[DeepLearning tutorial（1）Softmax回归原理简介+代码详解](http://blog.csdn.net/u012162613/article/details/43157801)

**@author：wepon**

**@blog：**[**http://blog.csdn.net/u012162613/article/details/43157801**](http://blog.csdn.net/u012162613/article/details/43157801)

本文介绍Softmax回归算法，特别是详细解读其代码实现，基于python theano，代码来自：[Classifying MNIST digits using Logistic Regression](http://deeplearning.net/tutorial/logreg.html#logreg)，参考[UFLDL](http://deeplearning.stanford.edu/wiki/index.php/Softmax_Regression)。

**一、Softmax回归简介**

关于算法的详细教程本文没必要多说，可以参考UFLDL。下面只简单地总结一下，以便更好地理解代码。

Softmax回归其实就相当于多类别情况下的逻辑回归，对比如下：

逻辑回归的假设函数（hypothesis）：

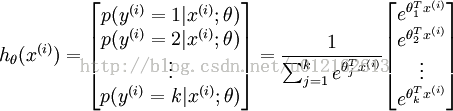
http://img.blog.csdn.net/20150126201745921?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvdTAxMjE2MjYxMw==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center

整个逻辑回归模型的参数就是theta，h(\*)是sigmoid函数，输出在0～1之间，一般作为二分类算法。对于具体的问题，找出最合适的theta便是最重要的步骤，这是最优化问题，一般通过定义代价函数，然后最小化代价函数来求解，逻辑回归的代价函数为：

http://img.blog.csdn.net/20150126202303274?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvdTAxMjE2MjYxMw==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center

最小化J(theta)，一般采用梯度下降算法，迭代计算梯度并更新theta。

Softmax的假设函数：



逻辑回归里将-theta\*x作为sigmoid函数的输入，得到的是0或者1，两个类别。而softmax有有k个类别，并且将-theta\*x作为指数的系数，所以就有e^(-theta\_1\*x)至e^( -theta\_k\*x)共k项，然后除以它们的累加和，这样做就实现了归一化，使得输出的k个数的和为1，而每一个数就代表那个类别出现的概率。因此：softmax的假设函数输出的是一个k维列向量，每一个维度的数就代表那个类别出现的概率。

Softmax的代价函数：

http://img.blog.csdn.net/20150126203904563?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvdTAxMjE2MjYxMw==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center

本质上跟逻辑回归是一样的，采用NLL，如果加上权重衰减项（正则化项），则为：

http://img.blog.csdn.net/20150126203954592?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvdTAxMjE2MjYxMw==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center

最小化代价函数，同样可以采用简单而有效的梯度下降，需要提到的是，在程序实现中，我们一般采用批量随机梯度下降，即MSGD，minibatch Stochastic Gradient Descent，简单来说，就是每遍历完一个batch的样本才计算梯度和更新参数，一个batch一般有几十到几百的单个样本。PS：随机梯度下降则是一个样本更新一次。

**二、Softmax代码详细解读**

首先说明一点，下面的程序采用的是MSGD算法，代价函数是不带权重衰减项的，整个程序实现用Softmax回归来classfy MINST数据集（识别手写数字0～9）。代码解读是个人理解，仅供参考，不一定正确，如有错误请不吝指出。

原始代码和经过我注释的代码:[github地址](https://github.com/wepe/MachineLearning-Demo/tree/master/DeepLearning%20Tutorials/Softmax_sgd(or%20logistic_sgd))

参数说明：上面第一部分我们的参数用theta表示，在下面的程序中，用的是W，权重，这两者是一样的。还有一点需要注意，上面的假设函数中是-theta\*x，而在程序中，用的是W\*X+b，本质也是一样的，因为可以将b看成W0，增加一个x0=1，则W\*X+b=WX=-theta\*x。

**（1）导入一些必要的模块**

**[python]** [view plaincopy](http://blog.csdn.net/u012162613/article/details/43157801)

1. **import** cPickle
2. **import** gzip
3. **import** os
4. **import** sys
5. **import** time
7. **import** numpy
9. **import** theano
10. **import** theano.tensor as T

**（2）定义Softmax回归模型**

在deeplearning tutorial中，直接将LogisticRegression视为Softmax，而我们所认识的二类别的逻辑回归就是当n\_out=2时的LogisticRegression，因此下面代码定义的LogisticRegression就是Softmax。

代码解读见注释：

**[python]** [view plaincopy](http://blog.csdn.net/u012162613/article/details/43157801)

1. #参数说明：
2. #input，输入的一个batch，假设一个batch有n个样本(n\_example)，则input大小就是(n\_example,n\_in)
3. #n\_in,每一个样本的大小，MNIST每个样本是一张28\*28的图片，故n\_in=784
4. #n\_out,输出的类别数，MNIST有0～9共10个类别，n\_out=10
5. **class** LogisticRegression(object):
6. **def** \_\_init\_\_(self, input, n\_in, n\_out):
8. #W大小是n\_in行n\_out列，b为n\_out维向量。即：每个输出对应W的一列以及b的一个元素。WX+b
9. #W和b都定义为theano.shared类型，这个是为了程序能在GPU上跑。
10. self.W = theano.shared(
11. value=numpy.zeros(
12. (n\_in, n\_out),
13. dtype=theano.config.floatX
14. ),
15. name='W',
16. borrow=True
17. )
19. self.b = theano.shared(
20. value=numpy.zeros(
21. (n\_out,),
22. dtype=theano.config.floatX
23. ),
24. name='b',
25. borrow=True
26. )
28. #input是(n\_example,n\_in)，W是（n\_in,n\_out）,点乘得到(n\_example,n\_out)，加上偏置b，
29. #再作为T.nnet.softmax的输入，得到p\_y\_given\_x
30. #故p\_y\_given\_x每一行代表每一个样本被估计为各类别的概率
31. #PS：b是n\_out维向量，与(n\_example,n\_out)矩阵相加，内部其实是先复制n\_example个b，
32. #然后(n\_example,n\_out)矩阵的每一行都加b
33. self.p\_y\_given\_x = T.nnet.softmax(T.dot(input, self.W) + self.b)
35. #argmax返回最大值下标，因为本例数据集是MNIST，下标刚好就是类别。axis=1表示按行操作。
36. self.y\_pred = T.argmax(self.p\_y\_given\_x, axis=1)
38. #params，模型的参数
39. self.params = [self.W, self.b]
41. #代价函数NLL
42. #因为我们是MSGD，每次训练一个batch，一个batch有n\_example个样本，则y大小是(n\_example,),
43. #y.shape[0]得出行数即样本数，将T.log(self.p\_y\_given\_x)简记为LP，
44. #则LP[T.arange(y.shape[0]),y]得到[LP[0,y[0]], LP[1,y[1]], LP[2,y[2]], ...,LP[n-1,y[n-1]]]
45. #最后求均值mean，也就是说，minibatch的SGD，是计算出batch里所有样本的NLL的平均值，作为它的cost
46. **def** negative\_log\_likelihood(self, y):
47. **return** -T.mean(T.log(self.p\_y\_given\_x)[T.arange(y.shape[0]), y])
49. #batch的误差率
50. **def** errors(self, y):
51. # 首先检查y与y\_pred的维度是否一样，即是否含有相等的样本数
52. **if** y.ndim != self.y\_pred.ndim:
53. **raise** TypeError(
54. 'y should have the same shape as self.y\_pred',
55. ('y', y.type, 'y\_pred', self.y\_pred.type)
56. )
57. # 再检查是不是int类型，是的话计算T.neq(self.y\_pred, y)的均值，作为误差率
58. #举个例子，假如self.y\_pred=[3,2,3,2,3,2],而实际上y=[3,4,3,4,3,4]
59. #则T.neq(self.y\_pred, y)=[0,1,0,1,0,1],1表示不等，0表示相等
60. #故T.mean(T.neq(self.y\_pred, y))=T.mean([0,1,0,1,0,1])=0.5，即错误率50%
61. **if** y.dtype.startswith('int'):
62. **return** T.mean(T.neq(self.y\_pred, y))
63. **else**:
64. **raise** NotImplementedError()

上面已经定义好了softmax模型，包括输入的batch ：input，每个样本的大小n\_in，输出的类别n\_out，模型的参数W、b，模型预测的输出y\_pred，代价函数NLL，以及误差率errors。

