4,4)
x = Variable(torch.Tensor(a))
x=x.view(1,1,4,4)
print("x variable:", x)
b=torch.ones(2,2)
b[0,0]=0.1
b[0,1]=0.2
b[1,0]=0.3
b[1,1]=0.4
weights = Variable(b)
weights=weights.view(1,1,2,2)
print ("weights:",weights)
y=F.conv2d(x, weights, padding=0)
print ("y:",y)
```

最终结果：

我们看看它是怎么计算的：

（1） 原始数据大小是1*1*4*4，1*1我们忽略掉，就是一个样本，每个样本一个通道的意思。4*4说明每个通道的数据是4*4大小的。而卷积核的大小是2*2。最后的卷积结果是3*3。

A. 第一步，卷积核与原始数据第一个数据做卷积乘法。图中示例部分的算法如下：

0.1*1+0.2*1+0.3*1+0.4*1=1.0。

B. 中间步骤，按顺序移动卷积核，并和目标区域做矩阵乘法。得到这一步的卷积值，作为结果矩阵的一个元素，图中示例部分的算法如下：

0.1*1+0.2*1+0.3*1+0.4*1=1.0

C. 最后一步，用卷积核卷积 input[2:4,2:4]，最后共 4 个元素。图中示例部分的算法和上面一样，最后的值也是 1。

因为原始数据都是 1，所有最后卷积出来的结果才是相同的，否则的话是不同的。最终卷积的结果就是：

9.2 Conv1d

conv1d 是一维卷积，它和 conv2d 的区别在于只对宽度进行卷积，对高度不卷积。

函数定义：

```
torch.nn.functional.conv1d(input, weight, bias=None, stride=1, padding=0, d
```

参数说明：

- input：输入的Tensor数据，格式为 (batch,channels,W)，三维数组，第一维度是样本数量，第二维度是通道数或者记录数，三维度是宽度。

- weight：卷积核权重，也就是卷积核本身。是一个三维数组，(out_channels, in_channels/groups, kW)。out_channels 是卷积核输出层的神经元个数，也就是这层有多少个卷积核；in_channels 是输入通道数；kW 是卷积核的宽度。

- bias：位移参数，可选项，一般也不用管。

- stride：滑动窗口，默认为 1，指每次卷积对原数据滑动 1 个单元格。

- padding：是否对输入数据填充 0。Padding 可以将输入数据的区域改造成是卷积核大小的整数倍，这样对不满足卷积核大小的部分数据就不会忽略了。通过 padding 参数指定填充区域的高度和宽度，默认 0（就是填充区域为0，不填充的意思）。

- dilation：卷积核之间的空格，默认 1。

- groups：将输入数据分组，通常不用管这个参数，没有太大意义。

测试代码

```
import torch
from torch.autograd import Variable
import torch.nn as nn
import torch.nn.functional as F
print("conv1d sample")
a=range(16)
x = Variable(torch.Tensor(a))
x=x.view(1,1,16)
print("x variable:", x)
b=torch.ones(3)
b[0]=0.1
b[1]=0.2
b[2]=0.3
weights = Variable(b)
weights=weights.view(1,1,3)
print ("weights:",weights)
y=F.conv1d(x, weights, padding=0)
```

```
print ("y:",y)
```

最终结果：

我们看看它是怎么计算的：

（1） 原始数据大小是 0-15 的一共 16 个数字，卷积核宽度是 3，向量是 [0.1,0.2,0.3]。

我们看第一个卷积是对 x[0:3] 共 3 个值 [0,1,2] 进行卷积，公式如下：

0*0.1+1*0.2+2*0.3=0.8

（2） 对第二个目标卷积，是 x[1:4] 共 3 个值 [1,2,3] 进行卷积，公式如下：

1*0.1+2*0.2+3*0.3=1.4

看到和计算结果完全一致！

该图就是conv1d的示意图，和conv2d的区别就是只对宽度卷积，不对高度卷积。最后结果的宽度是原始数据的宽度减去卷积核的宽度再加上 1，这里就是 14。

所以最终卷积之后的结果一共是 14 个数值，显示如下：

我们再看看输入数据有多个通道的情况：

核心代码：

```
print("conv1d sample")
a=range(16)
x = Variable(torch.Tensor(a))
x=x.view(1,2,8)
print("x variable:", x)
b=torch.ones(6)
b[0]=0.1
b[1]=0.2
b[2]=0.3
weights = Variable(b)
weights=weights.view(1,2,3)
print ("weights:",weights)
y=F.conv1d(x, weights, padding=0)
print ("y:",y)
```

我们看看返回结果第一个元素 27.8 是怎么计算出来的，这时候卷积核有 2 个通道：

[0.1,0.2,0.3] 和 [1,1,1]

第 1 个卷积对象也有 2 个通道：

[0,1,2] 和 [8,9,10]

结果是 2 个卷积核分别对应 2 个输入通道进行卷积然后求和。

卷积核对第 1 个卷积对象的卷积值：(0.1*0+0.2*1+0.3*2)+(1*8+1*9+1*10)=27.8

第2个卷积对象也有 2 个通道：

[1,2,3] 和 [9,10,11]

卷积核对第 2 个卷积对象的卷积值：(0.1*1+0.2*2+0.3*3)+(1*9+1*10+1*11)=31.4

和 pytorch 计算结果相同。

10 池化层

池化层比较容易理解，就是将多个元素用一个统计值来表示。为什么要池化？

比如对于一个图像来说，单个的像素其实不代表什么含义。

统计值可以取最大值，也可以取平均值，用不同的池化函数来表示。

10.1 max_pool2d

比如对于二维最大值池化来说，用 torch.nn.functional. F.max_pool2d 方法来操作。

比如：

```
import torch.nn.functional as F
from torch.autograd import Variable
print("conv2d sample")
a=range(20)
x = Variable(torch.Tensor(a))
x=x.view(1,1,4,5)
print("x variable:", x)
y=F.max_pool2d(x, kernel_size=2,stride=2)
print ("y:",y)
```

最后显示结果如下图：

x 是 4*5 的矩阵，表示高度 4，宽度 5，一个样本，每个样本一个通道。

x=x.view(1,1,4,5) 意思是将 x 矩阵转换成 (1,1,4,5) 的四维矩阵，第一个 1

是样本数，第二个1是通道数，第三个 4 和第四个 5 是高度和宽度。

b=F.max_pool2d(x, kernel_size=2,stride=2) 中的参数 2 表示池化的核大小是 2，也就是 (2,2)，表示核是一个行 2 列 2 的矩阵，每两行两列池化成一个数据。比如：

[[1,2],

[3,4]]

会被池化成最大的数，就是 4。

stride=2 表示滑动窗口为 2，第一个池化对象之后相隔 2 个元素距离，如果剩下的不够池化核的尺寸，则忽略掉不作池化处理。

第 1 个池化目标是 [[0,1],[5,6]]，因此最大池化结果是 6；

第 2 个池化目标是 [[2,3],[7,8]]，因此最大池化结果是 8。

max_pool2d 方法的说明如下：

```
torch.nn.functional.max_pool2d(input
, kernel_size
, stride=None
, padding=0
, dilation=1
, ceil_mode=False
, return_indices=False
)
```

那么具体的各个参数的含义说明如下：

- input：输入的 Tensor 数据，格式为 (channels,H，W)，三维数组，第一维度是通道数或者记录数，二、三维度是高度和宽度。

- kernel_size：池化区的大小，指定宽度和高度 (kh x kw)，如果只有一个值则表示宽度和高度相同。

- stride：滑动窗口，默认和 kernel_size 相同值，这样在池化的时候就不会重叠。如果设置的比 kernel_size 小，则池化时会重叠。它也是高度和宽度两个值。

- padding：是否对输入在左前端填充 0。池化时，如果剩余的区域不够池化区大小，则会丢弃掉。Padding 可以将输入数据的区域改造成是池化核的整数倍，这样就不会丢弃掉原始数据了。Padding 也是指定填充区域的高度和宽度，默认 0（就是填充区域为 0，不填充的意思）。

- ceil_mode：在计算输出 shape 大小时按照 ceil 还有 floor 计算，是数序函数（如ceil(4.5)=5;floor(4.5)=4）。

- count_include_pad：为 True 时，在求平均时会包含 0 填充区域的大小。这个参数只有在 avg_pool2d 并且 padding 参数不为 0 时才有意义。

10.2 avg_pool2d

那么同样的，avg_pool2d 和 max_pool2d 的计算原理是一样的！只不过 avg_pool2d 取的是平均值，而不是最大值而已。这里就不重复说明计算过程了。

10.3 max_pool1d

max_pool1d 和 max_pool2d 的区别和卷积操作类似，也是只对宽度进行池化。

先看看示例代码：

```
print("conv1d sample")
a=range(16)
x = Variable(torch.Tensor(a))
```

```
x=x.view(1,1,16)
print("x variable:", x)
y=F.max_pool1d(x, kernel_size=2,stride=2)
print ("y:",y)
```

输出结果：

max_pool1d 方法对输入参数的最后一个维度进行最大池化。

第一个池化目标 [0,1]，池化输出 1；

第二个池化目标 [2,3]，池化输出 3；

……

最后结果就是这样计算得来的。

同样，我们仿照卷积操作再看看多通道的池化示例。

代码：

```
print("conv1d sample")
a=range(16)
x = Variable(torch.Tensor(a))
x=x.view(1,2,8)
print("x variable:", x)
y=F.max_pool1d(x, kernel_size=2,stride=2)
print ("y:",y)
```

输出结果：

可以看到通道数保持不变！

11　Mnist手写数字图像识别

一个典型的神经网络的训练过程大致分为以下几个步骤：

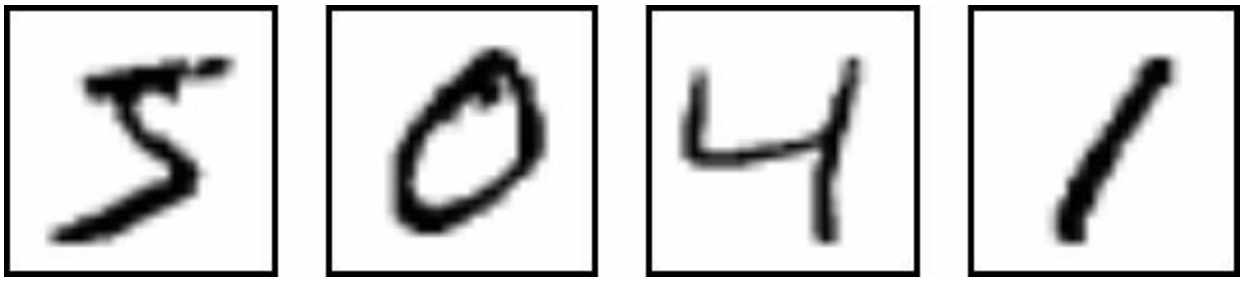
- 首先定义神经网络的结构，并且定义各层的权重参数的规模和初始值。
- 然后将输入数据分成多个批次输入神经网络。
- 将输入数据通过整个网络进行计算。
- 每次迭代根据计算结果和真实结果的差值计算损失。
- 根据损失对权重参数进行反向求导传播。
- 更新权重值，更新过程使用下面的公式：

  weight = weight + learning_rate * gradient

其中 weight 是上一次的权重值，learning_rate 是学习步长，gradient 是求导值。

下面我们还是以经典的图像识别卷积网络作为例子来学习 pytorch 的用法。选择一个较小的数据集进行训练，这里我们选择 mnist 手写识别数据集，该数据集是 0-9 个数字的训练数据和测试数据，训练数据 60000 张图片，测试数据 10000 张图片。每张图片是 28×28 像素大小的单通道黑白图片，这样读到内存数据组就是 28*28*1 大小。这里的 28*28 是宽度和高度的像素数量，1 是图像通道数据（如果是彩色图像就是 3 道路，内存数据为 28*28*3 大小）。

我们先看看样本图像的模样：



好，下一步我们构建一个简单的卷积神经网络模型，这个模型的输入是图片，输出是 0-9 的数字，这是一个典型的监督式分类神经网络。

11.1 加载数据

这里我们不打算下载原始的图像文件然后通过 opencv 等图像库读取到数组，而是直接下载中间数据。当然读者也可以下载原始图像文件从头开始装载数据，这样对整个模型会有更深刻的体会。

我们用 mnist 数据集作例子，下载方法有两种。

（1）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。注意调用 load_mnist 方法之前先解压上述四个文件。

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

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

显示结果：

```
D:\AI>python mnist_cnn.py
Rows: 60000, columns: 784
Rows: 10000, columns: 784
```

我们看到训练数据的行数是 60000，表示 60000 张图片，列是 784，表示 28*28*1=784 个像素，测试数据是 10000 张图片，在后面构建卷积模型时，要先将 784 个像素的列 reshape 成 28*28*1 维度。

例如：