**（3）加载MNIST数据集**

**[python]** [view plaincopy](http://blog.csdn.net/u012162613/article/details/43157801)

1. **def** load\_data(dataset):
2. # dataset是数据集的路径，程序首先检测该路径下有没有MNIST数据集，没有的话就下载MNIST数据集
3. #这一部分就不解释了，与softmax回归算法无关。
4. data\_dir, data\_file = os.path.split(dataset)
5. **if** data\_dir == "" **and** **not** os.path.isfile(dataset):
6. # Check if dataset is in the data directory.
7. new\_path = os.path.join(
8. os.path.split(\_\_file\_\_)[0],
9. "..",
10. "data",
11. dataset
12. )
13. **if** os.path.isfile(new\_path) **or** data\_file == 'mnist.pkl.gz':
14. dataset = new\_path
16. **if** (**not** os.path.isfile(dataset)) **and** data\_file == 'mnist.pkl.gz':
17. **import** urllib
18. origin = (
19. 'http://www.iro.umontreal.ca/~lisa/deep/data/mnist/mnist.pkl.gz'
20. )
21. **print** 'Downloading data from %s' % origin
22. urllib.urlretrieve(origin, dataset)
24. **print** '... loading data'
25. #以上是检测并下载数据集mnist.pkl.gz，不是本文重点。下面才是load\_data的开始
27. #从"mnist.pkl.gz"里加载train\_set, valid\_set, test\_set，它们都是包括label的
28. #主要用到python里的gzip.open()函数,以及 cPickle.load()。
29. #‘rb’表示以二进制可读的方式打开文件
30. f = gzip.open(dataset, 'rb')
31. train\_set, valid\_set, test\_set = cPickle.load(f)
32. f.close()

35. #将数据设置成shared variables，主要时为了GPU加速，只有shared variables才能存到GPU memory中
36. #GPU里数据类型只能是float。而data\_y是类别，所以最后又转换为int返回
37. **def** shared\_dataset(data\_xy, borrow=True):
38. data\_x, data\_y = data\_xy
39. shared\_x = theano.shared(numpy.asarray(data\_x,
40. dtype=theano.config.floatX),
41. borrow=borrow)
42. shared\_y = theano.shared(numpy.asarray(data\_y,
43. dtype=theano.config.floatX),
44. borrow=borrow)
45. **return** shared\_x, T.cast(shared\_y, 'int32')

48. test\_set\_x, test\_set\_y = shared\_dataset(test\_set)
49. valid\_set\_x, valid\_set\_y = shared\_dataset(valid\_set)
50. train\_set\_x, train\_set\_y = shared\_dataset(train\_set)
52. rval = [(train\_set\_x, train\_set\_y), (valid\_set\_x, valid\_set\_y),
53. (test\_set\_x, test\_set\_y)]
54. **return** rval

**（4）将模型应用于MNIST数据集**

**[python]** [view plaincopy](http://blog.csdn.net/u012162613/article/details/43157801)

1. **def** sgd\_optimization\_mnist(learning\_rate=0.13, n\_epochs=1000,
2. dataset='mnist.pkl.gz',
3. batch\_size=600):
4. #加载数据
5. datasets = load\_data(dataset)
6. train\_set\_x, train\_set\_y = datasets[0]
7. valid\_set\_x, valid\_set\_y = datasets[1]
8. test\_set\_x, test\_set\_y = datasets[2]
9. #计算有多少个minibatch，因为我们的优化算法是MSGD，是一个batch一个batch来计算cost的
10. n\_train\_batches = train\_set\_x.get\_value(borrow=True).shape[0] / batch\_size
11. n\_valid\_batches = valid\_set\_x.get\_value(borrow=True).shape[0] / batch\_size
12. n\_test\_batches = test\_set\_x.get\_value(borrow=True).shape[0] / batch\_size
14. ######################
15. # 开始建模            #
16. ######################
17. **print** '... building the model'

20. #设置变量，index表示minibatch的下标，x表示训练样本，y是对应的label
21. index = T.lscalar()
22. x = T.matrix('x')
23. y = T.ivector('y')

26. #定义分类器，用x作为input初始化。
27. classifier = LogisticRegression(input=x, n\_in=28 \* 28, n\_out=10)

30. #定义代价函数，用y来初始化，而其实还有一个隐含的参数x在classifier中。
31. #这样理解才是合理的，因为cost必须由x和y得来，单单y是得不到cost的。
32. cost = classifier.negative\_log\_likelihood(y)

35. #这里必须说明一下theano的function函数，givens是字典，其中的x、y是key，冒号后面是它们的value。
36. #在function被调用时，x、y将被具体地替换为它们的value，而value里的参数index就是inputs=[index]这里给出。
37. #下面举个例子：
38. #比如test\_model(1)，首先根据index=1具体化x为test\_set\_x[1 \* batch\_size: (1 + 1) \* batch\_size]，
39. #具体化y为test\_set\_y[1 \* batch\_size: (1 + 1) \* batch\_size]。然后函数计算outputs=classifier.errors(y)，
40. #这里面有参数y和隐含的x，所以就将givens里面具体化的x、y传递进去。
41. test\_model = theano.function(
42. inputs=[index],
43. outputs=classifier.errors(y),
44. givens={
45. x: test\_set\_x[index \* batch\_size: (index + 1) \* batch\_size],
46. y: test\_set\_y[index \* batch\_size: (index + 1) \* batch\_size]
47. }
48. )

51. validate\_model = theano.function(
52. inputs=[index],
53. outputs=classifier.errors(y),
54. givens={
55. x: valid\_set\_x[index \* batch\_size: (index + 1) \* batch\_size],
56. y: valid\_set\_y[index \* batch\_size: (index + 1) \* batch\_size]
57. }
59. # 计算各个参数的梯度
60. g\_W = T.grad(cost=cost, wrt=classifier.W)
61. g\_b = T.grad(cost=cost, wrt=classifier.b)
63. #更新的规则，根据梯度下降法的更新公式
64. updates = [(classifier.W, classifier.W - learning\_rate \* g\_W),
65. (classifier.b, classifier.b - learning\_rate \* g\_b)]
67. #train\_model跟上面分析的test\_model类似，只是这里面多了updatas，更新规则用上面定义的updates 列表。
68. train\_model = theano.function(
69. inputs=[index],
70. outputs=cost,
71. updates=updates,
72. givens={
73. x: train\_set\_x[index \* batch\_size: (index + 1) \* batch\_size],
74. y: train\_set\_y[index \* batch\_size: (index + 1) \* batch\_size]
75. }
76. )
78. ###############
79. # 开始训练     #
80. ###############
81. **print** '... training the model'
83. patience = 5000
84. patience\_increase = 2
85. #提高的阈值，在验证误差减小到之前的0.995倍时，会更新best\_validation\_loss
86. improvement\_threshold = 0.995
87. #这样设置validation\_frequency可以保证每一次epoch都会在验证集上测试。
88. validation\_frequency = min(n\_train\_batches, patience / 2)

91. best\_validation\_loss = numpy.inf   #最好的验证集上的loss，最好即最小。初始化为无穷大
92. test\_score = 0.
93. start\_time = time.clock()
95. done\_looping = False
96. epoch = 0
98. #下面就是训练过程了，while循环控制的时步数epoch，一个epoch会遍历所有的batch，即所有的图片。
99. #for循环是遍历一个个batch，一次一个batch地训练。for循环体里会用train\_model(minibatch\_index)去训练模型，
100. #train\_model里面的updatas会更新各个参数。
101. #for循环里面会累加训练过的batch数iter，当iter是validation\_frequency倍数时则会在验证集上测试，
102. #如果验证集的损失this\_validation\_loss小于之前最佳的损失best\_validation\_loss，
103. #则更新best\_validation\_loss和best\_iter，同时在testset上测试。
104. #如果验证集的损失this\_validation\_loss小于best\_validation\_loss\*improvement\_threshold时则更新patience。
105. #当达到最大步数n\_epoch时，或者patience<iter时，结束训练
106. **while** (epoch < n\_epochs) **and** (**not** done\_looping):
107. epoch = epoch + 1
108. **for** minibatch\_index **in** xrange(n\_train\_batches):
110. minibatch\_avg\_cost = train\_model(minibatch\_index)
111. # iteration number
112. iter = (epoch - 1) \* n\_train\_batches + minibatch\_index
114. **if** (iter + 1) % validation\_frequency == 0:
115. # compute zero-one loss on validation set
116. validation\_losses = [validate\_model(i)
117. **for** i **in** xrange(n\_valid\_batches)]
118. this\_validation\_loss = numpy.mean(validation\_losses)
120. **print**(
121. 'epoch %i, minibatch %i/%i, validation error %f %%' %
122. (
123. epoch,
124. minibatch\_index + 1,
125. n\_train\_batches,
126. this\_validation\_loss \* 100.
127. )
128. )
130. # if we got the best validation score until now
131. **if** this\_validation\_loss < best\_validation\_loss:
132. #improve patience if loss improvement is good enough
133. **if** this\_validation\_loss < best\_validation\_loss \*  \
134. improvement\_threshold:
135. patience = max(patience, iter \* patience\_increase)
137. best\_validation\_loss = this\_validation\_loss
138. # test it on the test set
140. test\_losses = [test\_model(i)
141. **for** i **in** xrange(n\_test\_batches)]
142. test\_score = numpy.mean(test\_losses)
144. **print**(
145. (
146. '     epoch %i, minibatch %i/%i, test error of'
147. ' best model %f %%'
148. ) %
149. (
150. epoch,
151. minibatch\_index + 1,
152. n\_train\_batches,
153. test\_score \* 100.
154. )
155. )
157. **if** patience <= iter:
158. done\_looping = True
159. **break**
161. #while循环结束
162. end\_time = time.clock()
163. **print**(
164. (
165. 'Optimization complete with best validation score of %f %%,'
166. 'with test performance %f %%'
167. )
168. % (best\_validation\_loss \* 100., test\_score \* 100.)
169. )
170. **print** 'The code run for %d epochs, with %f epochs/sec' % (
171. epoch, 1. \* epoch / (end\_time - start\_time))
172. **print** >> sys.stderr, ('The code for file ' +
173. os.path.split(\_\_file\_\_)[1] +
174. ' ran for %.1fs' % ((end\_time - start\_time)))