```
image1 = X_train[1]
image1 =  image1.astype('float32')
image1 =  image1.reshape(28,28,1)
```

我们还可以将这些图像数组导出为 jpg 文件，比如下面代码：

```
import cv2
cv2.imwrite('1.jpg',image1)
```

当前目录下的 1.jpg 文件都是我们导出的图像文件了。

图像的显示：

```
import cv2
import numpy as np
img = cv2.imread("C:\lena.jpg")
cv2.imshow("lena",img)
cv2.waitKey(10000)
```

（2）用 numpy 读取 mnist.npz

可以直接从亚马逊下载文件：https://s3.amazonaws.com/img-datasets/mnist.npz

Mnist.npz 是一个 numpy 数组文件。如果下载的是 mnist.npz.gz 文件，则用 gunzip mnist.npz.gz 先解压成 mnist.npz，然后再处理。

也可以调用 keras 的方法来下载：

```
from keras.datasets import mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

通过 mnist.load_data() 下载的 mnist.npz 会放在当前用户的 .keras 目录中。路径名称：~/.keras/datasets/mnist.npz

然后调用 numpy 来加载 mnist.npz。

示例代码：

```
import numpy as np
class mnist_data(object):
    def load_npz(self,path):
        f = np.load(path)
        for i in f:
           print i
        x_train = f['trainInps']
        y_train = f['trainTargs']
        x_test = f['testInps']
        y_test = f['testTargs']
        f.close()
        return (x_train, y_train), (x_test, y_test)
a = mnist_data()
(x_train, y_train), (x_test, y_test) = a.load_npz('D:/AI/torch/data/mnist.n
print ("train rows:%d,test rows:%d"% (x_train.shape[0], x_test.shape[0]))
print("x_train shape",x_train.shape)
print("y_train shape",y_train.shape )
```

结果：

一共 60000 张训练图片，10000 张测试图片。训练图像的大小 784 个像素，也就是 1 通道的 28*28 的手写图片。标签是长度为 10 的 (0，1) 向量，表示 0-9 是个数字。例如数字 1 就表示为 [0,1,0,0,0,0,0,0,0,0]。

我们打印出第一张图片的截图和标签。

代码如下：

```
tt=x_train[1]
tt=tt.astype('float32')
image = tt.reshape(28,28,1)
cv2.imwrite("001.png",image)
print("tt label:",y_train[1])
```

标签为 [0,0,0,1,0,0,0,0,0,0]

11.2 定义卷积模型

这一步我们定义自己的卷积模型，对于 28*28 的数组，我们定义 (5,5) 的卷积核大小比较合适，卷积核的大小可根据图像大小灵活设置，比如 (3,3)，(5,5)，(9,9) 等。一般卷积核的大小是奇数。

输入数据首先连接 1 个卷积层加 1 个池化层，conv1 和 pool1，假设我们定义 conv1 的神经元数目为 10，卷积核大小 (5,5)，定义 pool1 的大小 (2,2)，意思将 2*2 区域共 4 个像素统计为 1 个像素，这样这层数据量减少 4 倍。这时候输出图像大小为 (28-5+1)/2=12。输出数据维度 (10,12,12)。

接着再来一次卷积池化，连接 1 个卷积层加 1 个池化层，conv2 和 pool2，假设我们定义 conv2 的神经元数目为 20，卷积核大小 (5,5)，定义pool2的大小 (2,2)，意思将 2*2 区域共 4 个像素统计为 1 个像素，这样这层数据量减少 4 倍。这时候输出图像大小为 (12-5+1)/2=4。输出数据维度 (20,4,4)。

然后接一个 dropout 层，设置 dropout=0.2，随机抛弃部分数据。

最后连续接两个全连接层，第一个全连接层 dense1 输入维度 20*4*4，输出 320，第二个全连接层 dense1 输入维度 60，输出 10。

模型定义的 pytorch 代码如下：

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(320, 60)
```

```
        self.fc2 = nn.Linear(60, 10)
    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x)
model = Net()
print(model)
```

注解：nn.Conv2d(1, 10, kernel_size=5) 是指将 1 通道的图像数据的输入卷积成 10 个神经元，折1 个通道跟 10 个神经元都建立连接，然后神经网络对每个连接计算出不同的权重值，某个神经元上的权重值对原图卷积来提取该神经元负责的特征。这个过程是神经网络自动计算得出的。

那么 nn.Conv2d(10, 20, kernel_size=5) 是指将 10 通道的图像数据的输入卷积成 20 个神经元，同样的，这 10 个通道会和 20 个神经元的每一个建立连接，那么 10 个通道如何卷积到 1 个通道呢？一般是取 10 个通道的平均值作为最后的结果。

执行该脚本，将卷积模型的结构打印出来：

```
[root@iZ25ix41uc3Z ai]# python mnist_torch.py
Net (
  (conv1): Conv2d(1, 10, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(10, 20, kernel_size=(5, 5), stride=(1, 1))
  (conv2_drop): Dropout2d (p=0.5)
  (fc1): Linear (320 -> 50)
  (fc2): Linear (50 -> 10)
)
```

11.3 开始训练

数据准备好，模型建立好，下面根据神经网络的三部曲，就是选择损失函数和梯度算法对模型进行训练了。

损失函数通俗讲就是计算模型的计算结果和真实结果之间差异性的函数，典型的如距离的平方和再开平方，对于图像分类来说，我们取的损失函数是：

```
F.log_softmax(x)
```

在神经网络的训练过程中，我们使用 loss.backward() 来反向传递修正，反向修正就是根据计算和真实结果的差值（就是损失）来反向逆传播修正各层的权重参数，修正之后的结果保存在 .grad 中，因此每轮迭代执行 loss.backward() 的时候要先对 .grad 清零。

```
model.zero_grad()
print('conv1.bias.grad before backward')
print(model.conv1.bias.grad)
loss.backward()
print('conv1.bias.grad after backward')
print(model.conv1.bias.grad)
```

优化算法一般是指迭代更新权重参数的梯度算法，这里我们选择随机梯度算法 SGD。

SGD 的算法如下：

```
weight = weight - learning_rate * gradient
```

可以用一段简单的 python 脚本模拟这个 SGD 的过程：

```
learning_rate = 0.01
for f in model.parameters():
```

```
    f.data.sub_(f.grad.data * learning_rate)
```

pytorch 中含有其他的优化算法，如 Nesterov-SGD, Adam, RMSProp 等。它们的用法和 SGD 基本类似，这里就不一一介绍了。

训练代码如下：

```
import torch.optim as optim
input = Variable(torch.randn(1, 1, 32, 32))
out = model(input)
print(out)

# create your optimizer
optimizer = optim.SGD(model.parameters(), lr=0.01)

# in your training loop:
optimizer.zero_grad()   # zero the gradient buffers
output = model(input)
loss = criterion(output, target)
loss.backward()
optimizer.step()    # Does the update
```

11.4 完整代码

最后的代码如下所示，这里我们做了一点修改，每次迭代完会将模型参数保存到文件，下次再次执行脚本会自动加载上次迭代后的数据。整个完整的代码如下：

```
'''
Trains a simple convnet on the MNIST dataset.
'''

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

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.autograd import Variable

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),
    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]))

batch_size = 100
num_classes = 10
epochs = 2

# input image dimensions
img_rows, img_cols = 28, 28

x_train= X_train
x_test=X_test

if 'channels_first' == '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')
num_samples=x_train.shape[0]
print("num_samples:",num_samples)

'''
build torch model
'''
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x)

model = Net()
if os.path.exists('mnist_torch.pkl'):
    model =  torch.load('mnist_torch.pkl')
print(model)


'''
trainning
'''
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
#loss=torch.nn.CrossEntropyLoss(size_average=True)
def train(epoch,x_train,y_train):
    num_batchs = num_samples/ batch_size
    model.train()
    for k in range(num_batchs):
        start,end = k*batch_size,(k+1)*batch_size
        data, target = Variable(x_train[start:end],requires_grad=False), Va
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
```

```
        if k % 10 == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, k * len(data), num_samples,
                100. * k / num_samples, loss.data[0]))
    torch.save(model, 'mnist_torch.pkl')


'''
evaludate
'''
def test(epoch):
    model.eval()
    test_loss = 0
    correct = 0
    if 2>1:
        data, target = Variable(x_test, volatile=True), Variable(y_test)
        output = model(data)
        test_loss += F.nll_loss(output, target).data[0]
        pred = output.data.max(1)[1] # get the index of the max log-probabi
        correct += pred.eq(target.data).cpu().sum()

    test_loss = test_loss
    test_loss /= len(x_test) # loss function already averages over batch si
    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.f
        test_loss, correct, len(x_test),
        100. * correct / len(x_test)))

x_train=torch.from_numpy(x_train).float()
x_test=torch.from_numpy(x_test).float()
y_train=torch.from_numpy(y_train).long()
y_test=torch.from_numpy(y_test).long()
for epoch in range(1,epochs):
    train(epoch,x_train,y_train)
    test(epoch)
```

跑一段时间后（根据你机器性能，笔者跑了几分钟而已），看看最终的效果如何。

11.5 验证结果

2 次迭代后：Test set: Average loss: 0.3419, Accuracy: 9140/10000 (91%)

3 次迭代后：Test set: Average loss: 0.2362, Accuracy: 9379/10000 (94%)

4 次迭代后：Test set: Average loss: 0.2210, Accuracy: 9460/10000 (95%)

5 次迭代后：Test set: Average loss: 0.1789, Accuracy: 9532/10000 (95%)

顺便说一句，pytorch 的速度比 keras 确实要快很多，每次迭代几乎 1 分钟内就完成了，确实很赞！

11.6 修改参数

我们把卷积层的神经元个数重新设置一下，第一层卷积加到 32 个，第二层卷积神经元加到 64 个。则新的模型为下面的组织：

```
{
  (conv1): Conv2d(1, 32, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1))
  (conv2_drop): Dropout2d (p=0.5)
  (fc1): Linear (1024 -> 100)
  (fc2): Linear (100 -> 10)
)
```

然后同样的步骤我们跑一遍看看结果如何，这次明显慢了很多，但看的出来准确度也提高了一些：

第 1 次迭代：Test set: Average loss: 0.3159, Accuracy: 9038/10000 (90%)

第 2 次迭代：Test set: Average loss: 0.1742, Accuracy: 9456/10000 (95%)

第 3 次迭代：Test set: Average loss: 0.1234, Accuracy: 9608/10000 (96%)

第 4 次迭代：Test set: Average loss: 0.1009, Accuracy: 9694/10000 (97%)

12 图像处理

12.1 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：

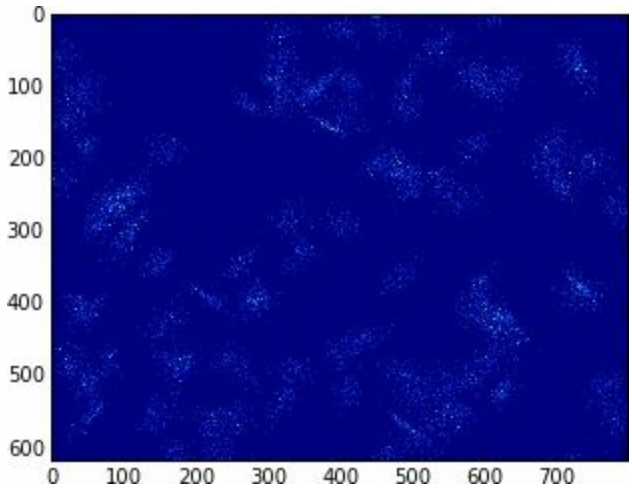
- OpenCV
- SimpleCV
- scikit-image
- Pillow

12.2 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

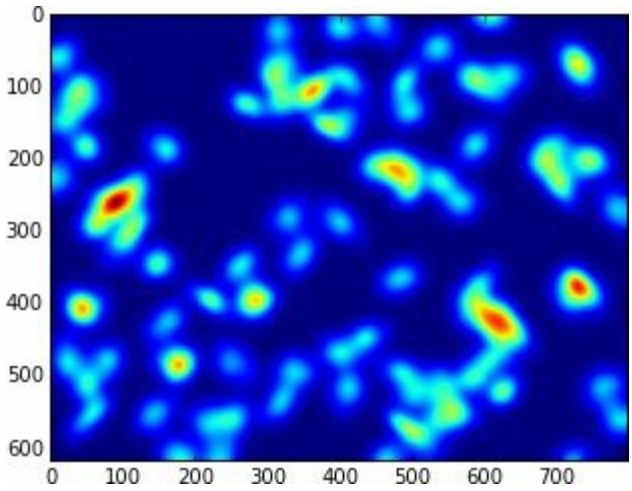
地图图片的网格中统计二维散列点的频度。由于散列点数量较少，histogram2d() 的结果并不能形成足够的热点信息：

```
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。

12.3 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用。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来实现加速。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：

其中 * 表示卷积操作；Gσ 是标准差为 σ 的二维高斯核，定义为：

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。

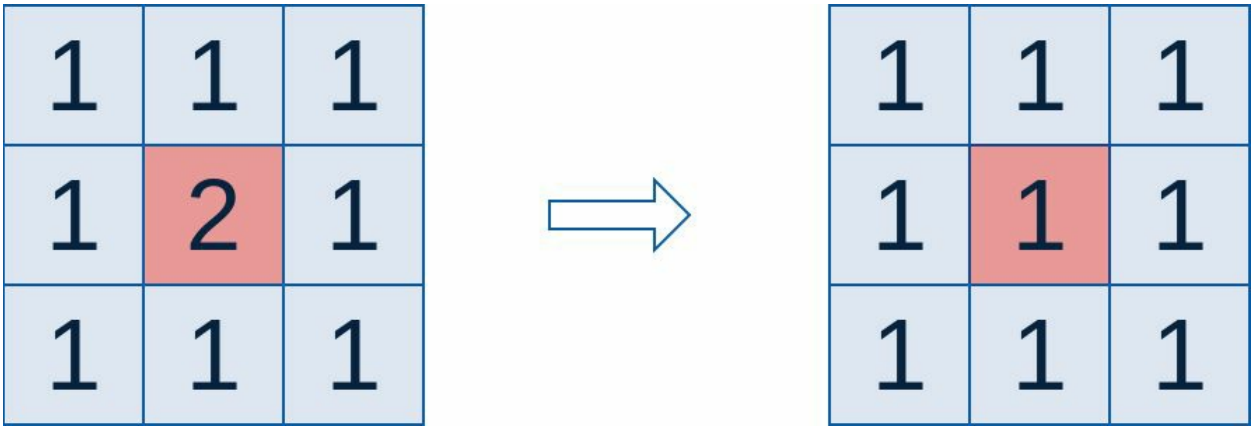
设：f(x)，g(x) 是 R1 上的两个可积函数，作积分：

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：

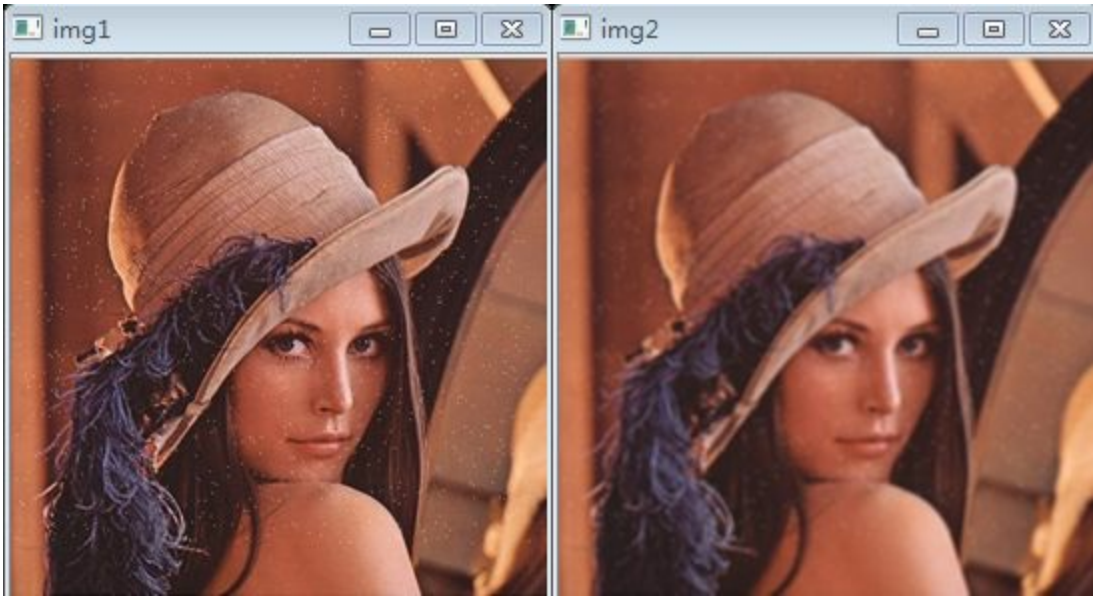
```
import cv2
#两个回调函数
def GaussianBlurSize(GaussianBlur_size):
    global KSIZE
    KSIZE = GaussianBlur_size * 2 +3
    print KSIZE, SIGMA
    dst = cv2.GaussianBlur(scr, (KSIZE,KSIZE), SIGMA, KSIZE)
    cv2.imshow(window_name,dst)
```

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）：



12.4 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。

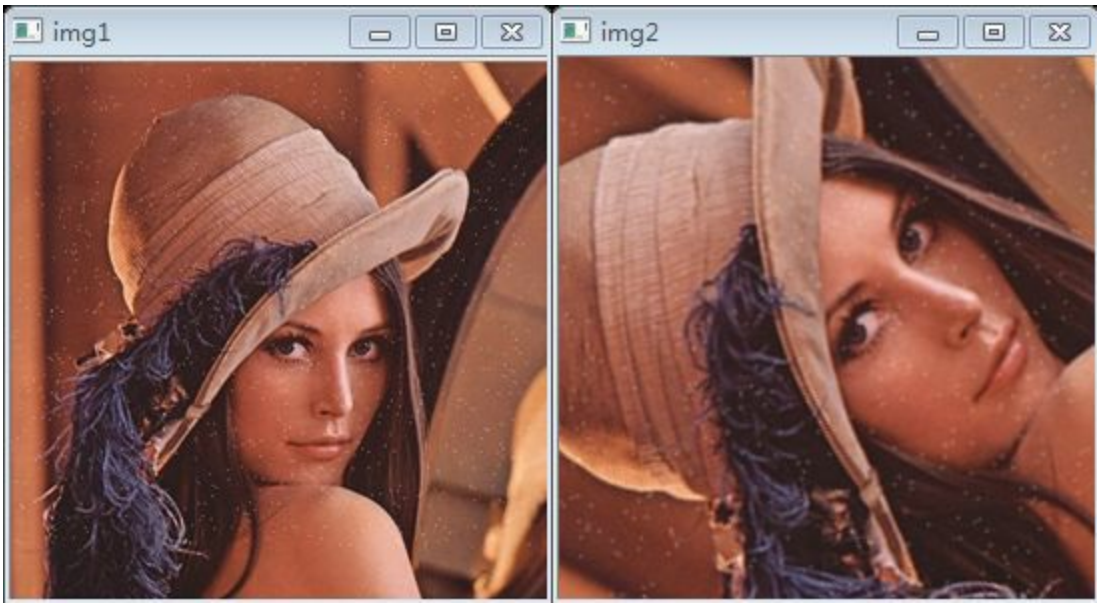
```
from __future__ import print_function
import os
import struct
import math
import numpy as np
import cv2
```

```
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))
    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()
```

通过以下这两个方法获得翻转后的图像矩阵：

```
rotateMat = cv2.getRotationMatrix2D((width/2, height/2), angle, scale)
 rotateImg = cv2.warpAffine(img, rotateMat, (width, height))
```

运行之后得出结果如下图所示：



12.5 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_APPRO
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_APPRO
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（abs（x1-x2），abs（y2-y1））==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。

12.6 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。

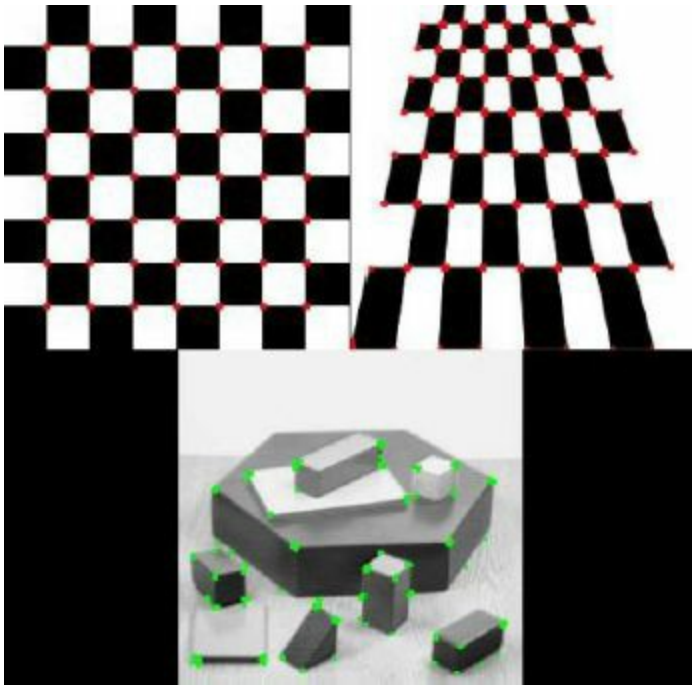
- 典型的角点检测算法：Harris 角点检测、CSS 角点检测。
- 好的角点检测算法的特点：1、检测出图像中“真实的”角点；2、准确的定位性能；3、很高的重复检测率（稳定性好）；4、具有对噪声的鲁棒性；5、具有较高的计算效率。

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：



12.7 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的灰度概念。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 轴是灰度级（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*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花直方图，这个时候我们可能只需要 1*(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。

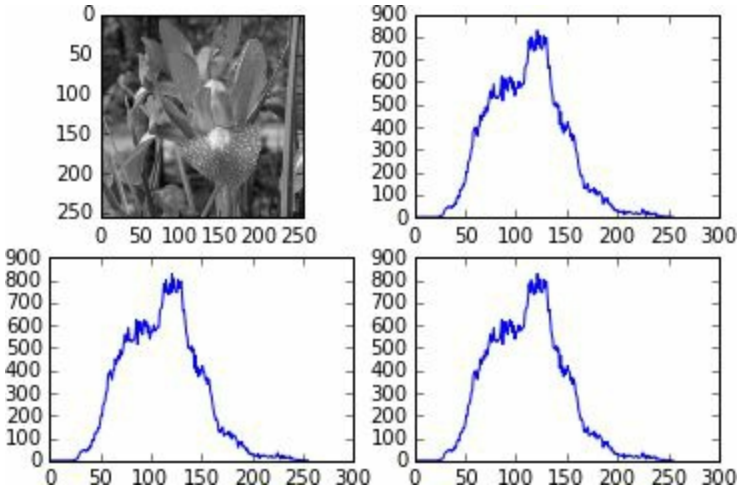
OpenCV 给我们提供的函数是 cv2.calcHist()，该函数有 5 个参数：

- 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，就得构造相应的炎掩膜来计算。
- 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()（用于以为统计直方图，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。

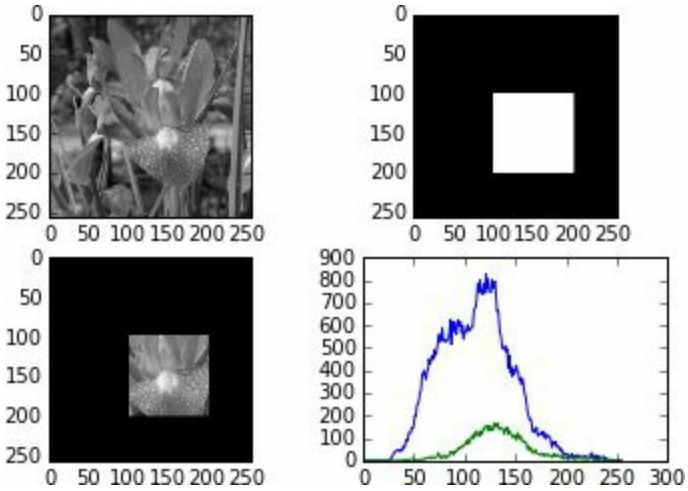
对于直方图的显示也是比较简单的，直接 plt.plot() 就可以。一个实例如下：

```
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)
```