[DeepLearning tutorial（2）机器学习算法在训练过程中保存参数](http://blog.csdn.net/u012162613/article/details/43169019)

@author：wepon

@blog：<http://blog.csdn.net/u012162613/article/details/43169019>

参考：[pickle — Python object serialization](https://docs.python.org/2/library/pickle.html)、[DeepLearning Getting started](http://deeplearning.net/tutorial/gettingstarted.html#theano-python-tips)

**一、python读取"\*\*\*.pkl.gz"文件**

用到python里的gzip以及cPickle模块，简单的使用代码如下，如果想详细了解可以参考上面给出的链接。

**[python]** [view plaincopy](http://blog.csdn.net/u012162613/article/details/43169019)

1. #以读取mnist.pkl.gz为例
2. **import** cPickle, gzip
3. f = gzip.open('mnist.pkl.gz', 'rb')
4. train\_set, valid\_set, test\_set = cPickle.load(f)
5. f.close()

其实就是分两步，先读取gz文件，再读取pkl文件。pkl文件的应用正是下文要讲的，我们用它来保存机器学习算法训练过程中的参数。

**二、机器学习算法在训练过程中如何保存参数？**

我们知道，机器学习算法的计算量特别大，跑起程序来少则几十分钟，多则几小时甚至几天，中间如果有什么状况（比如电脑过热重启、程序出现一些小bug...）程序就会中断，如果你没把参数定时保存下来，前面的训练就当白费了，所以很有必要在程序中加入定时保存参数的功能，这样下次训练就可以将参数初始化为上次保存下来的结果，而不是从头开始随机初始化。

那么如何保存模型参数？可以将参数深复制，或者调用python的数据永久存储cPickle模块，原理不多说，直接使用就行。（注：python里有cPickle和pickle，cPickle基于c实现，比pickle快。）

直接用一个例子来说明如何使用：

**[python]** [view plaincopy](http://blog.csdn.net/u012162613/article/details/43169019)

1. a=[1,2,3]
2. b={4:5,6:7}
3. #保存，cPickle.dump函数。/home/wepon/ab是路径，ab是保存的文件的名字，如果/home/wepon/下本来就有ab这个文件，将被覆写#，如果没有，则创建。'wb'表示以二进制可写的方式打开。dump中的-1表示使用highest protocol。
4. **import** cPickle
5. write\_file=open('/home/wepon/ab','wb')
6. cPickle.dump(a,write\_file,-1)
7. cPickle.dump(b,write\_file,-1)
8. write\_file.close()
10. #读取，cPickle.load函数。
11. read\_file=open('/home/wepon/ab','rb')
12. a\_1=cPickle.load(read\_file)
13. b\_1=cPickle.load(read\_file)
14. **print** a,b
15. read\_file.close()

在deeplearning算法中，因为用到GPU，经常是将参数声明为shared变量，因此必须用上get\_value()、set\_value，例如有w、v、u三个shared变量，使用代码如下：

**[python]** [view plaincopy](http://blog.csdn.net/u012162613/article/details/43169019)

1. **import** cPickle
2. #保存
3. write\_file = open('path', 'wb')
4. cPickle.dump(w.get\_value(borrow=True), write\_file, -1)
5. cPickle.dump(v.get\_value(borrow=True), write\_file, -1)
6. cPickle.dump(u.get\_value(borrow=True), write\_file, -1)
7. write\_file.close()
9. #读取
10. read\_file = open('path')
11. w.set\_value(cPickle.load(read\_file), borrow=True)
12. v.set\_value(cPickle.load(read\_file), borrow=True)
13. u.set\_value(cPickle.load(read\_file), borrow=True)
14. read\_file.close()

一个实例

下面我以一个实际的例子来说明如何在程序中加入保存参数的功能。以deeplearnig.net上的逻辑回归为例，它的代码地址：[logistic\_sgd.py](http://deeplearning.net/tutorial/code/logistic_sgd.py)。这个程序是将逻辑回归用于MNIST分类，程序运行过程并不会保存参数，甚至运行结束时也不保存参数。怎么做可以保存参数？

在logistic\_sgd.py代码里最后面的sgd\_optimization\_mnist()函数里，有个while循环，里面有一句代码：

**[python]** [view plaincopy](http://blog.csdn.net/u012162613/article/details/43169019)

1. **if** this\_validation\_loss < best\_validation\_loss:

这句代码的意思就是判断当前的验证损失是否小于最佳的验证损失，是的话，下面会更新best\_validation\_loss，也就是说当前参数下，模型比之前的有了优化，因此我们可以在这个if语句后面加入保存参数的代码:

**[python]** [view plaincopy](http://blog.csdn.net/u012162613/article/details/43169019)

1. save\_params(classifier.W,classifier.b)

save\_params函数定义如下：

**[python]** [view plaincopy](http://blog.csdn.net/u012162613/article/details/43169019)

1. **def** save\_params(param1,param2):
2. **import** cPickle
3. write\_file = open('params', 'wb')
4. cPickle.dump(param1.get\_value(borrow=True), write\_file, -1)
5. cPickle.dump(param2.get\_value(borrow=True), write\_file, -1)
6. write\_file.close()

当然参数的个数根据需要去定义。在logistic\_sgd.py中参数只有classifier.W,classifier.b，因此这里定义为save\_params(param1,param2)。

在logistic\_sgd.py里我加入了save\_params(classifier.W,classifier.b)，运行了3次epoch，中断掉程序，在代码所在的文件夹下，多出了一个params文件，我们来看看这个文件里是什么东西：

**[python]** [view plaincopy](http://blog.csdn.net/u012162613/article/details/43169019)

1. **import** cPickle
2. f=open('params')
3. w=cPickle.load(f)
4. b=cPickle.load(f)
6. #w大小是(n\_in,n\_out)，b大小时(n\_out,),b的值如下，因为MINST有10个类别，n\_out=10,下面正是10个数
7. array([-0.0888151 ,  0.16875755, -0.03238435, -0.06493175,  0.05245609,
8. 0.1754718 , -0.0155049 ,  0.11216578, -0.26740651, -0.03980861])

也就是说，params文件确实保存了我们训练过程中的参数。

那么如何用保存下来的参数来初始化我们的模型的参数呢？

在logistic\_sgd.py中的class LogisticRegression(object)下，self.W和self.b本来是初始化为0的，我们可以在下面加上几行代码，这样就可以用我们保存下来的params文件来初始化参数了：

**[python]** [view plaincopy](http://blog.csdn.net/u012162613/article/details/43169019)

1. **class** LogisticRegression(object):
2. **def** \_\_init\_\_(self, input, n\_in, n\_out):
3. self.W = theano.shared(
4. value=numpy.zeros(
5. (n\_in, n\_out),
6. dtype=theano.config.floatX
7. ),
8. name='W',
9. borrow=True
10. )
11. self.b = theano.shared(
12. value=numpy.zeros(
13. (n\_out,),
14. dtype=theano.config.floatX
15. ),
16. name='b',
17. borrow=True
18. )
19. #！！！
20. #加入的代码在这里，程序运行到这里将会判断当前路径下有没有params文件，有的话就拿来初始化W和b
21. **if** os.path.exists('params'):
22. f=open('params')
23. self.W.set\_value(cPickle.load(f), borrow=True)
24. self.b.set\_value(cPickle.load(f), borrow=True)

# [DeepLearning tutorial（3）MLP多层感知机原理简介+代码详解](http://blog.csdn.net/u012162613/article/details/43221829)

@author：wepon

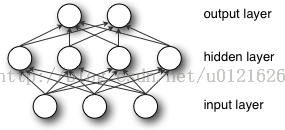
@blog：<http://blog.csdn.net/u012162613/article/details/43221829>

本文介绍多层感知机算法，特别是详细解读其代码实现，基于python theano，代码来自：[Multilayer Perceptron](http://deeplearning.net/tutorial/mlp.html#mlp)，如果你想详细了解多层感知机算法，可以参考：[UFLDL教程](http://deeplearning.stanford.edu/wiki/index.php/%E7%A5%9E%E7%BB%8F%E7%BD%91%E7%BB%9C)，或者参考本文第一部分的算法简介。

经详细注释的代码：[放在我的github地址上，可下载](https://github.com/wepe/MachineLearning-Demo/tree/master/DeepLearning%20Tutorials/mlp)。

# 一、多层感知机（MLP）原理简介

多层感知机（MLP，Multilayer Perceptron）也叫人工神经网络（ANN，Artificial Neural Network），除了输入输出层，它中间可以有多个隐层，最简单的MLP只含一个隐层，即三层的结构，如下图：



从上图可以看到，多层感知机层与层之间是全连接的（全连接的意思就是：上一层的任何一个神经元与下一层的所有神经元都有连接）。多层感知机最底层是输入层，中间是隐藏层，最后是输出层。

输入层没什么好说，你输入什么就是什么，比如输入是一个n维向量，就有n个神经元。

隐藏层的神经元怎么得来？首先它与输入层是全连接的，假设输入层用向量X表示，则隐藏层的输出就是

f(W1X+b1)，W1是权重（也叫连接系数），b1是偏置，函数f 可以是常用的sigmoid函数或者tanh函数：

http://img.blog.csdn.net/20150128033821825?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvdTAxMjE2MjYxMw==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center       http://img.blog.csdn.net/20150128033840093?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvdTAxMjE2MjYxMw==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center

最后就是输出层，输出层与隐藏层是什么关系？其实隐藏层到输出层可以看成是一个多类别的逻辑回归，也即softmax回归，所以输出层的输出就是softmax(W2X1+b2)，X1表示隐藏层的输出f(W1X+b1)。

MLP整个模型就是这样子的，上面说的这个三层的MLP用公式总结起来就是，函数G是softmax

http://img.blog.csdn.net/20150128034130632?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvdTAxMjE2MjYxMw==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center

因此，MLP所有的参数就是各个层之间的连接权重以及偏置，包括W1、b1、W2、b2。对于一个具体的问题，怎么确定这些参数？求解最佳的参数是一个最优化问题，解决最优化问题，最简单的就是梯度下降法了（SGD）：首先随机初始化所有参数，然后迭代地训练，不断地计算梯度和更新参数，直到满足某个条件为止（比如误差足够小、迭代次数足够多时）。这个过程涉及到代价函数、规则化（Regularization）、学习速率（learning rate）、梯度计算等，本文不详细讨论，读者可以参考本文顶部给出的两个链接。

了解了MLP的基本模型，下面进入代码实现部分。

## 二、多层感知机（MLP）代码详细解读（基于python+theano）

再次说明，代码来自：[Multilayer Perceptron](http://deeplearning.net/tutorial/mlp.html#mlp)，本文只是做一个详细解读，如有错误，请不吝指出。

这个代码实现的是一个三层的感知机，但是理解了代码之后，实现n层感知机都不是问题，所以只需理解好这个三层的MLP模型即可。概括地说，MLP的输入层X其实就是我们的训练数据，所以输入层不用实现，剩下的就是“输入层到隐含层”，“隐含层到输出层”这两部分。上面介绍原理时已经说到了，“输入层到隐含层”就是一个全连接的层，在下面的代码中我们把这一部分定义为HiddenLayer。“隐含层到输出层”就是一个分类器softmax回归（也有人叫逻辑回归），在下面的代码中我们把这一部分定义为LogisticRegression。

代码详解开始：

### （1）导入必要的python模块

主要是numpy、theano，以及python自带的os、sys、time模块，这些模块的使用在下面的程序中会看到。

**[python]** [view plaincopy](http://blog.csdn.net/u012162613/article/details/43221829)

1. **import** os
2. **import** sys
3. **import** time
5. **import** numpy
7. **import** theano
8. **import** theano.tensor as T

### （2）定义MLP模型（HiddenLayer+LogisticRegression）

这一部分定义MLP的基本“构件”，即上文一直在提的HiddenLayer和LogisticRegression

* HiddenLayer

隐含层我们需要定义连接系数W、偏置b，输入、输出，具体的代码以及解读如下：

**[python]** [view plaincopy](http://blog.csdn.net/u012162613/article/details/43221829)

1. **class** HiddenLayer(object):
2. **def** \_\_init\_\_(self, rng, input, n\_in, n\_out, W=None, b=None,
3. activation=T.tanh):
4. """
5. 注释：
6. 这是定义隐藏层的类，首先明确：隐藏层的输入即input，输出即隐藏层的神经元个数。输入层与隐藏层是全连接的。
7. 假设输入是n\_in维的向量（也可以说时n\_in个神经元），隐藏层有n\_out个神经元，则因为是全连接，
8. 一共有n\_in\*n\_out个权重，故W大小时(n\_in,n\_out),n\_in行n\_out列，每一列对应隐藏层的每一个神经元的连接权重。
9. b是偏置，隐藏层有n\_out个神经元，故b时n\_out维向量。
10. rng即随机数生成器，numpy.random.RandomState，用于初始化W。
11. input训练模型所用到的所有输入，并不是MLP的输入层，MLP的输入层的神经元个数时n\_in，而这里的参数input大小是（n\_example,n\_in）,每一行一个样本，即每一行作为MLP的输入层。
12. activation:激活函数,这里定义为函数tanh
13. """
15. self.input = input   #类HiddenLayer的input即所传递进来的input
17. """
18. 注释：
19. 代码要兼容GPU，则W、b必须使用 dtype=theano.config.floatX,并且定义为theano.shared
20. 另外，W的初始化有个规则：如果使用tanh函数，则在-sqrt(6./(n\_in+n\_hidden))到sqrt(6./(n\_in+n\_hidden))之间均匀
21. 抽取数值来初始化W，若时sigmoid函数，则以上再乘4倍。
22. """
23. #如果W未初始化，则根据上述方法初始化。
24. #加入这个判断的原因是：有时候我们可以用训练好的参数来初始化W，见我的上一篇文章。
25. **if** W **is** None:
26. W\_values = numpy.asarray(
27. rng.uniform(
28. low=-numpy.sqrt(6. / (n\_in + n\_out)),
29. high=numpy.sqrt(6. / (n\_in + n\_out)),
30. size=(n\_in, n\_out)
31. ),
32. dtype=theano.config.floatX
33. )
34. **if** activation == theano.tensor.nnet.sigmoid:
35. W\_values \*= 4
36. W = theano.shared(value=W\_values, name='W', borrow=True)
38. **if** b **is** None:
39. b\_values = numpy.zeros((n\_out,), dtype=theano.config.floatX)
40. b = theano.shared(value=b\_values, name='b', borrow=True)
42. #用上面定义的W、b来初始化类HiddenLayer的W、b
43. self.W = W
44. self.b = b
46. #隐含层的输出
47. lin\_output = T.dot(input, self.W) + self.b
48. self.output = (
49. lin\_output **if** activation **is** None
50. **else** activation(lin\_output)
51. )
53. #隐含层的参数
54. self.params = [self.W, self.b]

* LogisticRegression

逻辑回归（softmax回归），代码详解如下。

（如果你想详细了解softmax回归，可以参考： [DeepLearning tutorial（1）Softmax回归原理简介+代码详解](http://blog.csdn.net/u012162613/article/details/43157801)）

**[python]** [view plaincopy](http://blog.csdn.net/u012162613/article/details/43221829)

1. """
2. 定义分类层，Softmax回归
3. 在deeplearning tutorial中，直接将LogisticRegression视为Softmax，
4. 而我们所认识的二类别的逻辑回归就是当n\_out=2时的LogisticRegression
5. """
6. #参数说明：
7. #input，大小就是(n\_example,n\_in)，其中n\_example是一个batch的大小，
8. #因为我们训练时用的是Minibatch SGD，因此input这样定义
9. #n\_in,即上一层(隐含层)的输出
10. #n\_out,输出的类别数
11. **class** LogisticRegression(object):
12. **def** \_\_init\_\_(self, input, n\_in, n\_out):
14. #W大小是n\_in行n\_out列，b为n\_out维向量。即：每个输出对应W的一列以及b的一个元素。
15. self.W = theano.shared(
16. value=numpy.zeros(
17. (n\_in, n\_out),
18. dtype=theano.config.floatX
19. ),
20. name='W',
21. borrow=True
22. )
24. self.b = theano.shared(
25. value=numpy.zeros(
26. (n\_out,),
27. dtype=theano.config.floatX
28. ),
29. name='b',
30. borrow=True
31. )
33. #input是(n\_example,n\_in)，W是（n\_in,n\_out）,点乘得到(n\_example,n\_out)，加上偏置b，
34. #再作为T.nnet.softmax的输入，得到p\_y\_given\_x
35. #故p\_y\_given\_x每一行代表每一个样本被估计为各类别的概率
36. #PS：b是n\_out维向量，与(n\_example,n\_out)矩阵相加，内部其实是先复制n\_example个b，
37. #然后(n\_example,n\_out)矩阵的每一行都加b
38. self.p\_y\_given\_x = T.nnet.softmax(T.dot(input, self.W) + self.b)
40. #argmax返回最大值下标，因为本例数据集是MNIST，下标刚好就是类别。axis=1表示按行操作。
41. self.y\_pred = T.argmax(self.p\_y\_given\_x, axis=1)
43. #params，LogisticRegression的参数
44. self.params = [self.W, self.b]

ok！这两个基本“构件”做好了，现在我们可以将它们“组装”在一起。

我们要三层的MLP，则只需要HiddenLayer+LogisticRegression，

如果要四层的MLP，则为HiddenLayer+HiddenLayer+LogisticRegression........以此类推。

下面是三层的MLP：

**[python]** [view plaincopy](http://blog.csdn.net/u012162613/article/details/43221829)

1. #3层的MLP
2. **class** MLP(object):
3. **def** \_\_init\_\_(self, rng, input, n\_in, n\_hidden, n\_out):
5. self.hiddenLayer = HiddenLayer(
6. rng=rng,
7. input=input,
8. n\_in=n\_in,
9. n\_out=n\_hidden,
10. activation=T.tanh
11. )
13. #将隐含层hiddenLayer的输出作为分类层logRegressionLayer的输入，这样就把它们连接了
14. self.logRegressionLayer = LogisticRegression(
15. input=self.hiddenLayer.output,
16. n\_in=n\_hidden,
17. n\_out=n\_out
18. )