12.8 视频抽取图片

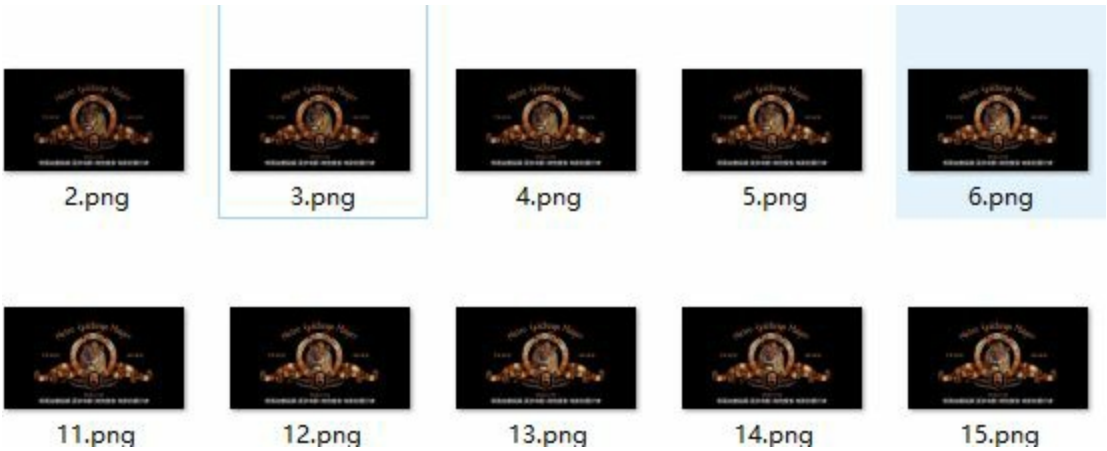
说完了图像识别，接下来看看如何给视频打标签。

视频其实就是一张张的图片，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类别。

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。

12.9 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。

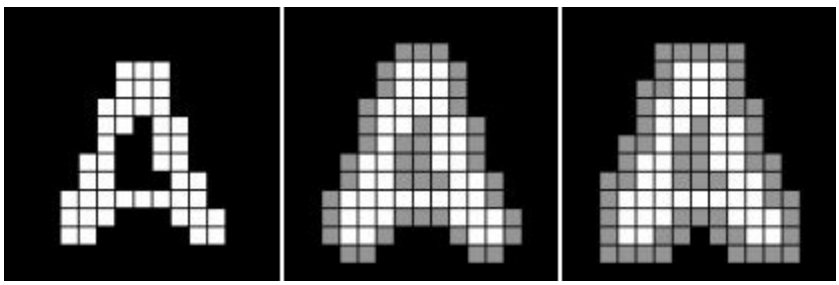
```
import numpy as np
def expand_image(img, value, out=None, size = 10):
    if out is None:
        w, h = img.shape
        out = np.zeros((w*size, h*size),dtype=np.uint8)
    tmp = np.repeat(np.repeat(img,size,0),size,1)
    out[:,:] = np.where(tmp, value, out)
    out[::size,:] = 0
    out[:,::size] = 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)
```

12.9.1 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）。

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：

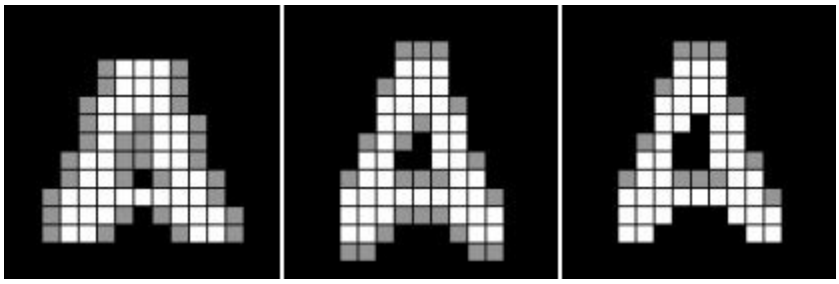
```
binary_dilation(a)
```

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]])
```

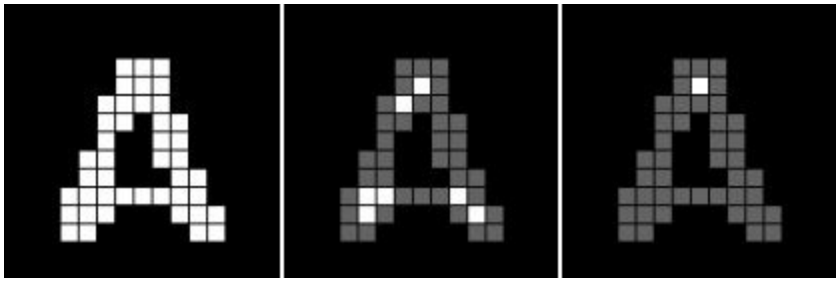
通过设置不同的结构元素，能够制作出各种不同的效果，下面显示了三个种不同结构元素的膨胀效果。图中的结构元素分别为：

```
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)
```



12.9.2 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：

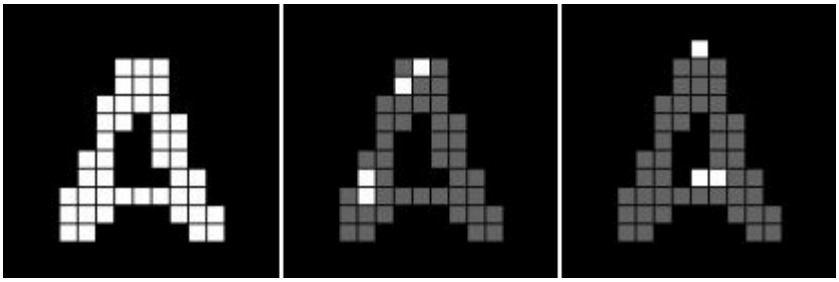
```
binary_hit_or_miss(input, structure1=None, structure2=None, ...)
```

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：

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”。

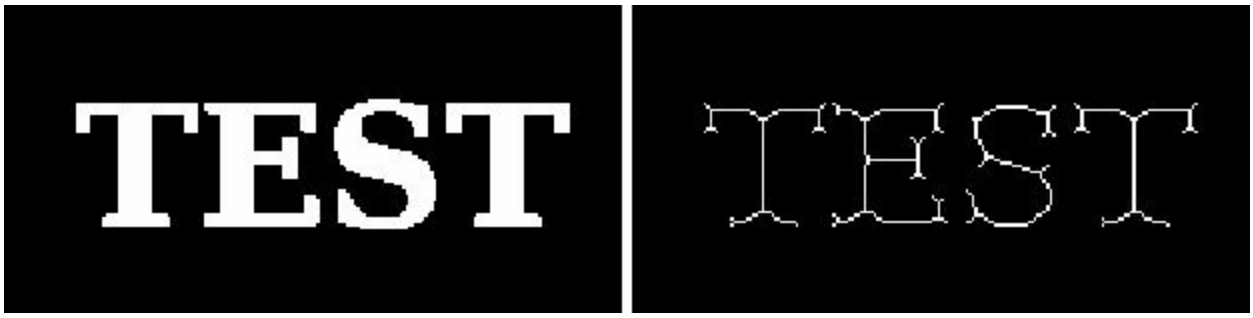
与右图对应的结构元素如下。通过它可以找到“下方三个像素为白色、左上像素为黑色的黑色像素”。

```
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。

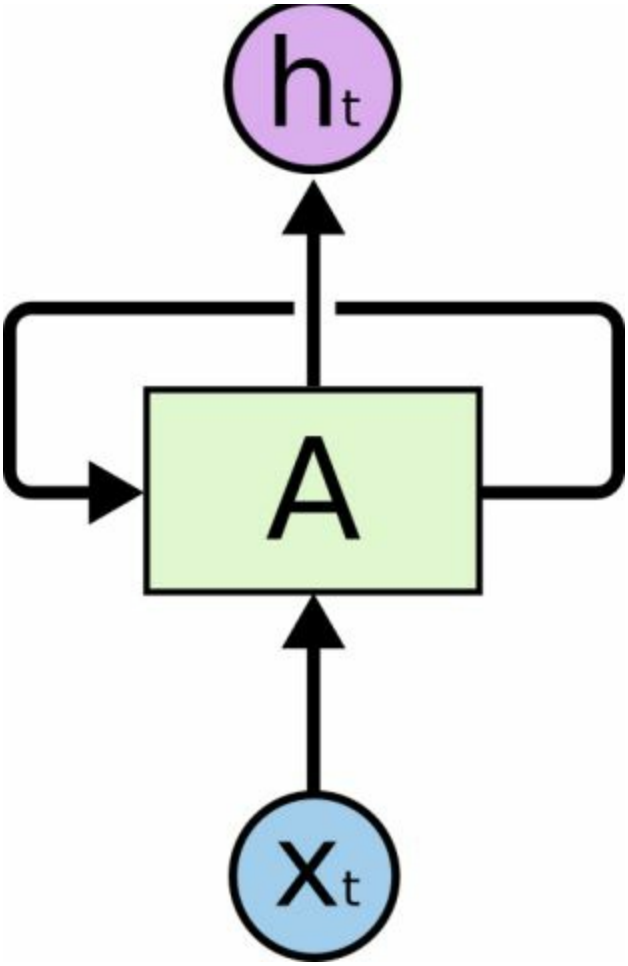
依次使用这些结构元素进行 Hit 和 Miss 运算，并从图像中删除运算所得到的白色像素，其效果就是依次从 8 个方向删除图像的边缘上的像素。重复运算直到没有像素可删除为止。

13　RNN和LSTM原理

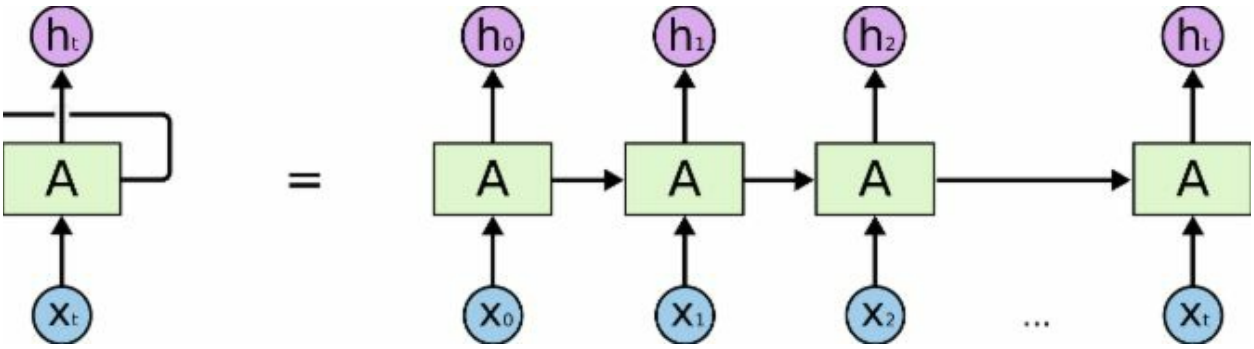
人类并不是每时每刻都从一片空白的大脑开始他们的思考。在你阅读这篇文章时候，你都是基于自己已经拥有的对先前所见词的理解来推断当前词的真实含义。我们不会将所有的东西都全部丢弃，然后用空白的大脑进行思考。我们的思想拥有持久性。

传统的神经网络并不能做到这点，看起来也像是一种巨大的弊端。例如，假设你希望对电影中的每个时间点的时间类型进行分类。传统的神经网络应该很难来处理这个问题——使用电影中先前的事件推断后续的事件。

RNN 解决了这个问题。RNN 是包含循环的网络，允许信息的持久化。



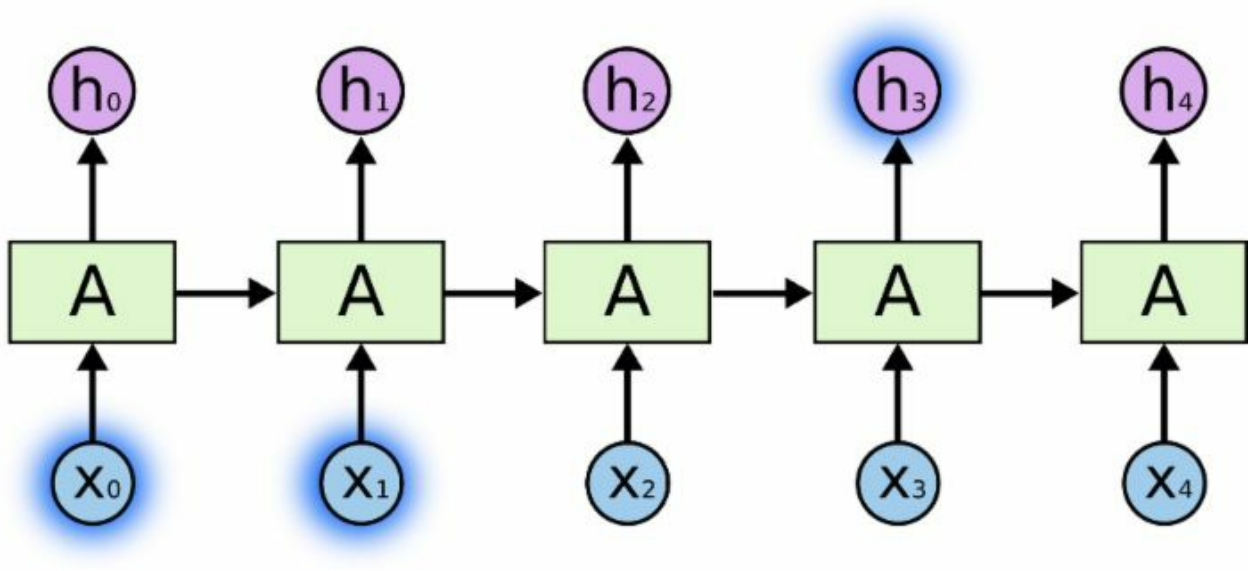
在上面的示例图中，神经网络的模块 A，正在读取某个输入 x_i，并输出一个值 h_i。循环可以使得信息可以从当前步传递到下一步。这些循环使得 RNN 看起来非常神秘。然而，如果你仔细想想，这样也不比一个正常的神经网络难于理解。RNN 可以被看做是同一神经网络的多次赋值，每个神经网络模块会把消息传递给下一个。所以，如果我们将这个循环展开：



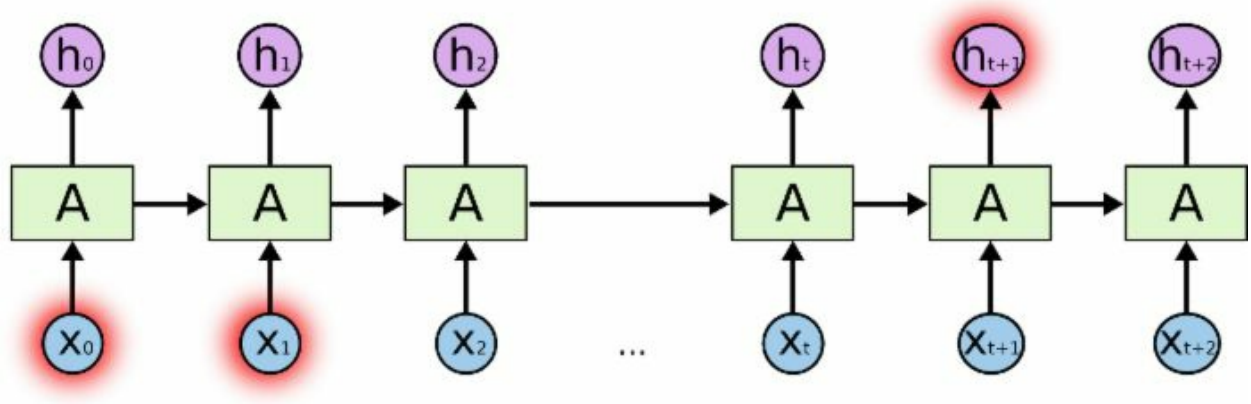
链式的特征揭示了 RNN 本质上是与序列和列表相关的。他们是对于这类数据的最自然的神经网络架构。在过去几年中，应用 RNN 在语音识别，语言建模，翻译，图片描述等问题上已经取得一定成功。而这些成功应用的关键之处就是 LSTM 的使用，这是一种特别的 RNN，比标准的 RNN 在很多的任务上都表现得更好。几乎所有的令人振奋的关于 RNN 的结果都是通过 LSTM 达到的。

13.1 长期依赖问题

RNN 的关键点之一就是他们可以用来连接先前的信息到当前的任务上，例如使用过去的视频段来推测对当前段的理解。如果 RNN 可以做到这个，他们就变得非常有用。但是真的可以么？答案是，还有很多依赖因素。有时候，我们仅仅需要知道先前的信息来执行当前的任务。例如，我们有一个语言模型用来基于先前的词来预测下一个词。如果我们试着预测 “the clouds are in the sky” 最后的词，我们并不需要任何其他的上下文 —— 因此下一个词很显然就应该是 sky。在这样的场景中，相关的信息和预测的词位置之间的间隔是非常小的，RNN 可以学会使用先前的信息。



但是同样会有一些更加复杂的场景。假设我们试着去预测“I grew up in France... I speak fluent French”最后的词。当前的信息建议下一个词可能是一种语言的名字，但是如果我们需要弄清楚是什么语言，我们是需要先前提到的离当前位置很远的 France 的上下文的。这说明相关信息和当前预测位置之间的间隔就肯定变得相当的大。不幸的是，在这个间隔不断增大时，RNN 会丧失学习到连接如此远的信息的能力。



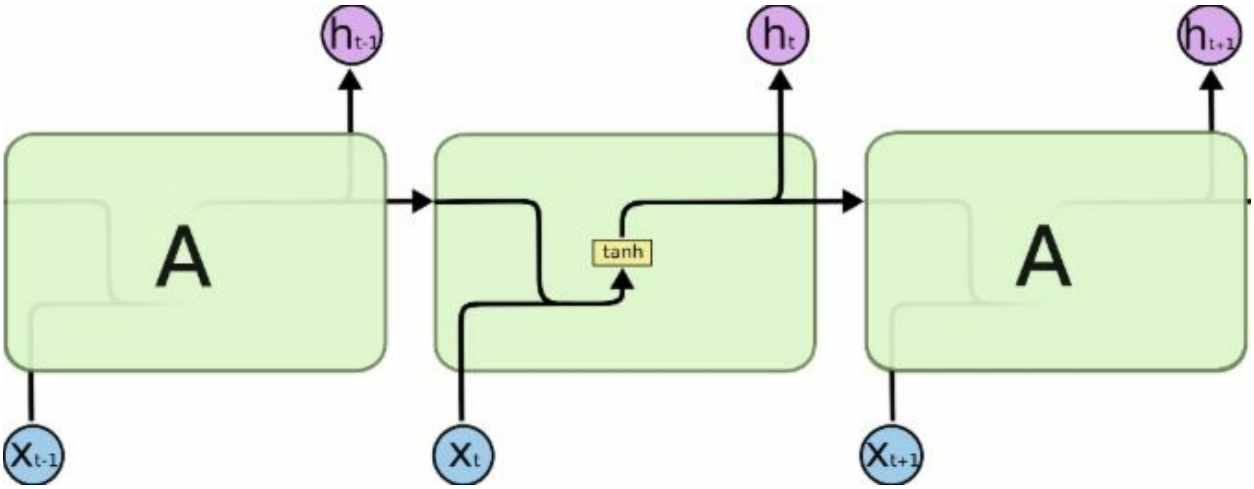
在理论上，RNN 绝对可以处理这样的 长期依赖 问题。人们可以仔细挑选参数来解决这类问题中的最初级形式，但在实践中，RNN 肯定不能够成功学习到这些知识。然而，幸运的是，LSTM 并没有这个问题！

13.2 LSTM 网络

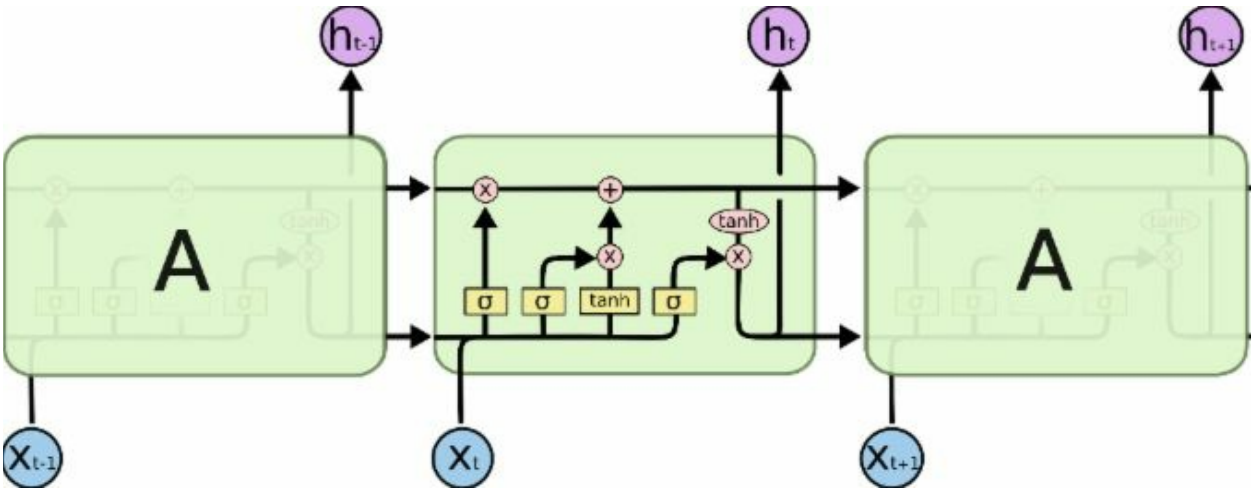
Long Short Term 网络—— 一般叫做 LSTM ——是一种 RNN 特殊的类型，可以学习长期依赖信息。LSTM 由 Hochreiter & Schmidhuber (1997) 提出，并在近期被 Alex Graves 进行了改良和推广。在很多问题，LSTM 都取得相当巨大的成功，并得到了广泛的使用。

LSTM 通过刻意的设计来避免长期依赖问题。记住长期的信息在实践中是LSTM的默认行为，而非需要付出很大代价才能获得的能力！

所有 RNN 都具有一种重复神经网络模块的链式的形式。在标准的RNN中，这个重复的模块只有一个非常简单的结构，例如一个tanh层。

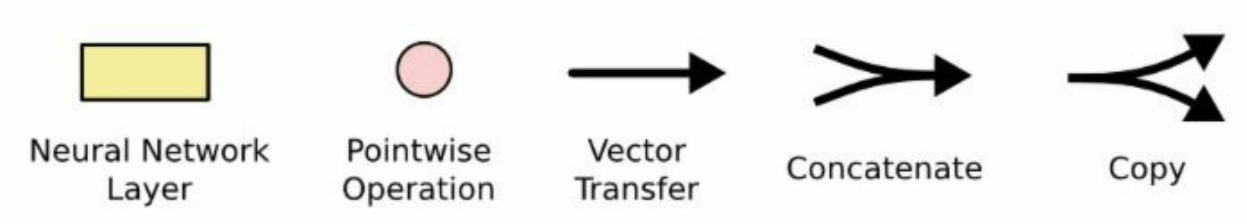


LSTM 同样是这样的结构，但是重复的模块拥有一个不同的结构。不同于单一神经网络层，这里有四个，以一种非常特殊的方式进行交互。



不必担心这里的细节。我们会一步一步地剖析LSTM解析图。现在，我

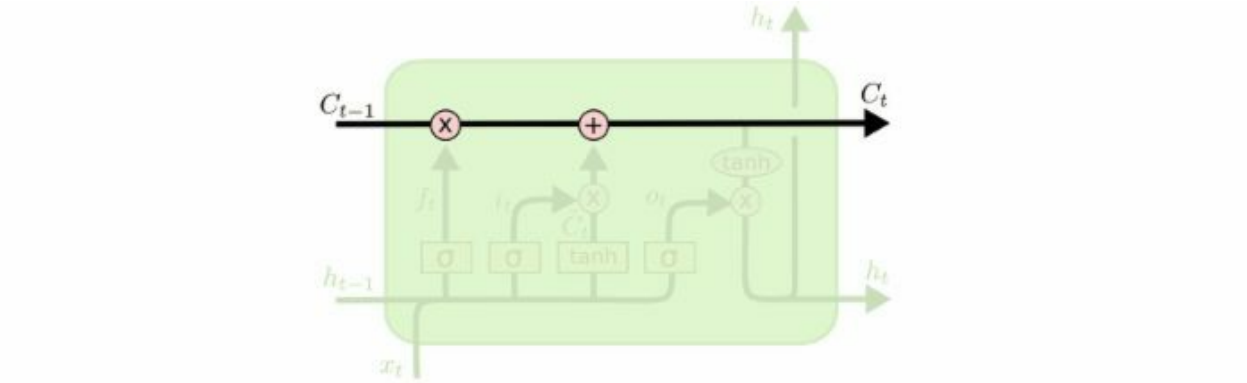
们先来熟悉一下图中使用的各种元素的图标。



在上面的图例中，每一条黑线传输着一整个向量，从一个节点的输出到其他节点的输入。粉色的圈代表 pointwise 的操作，诸如向量的和，而黄色的矩阵就是学习到的神经网络层。合在一起的线表示向量的连接，分开的线表示内容被复制，然后分发到不同的位置。

13.3 LSTM 的核心思想

LSTM 的关键就是细胞状态，水平线在图上方贯穿运行。细胞状态类似于传送带。直接在整个链上运行，只有一些少量的线性交互。信息在上面流传保持不变会很容易。



LSTM 有通过精心设计的称作为“门”的结构来去除或者增加信息到细胞状态的能力。门是一种让信息选择式通过的方法。他们包含一个 sigmoid 神经网络层和一个 pointwise 乘法操作。

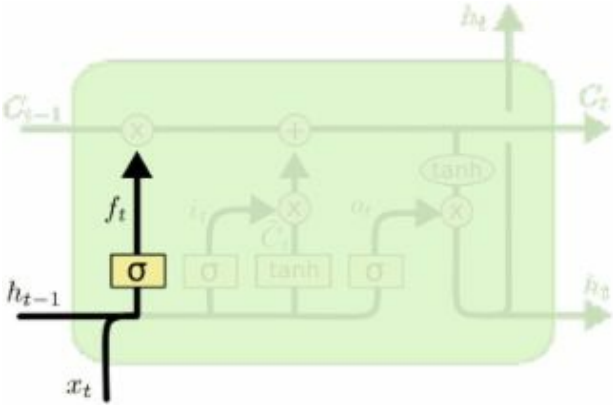
Sigmoid 层输出 0 到 1 之间的数值，描述每个部分有多少量可以通过。0 代表“不许任何量通过”，1 就指“允许任意量通过”!