21. #以上已经定义好MLP的基本结构，下面是MLP模型的其他参数或者函数
23. #规则化项：常见的L1、L2\_sqr
24. self.L1 = (
25. abs(self.hiddenLayer.W).sum()
26. + abs(self.logRegressionLayer.W).sum()
27. )
29. self.L2\_sqr = (
30. (self.hiddenLayer.W \*\* 2).sum()
31. + (self.logRegressionLayer.W \*\* 2).sum()
32. )

35. #损失函数Nll（也叫代价函数）
36. self.negative\_log\_likelihood = (
37. self.logRegressionLayer.negative\_log\_likelihood
38. )
40. #误差
41. self.errors = self.logRegressionLayer.errors
43. #MLP的参数
44. self.params = self.hiddenLayer.params + self.logRegressionLayer.params
45. # end-snippet-3

MLP类里面除了隐含层和分类层，还定义了损失函数、规则化项，这是在求解优化算法时用到的。

### （3）将MLP应用于MNIST（手写数字识别）

上面定义好了一个三层的MLP，接下来使用它在MNIST数据集上分类，MNIST是一个手写数字0～9的数据集。

首先定义加载数据 [mnist.pkl.gz](http://deeplearning.net/data/mnist/mnist.pkl.gz) 的函数load\_data()：

**[python]** [view plaincopy](http://blog.csdn.net/u012162613/article/details/43221829)

1. """
2. 加载MNIST数据集
3. """
4. **def** load\_data(dataset):
5. # dataset是数据集的路径，程序首先检测该路径下有没有MNIST数据集，没有的话就下载MNIST数据集
6. #这一部分就不解释了，与softmax回归算法无关。
7. data\_dir, data\_file = os.path.split(dataset)
8. **if** data\_dir == "" **and** **not** os.path.isfile(dataset):
9. # Check if dataset is in the data directory.
10. new\_path = os.path.join(
11. os.path.split(\_\_file\_\_)[0],
12. "..",
13. "data",
14. dataset
15. )
16. **if** os.path.isfile(new\_path) **or** data\_file == 'mnist.pkl.gz':
17. dataset = new\_path
19. **if** (**not** os.path.isfile(dataset)) **and** data\_file == 'mnist.pkl.gz':
20. **import** urllib
21. origin = (
22. 'http://www.iro.umontreal.ca/~lisa/deep/data/mnist/mnist.pkl.gz'
23. )
24. **print** 'Downloading data from %s' % origin
25. urllib.urlretrieve(origin, dataset)
27. **print** '... loading data'
28. #以上是检测并下载数据集mnist.pkl.gz，不是本文重点。下面才是load\_data的开始
30. #从"mnist.pkl.gz"里加载train\_set, valid\_set, test\_set，它们都是包括label的
31. #主要用到python里的gzip.open()函数,以及 cPickle.load()。
32. #‘rb’表示以二进制可读的方式打开文件
33. f = gzip.open(dataset, 'rb')
34. train\_set, valid\_set, test\_set = cPickle.load(f)
35. f.close()

38. #将数据设置成shared variables，主要时为了GPU加速，只有shared variables才能存到GPU memory中
39. #GPU里数据类型只能是float。而data\_y是类别，所以最后又转换为int返回
40. **def** shared\_dataset(data\_xy, borrow=True):
41. data\_x, data\_y = data\_xy
42. shared\_x = theano.shared(numpy.asarray(data\_x,
43. dtype=theano.config.floatX),
44. borrow=borrow)
45. shared\_y = theano.shared(numpy.asarray(data\_y,
46. dtype=theano.config.floatX),
47. borrow=borrow)
48. **return** shared\_x, T.cast(shared\_y, 'int32')

51. test\_set\_x, test\_set\_y = shared\_dataset(test\_set)
52. valid\_set\_x, valid\_set\_y = shared\_dataset(valid\_set)
53. train\_set\_x, train\_set\_y = shared\_dataset(train\_set)
55. rval = [(train\_set\_x, train\_set\_y), (valid\_set\_x, valid\_set\_y),
56. (test\_set\_x, test\_set\_y)]
57. **return** rval

加载了数据，可以开始训练这个模型了，以下就是主体函数test\_mlp()，将MLP用在MNIST上：

**[python]** [view plaincopy](http://blog.csdn.net/u012162613/article/details/43221829)

1. #test\_mlp是一个应用实例，用梯度下降来优化MLP，针对MNIST数据集
2. **def** test\_mlp(learning\_rate=0.01, L1\_reg=0.00, L2\_reg=0.0001, n\_epochs=10,
3. dataset='mnist.pkl.gz', batch\_size=20, n\_hidden=500):
4. """
5. 注释：
6. learning\_rate学习速率，梯度前的系数。
7. L1\_reg、L2\_reg：正则化项前的系数，权衡正则化项与Nll项的比重
8. 代价函数=Nll+L1\_reg\*L1或者L2\_reg\*L2\_sqr
9. n\_epochs：迭代的最大次数（即训练步数），用于结束优化过程
10. dataset：训练数据的路径
11. n\_hidden:隐藏层神经元个数
12. batch\_size=20，即每训练完20个样本才计算梯度并更新参数
13. """
15. #加载数据集，并分为训练集、验证集、测试集。
16. datasets = load\_data(dataset)
17. train\_set\_x, train\_set\_y = datasets[0]
18. valid\_set\_x, valid\_set\_y = datasets[1]
19. test\_set\_x, test\_set\_y = datasets[2]

22. #shape[0]获得行数，一行代表一个样本，故获取的是样本数，除以batch\_size可以得到有多少个batch
23. n\_train\_batches = train\_set\_x.get\_value(borrow=True).shape[0] / batch\_size
24. n\_valid\_batches = valid\_set\_x.get\_value(borrow=True).shape[0] / batch\_size
25. n\_test\_batches = test\_set\_x.get\_value(borrow=True).shape[0] / batch\_size
27. ######################
28. # BUILD ACTUAL MODEL #
29. ######################
30. **print** '... building the model'
32. #index表示batch的下标，标量
33. #x表示数据集
34. #y表示类别，一维向量
35. index = T.lscalar()
36. x = T.matrix('x')
37. y = T.ivector('y')

40. rng = numpy.random.RandomState(1234)
41. #生成一个MLP，命名为classifier
42. classifier = MLP(
43. rng=rng,
44. input=x,
45. n\_in=28 \* 28,
46. n\_hidden=n\_hidden,
47. n\_out=10
48. )
50. #代价函数，有规则化项
51. #用y来初始化，而其实还有一个隐含的参数x在classifier中
52. cost = (
53. classifier.negative\_log\_likelihood(y)
54. + L1\_reg \* classifier.L1
55. + L2\_reg \* classifier.L2\_sqr
56. )

59. #这里必须说明一下theano的function函数，givens是字典，其中的x、y是key，冒号后面是它们的value。
60. #在function被调用时，x、y将被具体地替换为它们的value，而value里的参数index就是inputs=[index]这里给出。
61. #下面举个例子：
62. #比如test\_model(1)，首先根据index=1具体化x为test\_set\_x[1 \* batch\_size: (1 + 1) \* batch\_size]，
63. #具体化y为test\_set\_y[1 \* batch\_size: (1 + 1) \* batch\_size]。然后函数计算outputs=classifier.errors(y)，
64. #这里面有参数y和隐含的x，所以就将givens里面具体化的x、y传递进去。
65. test\_model = theano.function(
66. inputs=[index],
67. outputs=classifier.errors(y),
68. givens={
69. x: test\_set\_x[index \* batch\_size:(index + 1) \* batch\_size],
70. y: test\_set\_y[index \* batch\_size:(index + 1) \* batch\_size]
71. }
72. )
74. validate\_model = theano.function(
75. inputs=[index],
76. outputs=classifier.errors(y),
77. givens={
78. x: valid\_set\_x[index \* batch\_size:(index + 1) \* batch\_size],
79. y: valid\_set\_y[index \* batch\_size:(index + 1) \* batch\_size]
80. }
81. )
83. #cost函数对各个参数的偏导数值，即梯度，存于gparams
84. gparams = [T.grad(cost, param) **for** param **in** classifier.params]
86. #参数更新规则
87. #updates[(),(),()....],每个括号里面都是(param, param - learning\_rate \* gparam)，即每个参数以及它的更新公式
88. updates = [
89. (param, param - learning\_rate \* gparam)
90. **for** param, gparam **in** zip(classifier.params, gparams)
91. ]
93. train\_model = theano.function(
94. inputs=[index],
95. outputs=cost,
96. updates=updates,
97. givens={
98. x: train\_set\_x[index \* batch\_size: (index + 1) \* batch\_size],
99. y: train\_set\_y[index \* batch\_size: (index + 1) \* batch\_size]
100. }
101. )