LSTM 拥有三个门，来保护和控制细胞状态。

13.4 逐步理解 LSTM

在 LSTM 中的第一步是决定我们会从细胞状态中丢弃什么信息。这个决定通过一个称为忘记门层 完成。该门会读取 h_{t-1} 和 x_t，输出一个在 0 到 1 之间的数值给每个在细胞状态 C_{t-1} 中的数字。1 表示“完全保留”，0 表示“完全舍弃”。

让我们回到语言模型的例子中来基于已经看到的预测下一个词。在这个问题中，细胞状态可能包含当前主语 的类别，因此正确的代词 可以被选择出来。当我们看到新的代词 ，我们希望忘记旧的代词。

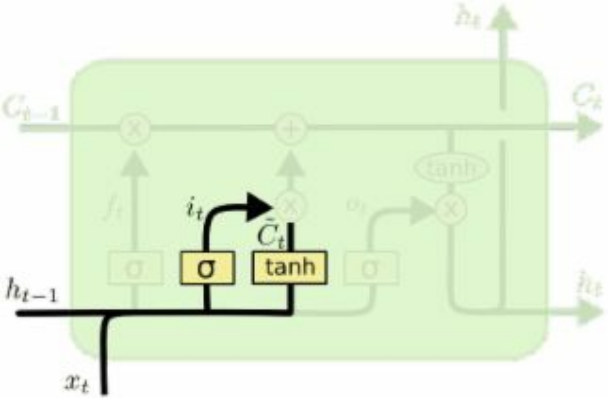


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

下一步是确定什么样的新信息被存放在细胞状态中。这里包含两个部分。第一，sigmoid 层称 “输入门层” 决定我们将要更新什么值。然后，

一个 tanh 层创建一个新的候选值向量——\tilde{C}_t 会被加入到状态中。下一步，我们会讲这两个信息来产生对状态的更新。

在我们语言模型的例子中，我们希望增加新的代词的类别到细胞状态中，来替代旧的需要忘记的代词。

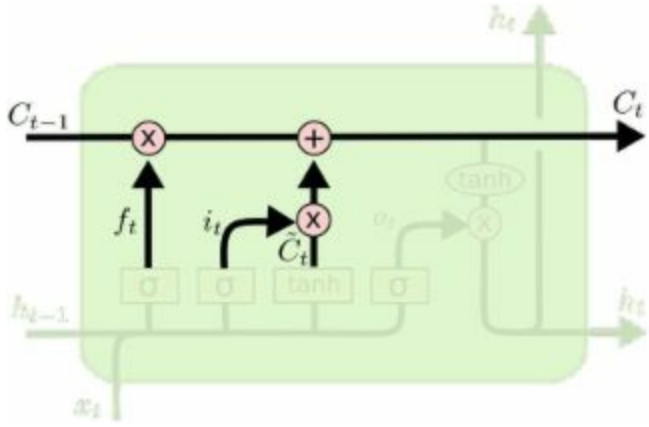


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \sigma\left(W_C \cdot [h_{t-1}, x_t] \; + \; b_C\right)$$

现在是更新旧细胞状态的时间了，C_{t-1} 更新为 C_t。前面的步骤已经决定了将会做什么，我们现在就是实际去完成。

我们把旧状态与 f_t 相乘，丢弃掉我们确定需要丢弃的信息。接着加上 i_t * \tilde{C}_t。这就是新的候选值，根据我们决定更新每个状态的程度进行变化。

在语言模型的例子中，这就是我们实际根据前面确定的目标，丢弃旧代词的类别信息并添加新的信息的地方。

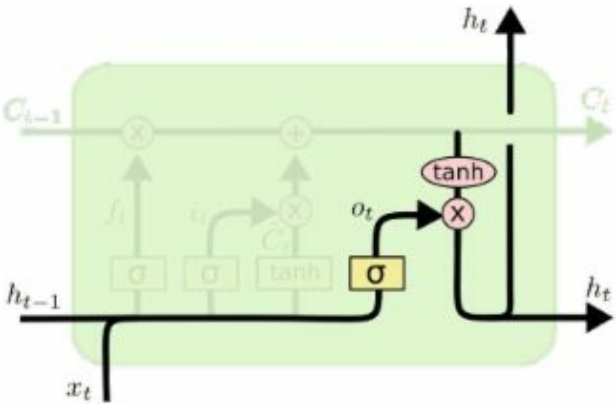


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

最终，我们需要确定输出什么值。这个输出将会基于我们的细胞状态，但是也是一个过滤后的版本。首先，我们运行一个 sigmoid 层来确定细

胞状态的哪个部分将输出出去。接着，我们把细胞状态通过 tanh 进行处理（得到一个在 -1 到 1 之间的值）并将它和 sigmoid 门的输出相乘，最终我们仅仅会输出我们确定输出的那部分。

在语言模型的例子中，因为他就看到了一个 代词 ，可能需要输出与一个动词 相关的信息。例如，可能输出是否代词是单数还是负数，这样如果是动词的话，我们也知道动词需要进行的词形变化。

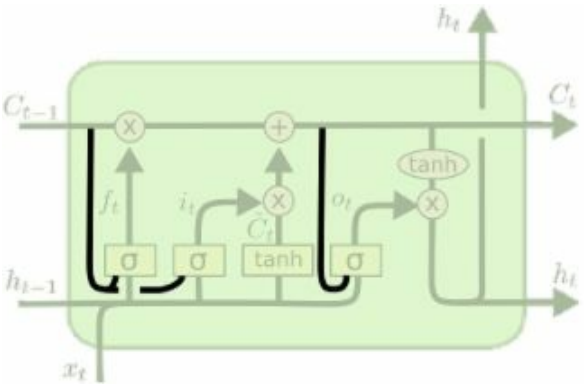


$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

13.5 LSTM 的变体

我们到目前为止都还在介绍正常的 LSTM。但是不是所有的 LSTM 都长成一个样子的。实际上，几乎所有包含 LSTM 的论文都采用了微小的变体。差异非常小，但是也值得拿出来讲一下。

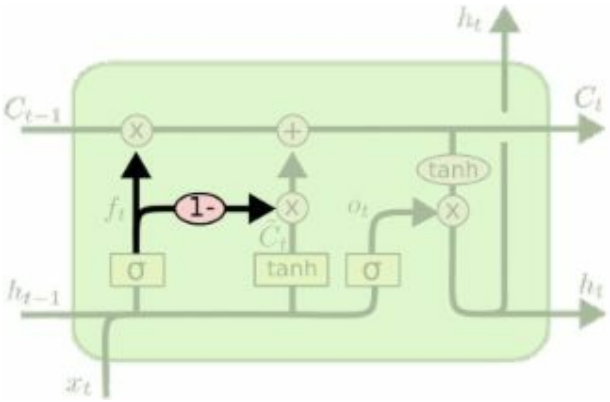
其中一个流形的 LSTM 变体，就是由 Gers & Schmidhuber (2000) 提出的，增加了 “peephole connection”。是说，我们让门层也会接受细胞状态的输入。



$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] + b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] + b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t] + b_o\right)$$

上面的图例中，我们增加了 peephole 到每个门上，但是许多论文会加入部分的 peephole 而非所有都加。

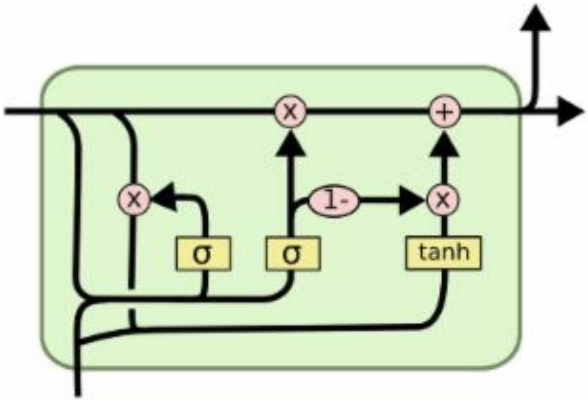
另一个变体是通过使用 coupled 忘记和输入门。不同于之前是分开确定什么忘记和需要添加什么新的信息，这里是一同做出决定。我们仅仅会当我们将要输入在当前位置时忘记。我们仅仅输入新的值到那些我们已经忘记旧的信息的那些状态 。



$$C_t = f_t * C_{t-1} + (1 - f_t) * \tilde{C}_t$$

coupled 忘记门和输入门

另一个改动较大的变体是 Gated Recurrent Unit (GRU)，这是由 Cho, et al. (2014) 提出。它将忘记门和输入门合成了一个单一的更新门 。同样还混合了细胞状态和隐藏状态，和其他一些改动。最终的模型比标准的 LSTM 模型要简单，也是非常流行的变体。



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

GRU

这里只是部分流行的 LSTM 变体。当然还有很多其他的，如 Yao, et al. (2015) 提出的 Depth Gated RNN。还有用一些完全不同的观点来解决长

期依赖的问题，如 Koutnik, et al. (2014) 提出的 Clockwork RNN。

14 PyTorch中的LSTM

LSTM 是深度学习中比较难以理解的模型了。我们还是以实践为主，主要学习怎么在 pytorch 中使用 LSTM，现阶段可以不必关注 LSTM 的内部实现细节，就好像我们天天在使用电脑，但也不是非要去搞清楚操作系统是怎么编写出来的，一样的道理，专业的事情交给更专业的人去做。

pytorch 中使用 nn.LSTM 类来搭建基于序列的循环神经网络。本节详细介绍 LSTM 编程接口的正确打开姿势。

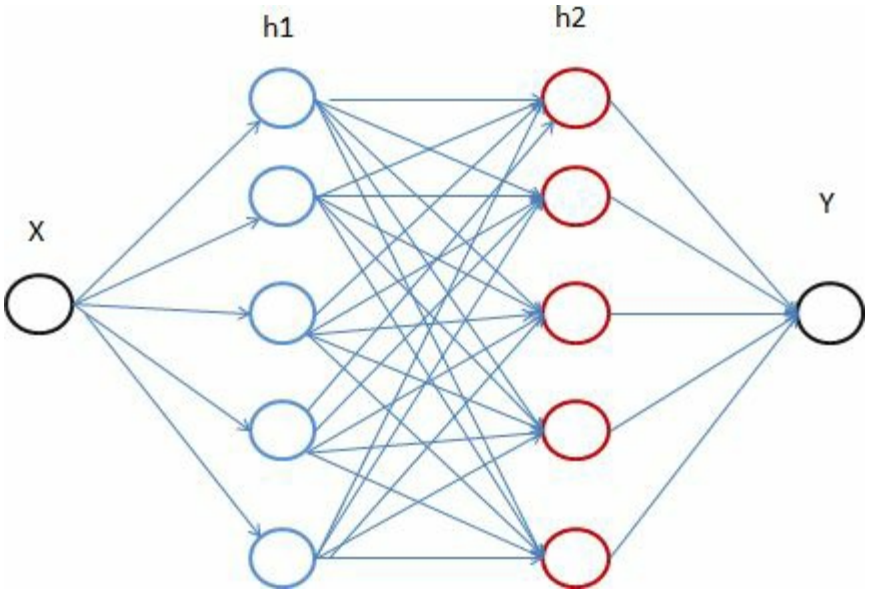
类的全称为 torch.nn.LSTM，它的构造函数有以下几个参数：

- input_size：输入数据X的特征值的数目。
- hidden_size：隐藏层的神经元数量，也就是隐藏层的特征数量。
- num_layers：循环神经网络的层数，默认值是 2。
- bias：默认为 True，如果为 false 则表示神经元不使用 bias 偏移参数。
- batch_first：如果设置为 True，则输入数据的维度中第一个维度就是 batch 值，默认为 False。默认情况下第一个维度是序列的长度，第二个维度才是 batch，第三个维度是特征数目。
- dropout：如果不为空，则表示最后跟一个 dropout 层抛弃部分数据，抛弃数据的比例由该参数指定。
- bidirectional ：布尔型，设置为 True 那么 LSTM 就是一个双向的 RNN 网络（双向是指同时可以往后影响参数），默认是 False。