104. ###############
105. # 开始训练模型 #
106. ###############
107. **print** '... training'


111. patience = 10000
112. patience\_increase = 2
113. #提高的阈值，在验证误差减小到之前的0.995倍时，会更新best\_validation\_loss
114. improvement\_threshold = 0.995
115. #这样设置validation\_frequency可以保证每一次epoch都会在验证集上测试。
116. validation\_frequency = min(n\_train\_batches, patience / 2)

119. best\_validation\_loss = numpy.inf
120. best\_iter = 0
121. test\_score = 0.
122. start\_time = time.clock()
124. #epoch即训练步数，每个epoch都会遍历所有训练数据
125. epoch = 0
126. done\_looping = False

129. #下面就是训练过程了，while循环控制的时步数epoch，一个epoch会遍历所有的batch，即所有的图片。
130. #for循环是遍历一个个batch，一次一个batch地训练。for循环体里会用train\_model(minibatch\_index)去训练模型，
131. #train\_model里面的updatas会更新各个参数。
132. #for循环里面会累加训练过的batch数iter，当iter是validation\_frequency倍数时则会在验证集上测试，
133. #如果验证集的损失this\_validation\_loss小于之前最佳的损失best\_validation\_loss，
134. #则更新best\_validation\_loss和best\_iter，同时在testset上测试。
135. #如果验证集的损失this\_validation\_loss小于best\_validation\_loss\*improvement\_threshold时则更新patience。
136. #当达到最大步数n\_epoch时，或者patience<iter时，结束训练
137. **while** (epoch < n\_epochs) **and** (**not** done\_looping):
138. epoch = epoch + 1
139. **for** minibatch\_index **in** xrange(n\_train\_batches):#训练时一个batch一个batch进行的
141. minibatch\_avg\_cost = train\_model(minibatch\_index)
142. # 已训练过的minibatch数，即迭代次数iter
143. iter = (epoch - 1) \* n\_train\_batches + minibatch\_index
144. #训练过的minibatch数是validation\_frequency倍数，则进行交叉验证
145. **if** (iter + 1) % validation\_frequency == 0:
146. # compute zero-one loss on validation set
147. validation\_losses = [validate\_model(i) **for** i
148. **in** xrange(n\_valid\_batches)]
149. this\_validation\_loss = numpy.mean(validation\_losses)
151. **print**(
152. 'epoch %i, minibatch %i/%i, validation error %f %%' %
153. (
154. epoch,
155. minibatch\_index + 1,
156. n\_train\_batches,
157. this\_validation\_loss \* 100.
158. )
159. )
160. #当前验证误差比之前的都小，则更新best\_validation\_loss，以及对应的best\_iter，并且在tsetdata上进行test
161. **if** this\_validation\_loss < best\_validation\_loss:
162. **if** (
163. this\_validation\_loss < best\_validation\_loss \*
164. improvement\_threshold
165. ):
166. patience = max(patience, iter \* patience\_increase)
168. best\_validation\_loss = this\_validation\_loss
169. best\_iter = iter
171. test\_losses = [test\_model(i) **for** i
172. **in** xrange(n\_test\_batches)]
173. test\_score = numpy.mean(test\_losses)
175. **print**(('     epoch %i, minibatch %i/%i, test error of '
176. 'best model %f %%') %
177. (epoch, minibatch\_index + 1, n\_train\_batches,
178. test\_score \* 100.))
179. #patience小于等于iter，则终止训练
180. **if** patience <= iter:
181. done\_looping = True
182. **break**
184. end\_time = time.clock()
185. **print**(('Optimization complete. Best validation score of %f %% '
186. 'obtained at iteration %i, with test performance %f %%') %
187. (best\_validation\_loss \* 100., best\_iter + 1, test\_score \* 100.))
188. **print** >> sys.stderr, ('The code for file ' +
189. os.path.split(\_\_file\_\_)[1] +
190. ' ran for %.2fm' % ((end\_time - start\_time) / 60.))

文章完，经详细注释的代码：[放在我的github地址上，可下载](https://github.com/wepe/MachineLearning-Demo/tree/master/DeepLearning%20Tutorials/mlp)。

# [DeepLearning tutorial（4）CNN卷积神经网络原理简介+代码详解](http://blog.csdn.net/u012162613/article/details/43225445)

**DeepLearning tutorial（4）CNN卷积神经网络原理简介+代码详解**

**@author：wepon**

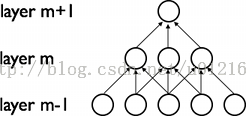
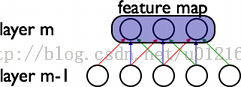
**@blog：**[**http://blog.csdn.net/u012162613/article/details/43225445**](http://blog.csdn.net/u012162613/article/details/43225445)

本文介绍多层感知机算法，特别是详细解读其代码实现，基于python theano，代码来自：[Convolutional Neural Networks (LeNet)](http://deeplearning.net/tutorial/lenet.html#lenet)。经详细注释的代码和原始代码：[放在我的github地址上，可下载](https://github.com/wepe/MachineLearning-Demo/tree/master/DeepLearning%20Tutorials/cnn_LeNet)。

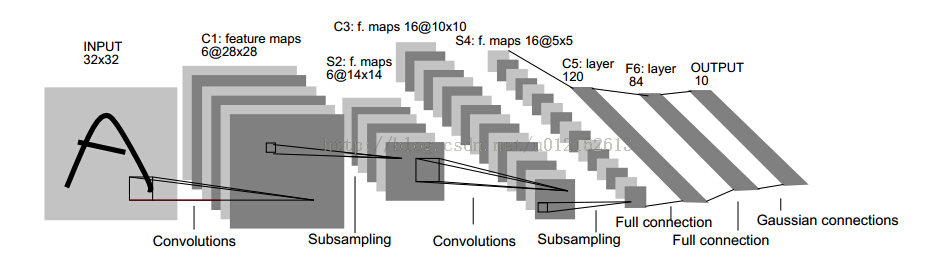
## 一、CNN卷积神经网络原理简介

要讲明白卷积神经网络，估计得长篇大论，网上有很多博文已经写得很好了，所以本文就不重复了，如果你了解CNN，那可以往下看，本文主要是详细地解读CNN的实现代码。如果你没学习过CNN，在此推荐周晓艺师兄的博文：[Deep Learning（深度学习）学习笔记整理系列之（七）](http://blog.csdn.net/zouxy09/article/details/8781543)，以及UFLDL上的[卷积特征提取](http://deeplearning.stanford.edu/wiki/index.php/%E5%8D%B7%E7%A7%AF%E7%89%B9%E5%BE%81%E6%8F%90%E5%8F%96)、[池化](http://deeplearning.stanford.edu/wiki/index.php/%E6%B1%A0%E5%8C%96)

CNN的最大特点就是稀疏连接（局部感受）和权值共享，如下面两图所示，左为稀疏连接，右为权值共享。稀疏连接和权值共享可以减少所要训练的参数，减少计算复杂度。

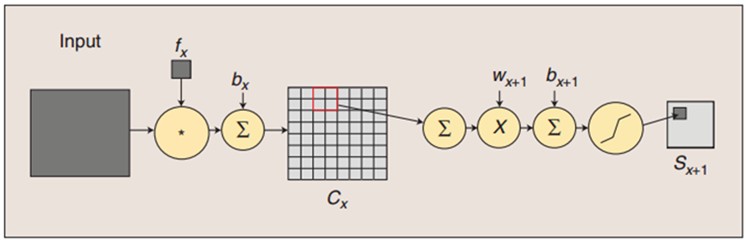
       

至于CNN的结构，以经典的LeNet5来说明：



这个图真是无处不在，一谈CNN，必说LeNet5，这图来自于这篇论文：[Gradient-Based Learning Applied to Document Recognition](http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf)，论文很长，第7页那里开始讲LeNet5这个结构，建议看看那部分。

我这里简单说一下，LeNet5这张图从左到右，先是input，这是输入层，即输入的图片。input-layer到C1这部分就是一个卷积层（convolution运算），C1到S2是一个子采样层（pooling运算），关于卷积和子采样的具体过程可以参考下图：



然后，S2到C3又是卷积，C3到S4又是子采样，可以发现，卷积和子采样都是成对出现的，卷积后面一般跟着子采样。S4到C5之间是全连接的，这就相当于一个MLP的隐含层了（如果你不清楚MLP，参考《[DeepLearning tutorial（3）MLP多层感知机原理简介+代码详解](http://blog.csdn.net/u012162613/article/details/43221829)》）。C5到F6同样是全连接，也是相当于一个MLP的隐含层。最后从F6到输出output，其实就是一个分类器，这一层就叫分类层。

ok，CNN的基本结构大概就是这样，由输入、卷积层、子采样层、全连接层、分类层、输出这些基本“构件”组成，一般根据具体的应用或者问题，去确定要多少卷积层和子采样层、采用什么分类器。当确定好了结构以后，如何求解层与层之间的连接参数？一般采用向前传播（FP）+向后传播（BP）的方法来训练。具体可参考上面给出的链接。

## 二、CNN卷积神经网络代码详细解读（基于python+theano）

代码来自于深度学习教程：[Convolutional Neural Networks (LeNet)](http://deeplearning.net/tutorial/lenet.html#lenet)，这个代码实现的是一个简化了的LeNet5，具体如下：

* 没有实现location-specific gain and bias parameters
* 用的是maxpooling，而不是average\_pooling
* 分类器用的是softmax，LeNet5用的是rbf
* LeNet5第二层并不是全连接的，本程序实现的是全连接

另外，代码里将卷积层和子采用层合在一起，定义为“LeNetConvPoolLayer“（卷积采样层），这好理解，因为它们总是成对出现。但是有个地方需要注意，代码中将卷积后的输出直接作为子采样层的输入，而没有加偏置b再通过sigmoid函数进行映射，即没有了下图中fx后面的bx以及sigmoid映射，也即直接由fx得到Cx。

最后，代码中第一个卷积层用的卷积核有20个，第二个卷积层用50个，而不是上面那张LeNet5图中所示的6个和16个。

了解了这些，下面看代码：

### （1）导入必要的模块

**[python]** [view plaincopy](http://blog.csdn.net/u012162613/article/details/43225445)

1. **import** cPickle
2. **import** gzip
3. **import** os
4. **import** sys
5. **import** time
7. **import** numpy
9. **import** theano
10. **import** theano.tensor as T
11. **from** theano.tensor.signal **import** downsample
12. **from** theano.tensor.nnet **import** conv

### （2）定义CNN的基本"构件"

CNN的基本构件包括卷积采样层、隐含层、分类器，如下

* 定义LeNetConvPoolLayer（卷积+采样层）

见代码注释：

**[python]** [view plaincopy](http://blog.csdn.net/u012162613/article/details/43225445)

1. """
2. 卷积+下采样合成一个层LeNetConvPoolLayer
3. rng:随机数生成器，用于初始化W
4. input:4维的向量，theano.tensor.dtensor4
5. filter\_shape:(number of filters, num input feature maps,filter height, filter width)
6. image\_shape:(batch size, num input feature maps,image height, image width)
7. poolsize: (#rows, #cols)
8. """
9. **class** LeNetConvPoolLayer(object):
10. **def** \_\_init\_\_(self, rng, input, filter\_shape, image\_shape, poolsize=(2, 2)):
12. #assert condition，condition为True，则继续往下执行，condition为False，中断程序
13. #image\_shape[1]和filter\_shape[1]都是num input feature maps，它们必须是一样的。
14. **assert** image\_shape[1] == filter\_shape[1]
15. self.input = input
17. #每个隐层神经元（即像素）与上一层的连接数为num input feature maps \* filter height \* filter width。
18. #可以用numpy.prod(filter\_shape[1:])来求得
19. fan\_in = numpy.prod(filter\_shape[1:])
21. #lower layer上每个神经元获得的梯度来自于："num output feature maps \* filter height \* filter width" /pooling size
22. fan\_out = (filter\_shape[0] \* numpy.prod(filter\_shape[2:]) /
23. numpy.prod(poolsize))
25. #以上求得fan\_in、fan\_out ，将它们代入公式，以此来随机初始化W,W就是线性卷积核
26. W\_bound = numpy.sqrt(6. / (fan\_in + fan\_out))
27. self.W = theano.shared(
28. numpy.asarray(
29. rng.uniform(low=-W\_bound, high=W\_bound, size=filter\_shape),
30. dtype=theano.config.floatX
31. ),
32. borrow=True
33. )
35. # the bias is a 1D tensor -- one bias per output feature map
36. #偏置b是一维向量，每个输出图的特征图都对应一个偏置，
37. #而输出的特征图的个数由filter个数决定，因此用filter\_shape[0]即number of filters来初始化
38. b\_values = numpy.zeros((filter\_shape[0],), dtype=theano.config.floatX)
39. self.b = theano.shared(value=b\_values, borrow=True)
41. #将输入图像与filter卷积，conv.conv2d函数
42. #卷积完没有加b再通过sigmoid，这里是一处简化。
43. conv\_out = conv.conv2d(
44. input=input,
45. filters=self.W,
46. filter\_shape=filter\_shape,
47. image\_shape=image\_shape
48. )
50. #maxpooling，最大子采样过程
51. pooled\_out = downsample.max\_pool\_2d(
52. input=conv\_out,
53. ds=poolsize,
54. ignore\_border=True
55. )
57. #加偏置，再通过tanh映射，得到卷积+子采样层的最终输出
58. #因为b是一维向量，这里用维度转换函数dimshuffle将其reshape。比如b是(10,)，
59. #则b.dimshuffle('x', 0, 'x', 'x'))将其reshape为(1,10,1,1)
60. self.output = T.tanh(pooled\_out + self.b.dimshuffle('x', 0, 'x', 'x'))
61. #卷积+采样层的参数
62. self.params = [self.W, self.b]

* 定义隐含层HiddenLayer

这个跟上一篇文章《 [DeepLearning tutorial（3）MLP多层感知机原理简介+代码详解](http://blog.csdn.net/u012162613/article/details/43221829)》中的HiddenLayer是一致的，直接拿过来：

**[python]** [view plaincopy](http://blog.csdn.net/u012162613/article/details/43225445)

1. """
2. 注释：
3. 这是定义隐藏层的类，首先明确：隐藏层的输入即input，输出即隐藏层的神经元个数。输入层与隐藏层是全连接的。
4. 假设输入是n\_in维的向量（也可以说时n\_in个神经元），隐藏层有n\_out个神经元，则因为是全连接，
5. 一共有n\_in\*n\_out个权重，故W大小时(n\_in,n\_out),n\_in行n\_out列，每一列对应隐藏层的每一个神经元的连接权重。
6. b是偏置，隐藏层有n\_out个神经元，故b时n\_out维向量。
7. rng即随机数生成器，numpy.random.RandomState，用于初始化W。
8. input训练模型所用到的所有输入，并不是MLP的输入层，MLP的输入层的神经元个数时n\_in，而这里的参数input大小是（n\_example,n\_in）,每一行一个样本，即每一行作为MLP的输入层。
9. activation:激活函数,这里定义为函数tanh
10. """
11. **class** HiddenLayer(object):
12. **def** \_\_init\_\_(self, rng, input, n\_in, n\_out, W=None, b=None,
13. activation=T.tanh):
14. self.input = input   #类HiddenLayer的input即所传递进来的input
16. """
17. 注释：
18. 代码要兼容GPU，则必须使用 dtype=theano.config.floatX,并且定义为theano.shared
19. 另外，W的初始化有个规则：如果使用tanh函数，则在-sqrt(6./(n\_in+n\_hidden))到sqrt(6./(n\_in+n\_hidden))之间均匀
20. 抽取数值来初始化W，若时sigmoid函数，则以上再乘4倍。
21. """
22. #如果W未初始化，则根据上述方法初始化。
23. #加入这个判断的原因是：有时候我们可以用训练好的参数来初始化W，见我的上一篇文章。
24. **if** W **is** None:
25. W\_values = numpy.asarray(
26. rng.uniform(
27. low=-numpy.sqrt(6. / (n\_in + n\_out)),
28. high=numpy.sqrt(6. / (n\_in + n\_out)),
29. size=(n\_in, n\_out)
30. ),
31. dtype=theano.config.floatX
32. )
33. **if** activation == theano.tensor.nnet.sigmoid:
34. W\_values \*= 4
35. W = theano.shared(value=W\_values, name='W', borrow=True)
37. **if** b **is** None:
38. b\_values = numpy.zeros((n\_out,), dtype=theano.config.floatX)
39. b = theano.shared(value=b\_values, name='b', borrow=True)
41. #用上面定义的W、b来初始化类HiddenLayer的W、b
42. self.W = W
43. self.b = b
45. #隐含层的输出
46. lin\_output = T.dot(input, self.W) + self.b
47. self.output = (
48. lin\_output **if** activation **is** None
49. **else** activation(lin\_output)
50. )
52. #隐含层的参数
53. self.params = [self.W, self.b]

* 定义分类器 （Softmax回归）

采用Softmax，这跟《[DeepLearning tutorial（1）Softmax回归原理简介+代码详解](http://blog.csdn.net/u012162613/article/details/43157801)》中的LogisticRegression是一样的，直接拿过来：