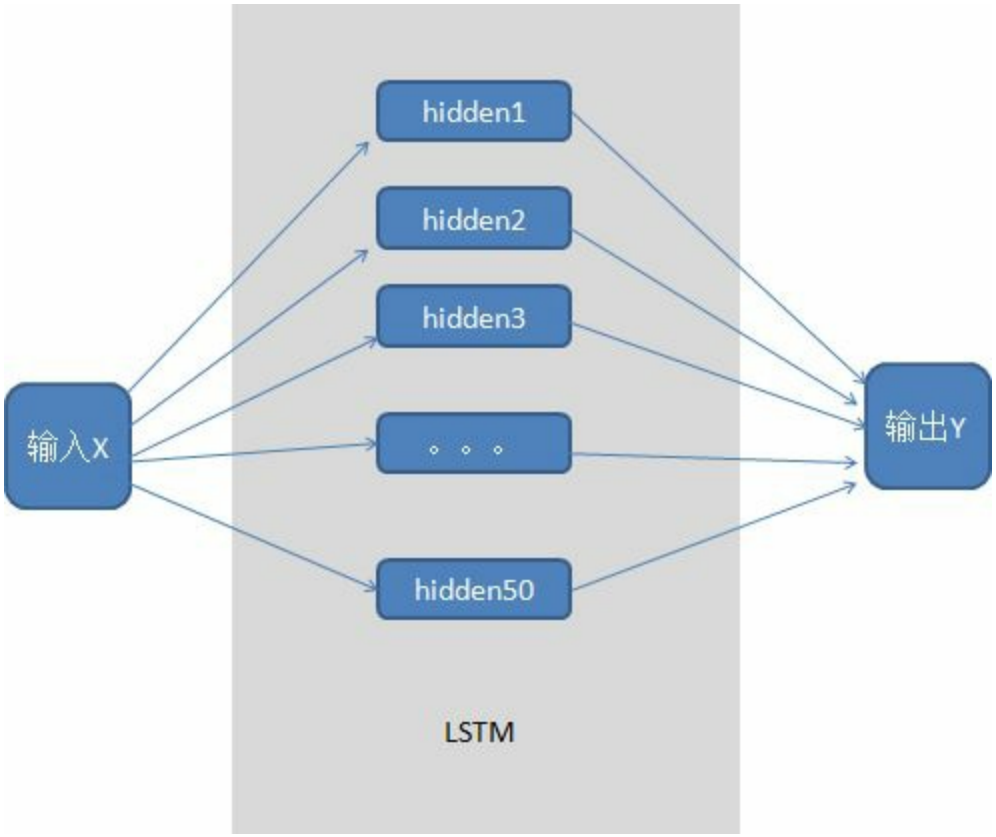
其中 input_size，hidden_size 和 num_layers 是前三个参数，也是需要手工设置的参数，num_layers 一般取默认值 2，这个值我们不必动它，就用 2 好了，它的含义是说隐藏层配置几层。LSTM 中最主要的参数是 input_size 和 hidden_size，这两个参数务必要搞清楚。其余的参数通常

不用设置，LSTM 会采用默认值就可以了。

下面这个图就是 num_layers 等于 2 的示意图，意思是说隐藏层有 2 个。



如果隐藏层只有一个的话则是下面这个图的样子。



num_layers 大部分情况下一般取默认值 2，我们在使用过程中也使用2吧。

LSTM 的用法形式是这样的：

```
output,(h_n,c_n)=LSTM(input,(h0,c0))
```

下面解释各个参数：

输入参数：input, (h_0, c_0)

- input (seq_len, batch, input_size)：输入数据 input 是一个三维向量，第一个维度是序列长度，第二个维度是 batch，也就是一批同时训练多少条数据，第三个维度是特征数目。如果实际数据的长度达不到序列长度 seq_len 的值，则可以先用 torch.nn.utils.rnn.pack_padded_sequence() 方法进行填充，这就是变长序列的由来。

- h_0 (num_layers * num_directions, batch, hidden_size)：隐藏层的初始权重，num_directions 一般为 1。

- c_0 (num_layers * num_directions, batch, hidden_size)：隐藏层的初始状态，num_directions 一般为 1。

输出数据：output, (h_n, c_n)

- output (seq_len, batch, hidden_size * num_directions)：输出数据。

- h_n (num_layers * num_directions, batch, hidden_size)：隐藏层的输出权重。

- c_n (num_layers * num_directions, batch, hidden_size)：隐藏层的输出状态。

示例代码：

```
rnn = nn.LSTM(10, 20, 2)
input = Variable(torch.randn(5, 3, 10))
h0 = Variable(torch.randn(2, 3, 20))
c0 = Variable(torch.randn(2, 3, 20))
output, hn = rnn(input, (h0, c0))
```

输入 input 的维度是 (5,3,10)，其中 5 是序列长度，3 是 batch，10 是特征数目。

output 的维度是 (5,3,20)，是输出数据。

hn 是隐藏层的参数，是一个 tuple。hn 的长度是 2：len（hn）=2。其中 hn[0].size()=（2,3,20）；hn[1].size()=（2,3,20）。hn[0] 是隐藏层的权重值，hn[1] 是隐藏层的状态值。对应关系如下：

```
hn[0] ==>  h0
hn[1] ==>  c0
```

我们说 LSTM 做深度学习模型，其实质是基于这样一个道理，在日常生活中，我们通常表示一句话的含义会有多种不同的表述方式，比如“我饿了”，“我要吃饭了”，“我现在好饿啊”等不同的话，其意义是一样的，都是“吃饭”。那么我们就可以将“我饿了”，“我要吃饭了”，“我现在好饿啊”作为输入数据，将“吃饭”作为结果标签，通过 LSTM 模型自动学习输入数据和结果标签之间的关联关系，自动提取特征值，模型训练之后会对诸如“我马上就要饿了”等语句自动分类到“吃饭”这个类别。这就是 LSTM 的现实意义。

15 Embedding层

在自然语言处理和文本分析的问题中，词袋（Bag of Words, BOW）和词向量（Word Embedding）是两种最常用的模型。更准确地说，词向量只能表征单个词，如果要表示文本，需要做一些额外的处理。下面就简单聊一下两种模型的应用。

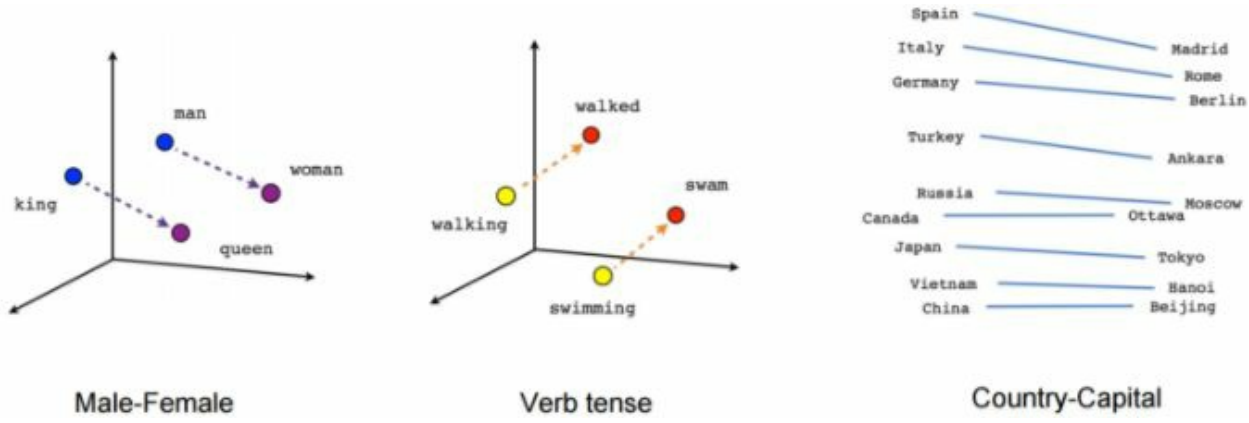
所谓 BOW，就是将文本 /Query 看作是一系列词的集合。由于词很多，所以咱们就用袋子把它们装起来，简称词袋。至于为什么用袋子而不用筐（basket）或者桶（bucket），这咱就不知道了。举个例子：

文本1：苏宁易购/是/国内/著名/的/B2C/电商/之一

这是一个短文本。“/”作为词与词之间的分割。从中我们可以看到这个文本包含“苏宁易购”，“B2C”，“电商”等词。换句话说，该文本的的词袋由“苏宁易购”，“电商”等词构成。就像这样：

但计算机不认识字，只认识数字，那在计算机中怎么表示词袋模型呢？其实很简单，给每个词一个位置/索引就可以了。例如，我们令“苏宁易购”的索引为0，“电商”的索引为 1，其他以此类推。则该文本的词袋就变成了：

是的，词袋变成了一串数字的（索引）的集合，这样计算机就能读懂了。



将意思相近的词的向量值安排的靠近一些，将意思不同的词的距离靠近的远一些。其原理大致就是在一段文本中的词意思一般来说是相近的，在不同的文本中的词意思大致是不同的，根据这个原理来计算 Embedding 模型。

在 PyTorch 中我们用 nn.Embedding 层来做嵌入词袋模型，下面我们就按照教程一步步走一遍，看看 Pytorch 的 Embedding 层是如何工作的。

16　LSTM文本分类

16.1　数据准备

先看看几个 pytorch 关于矩阵的方法的使用。

首先是生成数据：

```
import torch
```

生成每个元素都是1的矩阵：

torch.ones(2,2) ：输出[[1,1],[1,1]]

torch.ones(1,2,1)：输出[　[[1], [1]]　]

随机数：

torch.randn(1,2)：生成维度（1，2）的随机数。

torch.randn(1,2，4)：生成维度（1，2，4）的随机数。

再看维度转换：

a=torch.ones(2,2)：

a.view(-1)：转换为1维数组：

a.view(-1,2,1)：

16.2 数据源

20 Newsgroup dataset

本文使用的数据集是著名的“20 Newsgroup dataset”。该数据集共有 20 种新闻文本数据，我们将实现对该数据集的文本分类任务。数据集的说明和下载请参考这里。

不同类别的新闻包含大量不同的单词，在语义上存在极大的差别，。一些新闻类别如下所示：

以下是我们如何解决分类问题的步骤：

- 将所有的新闻样本转化为词索引序列。所谓词索引就是为每一个词依次分配一个整数 ID。遍历所有的新闻文本，我们只保留最参见的 20,000 个词，而且每个新闻文本最多保留400个词。
- 生成一个词向量矩阵。第i列表示词索引为i的词的词向量。
- 将词向量矩阵载入 Embedding 层，设置该层的权重不可再训练（也就是说在之后的网络训练过程中，词向量不再改变）。

Embedding 层之后连接一个 1D 的卷积层，并用一个 softmax 全连接输出新闻类别。

可以从网站下载训练数据：

http://www.cs.cmu.edu/afs/cs.cmu.edu/project/theo-20/www/data/news20.html

CMU Text Learning Group Data Archives

20_newsgroup

Download 20 Newshroup DataSet

This data set is a collection of 20,000 messages, collected from 20 different netnews newsgroups. One thousand messages from each of the twenty newsgroups were chosen at random and partitioned by newsgroup name. The list of newsgroups from which the messages were chose is as follows:

alt.atheism
talk.politics.guns
talk.politics.mideast
talk.politics.misc
talk.religion.misc
soc.religion.christian

comp.sys.ibm.pc.hardware
comp.graphics
comp.os.ms-windows.misc
comp.sys.mac.hardware
comp.windows.x

rec.autos
rec.motorcycles
rec.sport.baseball
rec.sport.hockey

sci.crypt
sci.electronics
sci.space
sci.med

misc.forsale

20 Newsgroup 数据，经典的 LSTM 情绪训练数据集。直接下载地址：

http://www.cs.cmu.edu/afs/cs.cmu.edu/project/theo-20/www/data/news20.tar.gz

下载完成后解压，我们随便打开一个文本文件看看里面是什么内容：

里面是一段英文句子。整个数据集分成 20 个目录，每个目录里有无数这样的文本文件。20 个目录就是这些文本文件的分类标签。

数据预处理的载入方法：我们首先遍历下语料文件下的所有文件夹，获得不同类别的新闻以及对应的类别标签，这里我们用到了 keras 的文本预处理的方法。

代码如下所示：

```
texts = []  # list of text samples
labels_index = {}  # dictionary mapping label name to numeric id
labels = []  # list of label ids
TEXT_DATA_DIR='data/20_newsgroup'
for name in sorted(os.listdir(TEXT_DATA_DIR)):
    path = os.path.join(TEXT_DATA_DIR, name)
    if os.path.isdir(path):
        label_id = len(labels_index)
        labels_index[name] = label_id
```

```
        for fname in sorted(os.listdir(path)):
            if fname.isdigit():
                fpath = os.path.join(path, fname)
                f = open(fpath)
                texts.append(f.read())
                f.close()
                labels.append(label_id)

print('Found %s texts.' % len(texts))
print('labels length %s .' % len(labels))
```

显示结果：

一共19997个文本文件。

下面我们将每篇文本分割成单词，并且给单词编上序号，将每篇文章转化成序号表示的矩阵。

```
MAX_NB_WORDS=20000
EMBEDDING_DIM=100
HIDDEN_DIM=100
MAX_SEQUENCE_LENGTH=400
epochs = 2
batch_size = 1
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
tokenizer = Tokenizer(num_words=MAX_NB_WORDS)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)
word_index = tokenizer.word_index
print('Found %s unique tokens.' % len(word_index))
print('word_index:',word_index)
data=[]
for i in range(len(sequences)):
    data.append(np.asarray(sequences[i]))
labels =np.asarray(labels)
print("data len:",len(data))
print("data sample:",data[0])
print('Shape of data tensor:', len(data))
print('Shape of label tensor:', labels.shape)
```

先看看 word_index 的内容，word_index 是词字典，给每个不同的单词标注了唯一的整型作为序号，部分内容如下：

最终生成 19997 条数组，每个数组是一个整型序号的集合，整型序号代表了单词的编码。

需要注意的是，这时候的 19997 条数据的长度是不一样的，单词多的文本它的数据长度就稍长一些。

接下来我们将数据分成训练数据和测试数据两部分，训练数据为 (x_train,y_train)，测试数据是 (x_test,y_test)。训练数据 15998 条，测试数据 3999 条。这段代码如下：

```
# split the data into a training set and a validation set
VALIDATION_SPLIT=0.2
nb_validation_samples = int(VALIDATION_SPLIT * len(data))

x_train = data[:-nb_validation_samples]
y_train = labels[:-nb_validation_samples]
x_test = data[-nb_validation_samples:]
y_test = labels[-nb_validation_samples:]
print("train length:",len(x_train))
print("test length:",len(x_test))

target_size= len(labels_index)
num_samples=len(x_train)
```

target_size 是整个数据的数量 19997，num_samples 是训练数据 的数目

15998。

16.3 中文分词

我们要对中文完成同样的分类任务，其实现原理都是一样的，除了一点不同，那就是需要先把中文进行分词。因为中文句子词与词之间是没有分隔符的，只要先将中文分词，然后就可以采用与英文分类同样的模型进行计算了。

这节主要介绍中文分词的方法。

中文和英文不同，各个单词是连在一起的，并没有空格把单词隔开。训练语言模型，我们首先需要将句子分词，对词空间进行向量分析。

我们选择"结巴"中文分词：号称做最好的 Python 中文分词组件，组件名称为 jieba。

从https://pypi.python.org/pypi/jieba/ 下载工具包：

解压后进入目录下，运行：python setup.py install

使用示例：

```
#! -*- coding:utf-8 -*-
import jieba
seg_list = jieba.cut("我来到北京清华大学", cut_all = True)
print "Full Mode:", ' '.join(seg_list)
seg_list = jieba.cut("我来到北京清华大学")
print "Default Mode:", ' '.join(seg_list)
```

显示结果：

```
Full Mode: 我 来到 北京 清华 清华大学 华大 大学
Default Mode: 我 来到 北京 清华大学
```

OK，jieba 分词安装成功！

下一步可以尝试中文 LSTM 模型了，其操作和英文的 LSTM 几乎是一模一样的，分词之后就可以按照英文示例来进行了，自己动手试试看吧。

16.4 模型

代码：

```
class LSTMNet(nn.Module):
     def __init__(self):
          super(LSTMNet, self).__init__()
          self.hidden_dim = HIDDEN_DIM
          self.embedding_dim = EMBEDDING_DIM
          self.word_embeddings = nn.Embedding(MAX_NB_WORDS, self.embedding_
          self.lstm = nn.LSTM(self.embedding_dim, self.hidden_dim,2,batch_f
          self.hidden2tag = nn.Linear(self.hidden_dim, 20)
          self.hidden = self.init_hidden()
          self.drop = nn.Dropout(p=0.2)


     def init_hidden(self):
          return (Variable(torch.zeros(2, batch_size, self.hidden_dim)), Va

     def forward(self, sentence):
          embeds = self.word_embeddings(sentence)
          embeds = self.drop(embeds)
          #lstm_out, self.hidden = self.lstm(embeds,self.hidden)
          lstm_out= self.lstm(embeds)
          out = lstm_out[0][:,-1,:]
          flat = out.view(-1, HIDDEN_DIM)
          tag_space = self.hidden2tag(flat)
          tag_scores = F.log_softmax(tag_space)
          return tag_scores

model = LSTMNet()
#if os.path.exists('torch_lstm.pkl'):
#     model =  torch.load('torch_lstm.pkl')
```

```
print(model)
```

先将不定长的文本数组经过Embedding层转换，输出 100 个嵌入层通道；然后再通过一个 LSTM 模型将 100 个 Embedding 通道转换成 100 个隐藏层，采用 2 个 Layer 的 LSTM 模型；这时候 LSTM 输出的也是 100 个隐藏层；最后将 100 个隐藏层全连接到 20 个输出神经元，这 20 个输出神经元就代表了 20 个分类标签。

16.5 训练

训练代码：

```
'''
trainning
'''
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
#loss=torch.nn.CrossEntropyLoss(size_average=True)

def train(epoch,x_train,y_train):
     num_batchs = num_samples/ batch_size
     model.train()
     model.hidden = model.init_hidden()
     for k in range(num_batchs):
          start,end = k*batch_size,(k+1)*batch_size
          data=Variable( torch.Tensor(x_train[start:end]).long())
          target = Variable(torch.Tensor(y_train[start:end]).long(),require
          #embeds = word_embeddings( Variable(t)) #,requires_grad=False)) #
          #data, target = Variable(x_train[start:end],requires_grad=False),
          #data, target = Variable(x_train[start:end]), Variable(y_train[st
          optimizer.zero_grad()
          #print("train data size:",data.size())
          output = model(data)
          #print("output :",output.size())
          #print("target :",target.size())
          loss = F.nll_loss(output,target) #criterion(output,target)
          loss.backward()
          optimizer.step()
```

```
          if k % 10 == 0:
               print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.form
                    epoch, k * len(data), num_samples,
                    100. * k / num_samples, loss.data[0]))
     torch.save(model, 'torch_lstm.pkl')
```

16.6 验证结果

```
'''
evaludate
'''
def test(epoch):
     model.eval()
     test_loss = 0
     correct = 0
     len=400
     print("x_test size:",len(x_test))
     for i in range(len):
          data, target = Variable(torch.Tensor(x_test[i:i+1]).long()), Vari
          output = model(data)
          test_loss += F.nll_loss(output, target).data[0]
          pred = output.data.max(1)[1] # get the index of the max log-proba
          correct += pred.eq(target.data).cpu().sum()
          test_loss = test_loss
          test_loss /= len(x_test) # loss function already averages over ba
          if i % 10 == 0:
               print("single loss:",test_loss,",right counts:",correct)
     print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.
               test_loss, correct, len(x_test),
               100. * correct / len))
```

用测试数据验证正确性

2 次迭代后：Test set: Average loss: 0.3419, Accuracy: 9140/10000 (91%)

3 次迭代后：Test set: Average loss: 0.2362, Accuracy: 9379/10000 (94%)

16.7 源码

```
# coding:utf-8
from __future__ import print_function
import numpy as np
import os
import struct
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.autograd import Variable

texts = []  # list of text samples
labels_index = {}  # dictionary mapping label name to numeric id
labels = []  # list of label ids
TEXT_DATA_DIR='data/20_newsgroup'
for name in sorted(os.listdir(TEXT_DATA_DIR)):
    path = os.path.join(TEXT_DATA_DIR, name)
    if os.path.isdir(path):
        label_id = len(labels_index)
        labels_index[name] = label_id
        for fname in sorted(os.listdir(path)):
            if fname.isdigit():
                fpath = os.path.join(path, fname)
                f = open(fpath)
                texts.append(f.read())
                f.close()
                labels.append(label_id)

print('Found %s texts.' % len(texts))
print('labels length %s .' % len(labels))

MAX_NB_WORDS=20000
EMBEDDING_DIM=100
HIDDEN_DIM=100
MAX_SEQUENCE_LENGTH=400
epochs = 2
batch_size = 1

from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
tokenizer = Tokenizer(num_words=MAX_NB_WORDS)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)
```

```
word_index = tokenizer.word_index
print('Found %s unique tokens.' % len(word_index))
print('word_index:',word_index)
data=[]
for i in range(len(sequences)):
     data.append(np.asarray(sequences[i]))

labels =np.asarray(labels)
print("data len:",len(data))
#print("data sample:",data[0])

print('Shape of data tensor:', len(data))
print('Shape of label tensor:', labels.shape)

# split the data into a training set and a validation set
VALIDATION_SPLIT=0.2
nb_validation_samples = int(VALIDATION_SPLIT * len(data))

x_train = data[:-nb_validation_samples]
y_train = labels[:-nb_validation_samples]
x_test = data[-nb_validation_samples:]
y_test = labels[-nb_validation_samples:]
print("train length:",len(x_train))
print("test length:",len(x_test))

target_size= len(labels_index)
num_samples=len(x_train)

'''
build torch model
'''
class LSTMNet(nn.Module):
     def __init__(self):
          super(LSTMNet, self).__init__()
          self.hidden_dim = HIDDEN_DIM
          self.embedding_dim = EMBEDDING_DIM
          self.word_embeddings = nn.Embedding(MAX_NB_WORDS, self.embedding_
          self.lstm = nn.LSTM(self.embedding_dim, self.hidden_dim,2,batch_f
          self.hidden2tag = nn.Linear(self.hidden_dim, 20)
          self.hidden = self.init_hidden()
          self.drop = nn.Dropout(p=0.2)

     def init_hidden(self):
          return (Variable(torch.zeros(2, batch_size, self.hidden_dim)), Va

     def forward(self, sentence):
          embeds = self.word_embeddings(sentence)
```

```
          embeds = self.drop(embeds)
          #lstm_out, self.hidden = self.lstm(embeds,self.hidden)
          lstm_out= self.lstm(embeds)
          out = lstm_out[0][:,-1,:]
          flat = out.view(-1, HIDDEN_DIM)
          tag_space = self.hidden2tag(flat)
          tag_scores = F.log_softmax(tag_space)
          return tag_scores

model = LSTMNet()
#if os.path.exists('torch_lstm.pkl'):
#     model =  torch.load('torch_lstm.pkl')
print(model)

'''
trainning
'''
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
#loss=torch.nn.CrossEntropyLoss(size_average=True)

def train(epoch,x_train,y_train):
     num_batchs = num_samples/ batch_size
     model.train()
     model.hidden = model.init_hidden()
     for k in range(num_batchs):
          start,end = k*batch_size,(k+1)*batch_size
          data=Variable( torch.Tensor(x_train[start:end]).long())
          target = Variable(torch.Tensor(y_train[start:end]).long(),require
          #embeds = word_embeddings( Variable(t)) #,requires_grad=False)) #
          #data, target = Variable(x_train[start:end],requires_grad=False),
          #data, target = Variable(x_train[start:end]), Variable(y_train[st
          optimizer.zero_grad()
          #print("train data size:",data.size())
          output = model(data)
          #print("output :",output.size())
          #print("target :",target.size())
          loss = F.nll_loss(output,target) #criterion(output,target)
          loss.backward()
          optimizer.step()
          if k % 10 == 0:
               print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.form
                    epoch, k * len(data), num_samples,
                    100. * k / num_samples, loss.data[0]))
     torch.save(model, 'torch_lstm.pkl')

'''
evaludate
```

```
'''
def test(epoch):
     model.eval()
     test_loss = 0
     correct = 0
     len=400
     print("x_test size:",len(x_test))
     for i in range(len):
          data, target = Variable(torch.Tensor(x_test[i:i+1]).long()), Vari
          output = model(data)
          test_loss += F.nll_loss(output, target).data[0]
          pred = output.data.max(1)[1] # get the index of the max log-proba
          correct += pred.eq(target.data).cpu().sum()
          test_loss = test_loss
          test_loss /= len(x_test) # loss function already averages over ba
          if i % 10 == 0:
               print("single loss:",test_loss,",right counts:",correct)
     print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.
               test_loss, correct, len(x_test),
               100. * correct / len))

for epoch in range(1,epochs):
    train(epoch,x_train,y_train)
    test(epoch)
```

17 参考

- 如何用PyTorch实现递归神经网络

  http://www.tuicool.com/articles/AV3AriB

- 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#smoothi
  filters

- pytorch优化器

  http://pytorch.org/docs/master/optim.html

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