**[python]** [view plaincopy](http://blog.csdn.net/u012162613/article/details/43225445)

1. """
2. 定义分类层LogisticRegression，也即Softmax回归
3. 在deeplearning tutorial中，直接将LogisticRegression视为Softmax，
4. 而我们所认识的二类别的逻辑回归就是当n\_out=2时的LogisticRegression
5. """
6. #参数说明：
7. #input，大小就是(n\_example,n\_in)，其中n\_example是一个batch的大小，
8. #因为我们训练时用的是Minibatch SGD，因此input这样定义
9. #n\_in,即上一层(隐含层)的输出
10. #n\_out,输出的类别数
11. **class** LogisticRegression(object):
12. **def** \_\_init\_\_(self, input, n\_in, n\_out):
14. #W大小是n\_in行n\_out列，b为n\_out维向量。即：每个输出对应W的一列以及b的一个元素。
15. self.W = theano.shared(
16. value=numpy.zeros(
17. (n\_in, n\_out),
18. dtype=theano.config.floatX
19. ),
20. name='W',
21. borrow=True
22. )
24. self.b = theano.shared(
25. value=numpy.zeros(
26. (n\_out,),
27. dtype=theano.config.floatX
28. ),
29. name='b',
30. borrow=True
31. )
33. #input是(n\_example,n\_in)，W是（n\_in,n\_out）,点乘得到(n\_example,n\_out)，加上偏置b，
34. #再作为T.nnet.softmax的输入，得到p\_y\_given\_x
35. #故p\_y\_given\_x每一行代表每一个样本被估计为各类别的概率
36. #PS：b是n\_out维向量，与(n\_example,n\_out)矩阵相加，内部其实是先复制n\_example个b，
37. #然后(n\_example,n\_out)矩阵的每一行都加b
38. self.p\_y\_given\_x = T.nnet.softmax(T.dot(input, self.W) + self.b)
40. #argmax返回最大值下标，因为本例数据集是MNIST，下标刚好就是类别。axis=1表示按行操作。
41. self.y\_pred = T.argmax(self.p\_y\_given\_x, axis=1)
43. #params，LogisticRegression的参数
44. self.params = [self.W, self.b]

到这里，CNN的基本”构件“都有了，下面要用这些”构件“组装成LeNet5（当然，是简化的，上面已经说了），具体来说，就是组装成：LeNet5=input+LeNetConvPoolLayer\_1+LeNetConvPoolLayer\_2+HiddenLayer+LogisticRegression+output。

然后将其应用于MNIST数据集，用BP算法去解这个模型，得到最优的参数。

### （3）加载MNIST数据集（[mnist.pkl.gz](http://deeplearning.net/data/mnist/mnist.pkl.gz)）

**[python]** [view plaincopy](http://blog.csdn.net/u012162613/article/details/43225445)

1. """
2. 加载MNIST数据集load\_data()
3. """
4. **def** load\_data(dataset):
5. # dataset是数据集的路径，程序首先检测该路径下有没有MNIST数据集，没有的话就下载MNIST数据集
6. #这一部分就不解释了，与softmax回归算法无关。
7. data\_dir, data\_file = os.path.split(dataset)
8. **if** data\_dir == "" **and** **not** os.path.isfile(dataset):
9. # Check if dataset is in the data directory.
10. new\_path = os.path.join(
11. os.path.split(\_\_file\_\_)[0],
12. "..",
13. "data",
14. dataset
15. )
16. **if** os.path.isfile(new\_path) **or** data\_file == 'mnist.pkl.gz':
17. dataset = new\_path
19. **if** (**not** os.path.isfile(dataset)) **and** data\_file == 'mnist.pkl.gz':
20. **import** urllib
21. origin = (
22. 'http://www.iro.umontreal.ca/~lisa/deep/data/mnist/mnist.pkl.gz'
23. )
24. **print** 'Downloading data from %s' % origin
25. urllib.urlretrieve(origin, dataset)
27. **print** '... loading data'
28. #以上是检测并下载数据集mnist.pkl.gz，不是本文重点。下面才是load\_data的开始
30. #从"mnist.pkl.gz"里加载train\_set, valid\_set, test\_set，它们都是包括label的
31. #主要用到python里的gzip.open()函数,以及 cPickle.load()。
32. #‘rb’表示以二进制可读的方式打开文件
33. f = gzip.open(dataset, 'rb')
34. train\_set, valid\_set, test\_set = cPickle.load(f)
35. f.close()

38. #将数据设置成shared variables，主要时为了GPU加速，只有shared variables才能存到GPU memory中
39. #GPU里数据类型只能是float。而data\_y是类别，所以最后又转换为int返回
40. **def** shared\_dataset(data\_xy, borrow=True):
41. data\_x, data\_y = data\_xy
42. shared\_x = theano.shared(numpy.asarray(data\_x,
43. dtype=theano.config.floatX),
44. borrow=borrow)
45. shared\_y = theano.shared(numpy.asarray(data\_y,
46. dtype=theano.config.floatX),
47. borrow=borrow)
48. **return** shared\_x, T.cast(shared\_y, 'int32')

51. test\_set\_x, test\_set\_y = shared\_dataset(test\_set)
52. valid\_set\_x, valid\_set\_y = shared\_dataset(valid\_set)
53. train\_set\_x, train\_set\_y = shared\_dataset(train\_set)
55. rval = [(train\_set\_x, train\_set\_y), (valid\_set\_x, valid\_set\_y),
56. (test\_set\_x, test\_set\_y)]
57. **return** rval

### （4）实现LeNet5并测试

**[python]** [view plaincopy](http://blog.csdn.net/u012162613/article/details/43225445)

1. """
2. 实现LeNet5
3. LeNet5有两个卷积层，第一个卷积层有20个卷积核，第二个卷积层有50个卷积核
4. """
5. **def** evaluate\_lenet5(learning\_rate=0.1, n\_epochs=200,
6. dataset='mnist.pkl.gz',
7. nkerns=[20, 50], batch\_size=500):
8. """
9. learning\_rate:学习速率，随机梯度前的系数。
10. n\_epochs训练步数，每一步都会遍历所有batch，即所有样本
11. batch\_size,这里设置为500，即每遍历完500个样本，才计算梯度并更新参数
12. nkerns=[20, 50],每一个LeNetConvPoolLayer卷积核的个数，第一个LeNetConvPoolLayer有
13. 20个卷积核，第二个有50个
14. """
16. rng = numpy.random.RandomState(23455)
18. #加载数据
19. datasets = load\_data(dataset)
20. train\_set\_x, train\_set\_y = datasets[0]
21. valid\_set\_x, valid\_set\_y = datasets[1]
22. test\_set\_x, test\_set\_y = datasets[2]
24. # 计算batch的个数
25. n\_train\_batches = train\_set\_x.get\_value(borrow=True).shape[0]
26. n\_valid\_batches = valid\_set\_x.get\_value(borrow=True).shape[0]
27. n\_test\_batches = test\_set\_x.get\_value(borrow=True).shape[0]
28. n\_train\_batches /= batch\_size
29. n\_valid\_batches /= batch\_size
30. n\_test\_batches /= batch\_size
32. #定义几个变量，index表示batch下标，x表示输入的训练数据，y对应其标签
33. index = T.lscalar()
34. x = T.matrix('x')
35. y = T.ivector('y')
37. ######################
38. # BUILD ACTUAL MODEL #
39. ######################
40. **print** '... building the model'

43. #我们加载进来的batch大小的数据是(batch\_size, 28 \* 28)，但是LeNetConvPoolLayer的输入是四维的，所以要reshape
44. layer0\_input = x.reshape((batch\_size, 1, 28, 28))
46. # layer0即第一个LeNetConvPoolLayer层
47. #输入的单张图片(28,28)，经过conv得到(28-5+1 , 28-5+1) = (24, 24)，
48. #经过maxpooling得到(24/2, 24/2) = (12, 12)
49. #因为每个batch有batch\_size张图，第一个LeNetConvPoolLayer层有nkerns[0]个卷积核，
50. #故layer0输出为(batch\_size, nkerns[0], 12, 12)
51. layer0 = LeNetConvPoolLayer(
52. rng,
53. input=layer0\_input,
54. image\_shape=(batch\_size, 1, 28, 28),
55. filter\_shape=(nkerns[0], 1, 5, 5),
56. poolsize=(2, 2)
57. )

60. #layer1即第二个LeNetConvPoolLayer层
61. #输入是layer0的输出，每张特征图为(12,12),经过conv得到(12-5+1, 12-5+1) = (8, 8),
62. #经过maxpooling得到(8/2, 8/2) = (4, 4)
63. #因为每个batch有batch\_size张图（特征图），第二个LeNetConvPoolLayer层有nkerns[1]个卷积核
64. #，故layer1输出为(batch\_size, nkerns[1], 4, 4)
65. layer1 = LeNetConvPoolLayer(
66. rng,
67. input=layer0.output,
68. image\_shape=(batch\_size, nkerns[0], 12, 12),#输入nkerns[0]张特征图，即layer0输出nkerns[0]张特征图
69. filter\_shape=(nkerns[1], nkerns[0], 5, 5),
70. poolsize=(2, 2)
71. )

74. #前面定义好了两个LeNetConvPoolLayer（layer0和layer1），layer1后面接layer2，这是一个全连接层，相当于MLP里面的隐含层
75. #故可以用MLP中定义的HiddenLayer来初始化layer2，layer2的输入是二维的(batch\_size, num\_pixels) ，
76. #故要将上层中同一张图经不同卷积核卷积出来的特征图合并为一维向量，
77. #也就是将layer1的输出(batch\_size, nkerns[1], 4, 4)flatten为(batch\_size, nkerns[1]\*4\*4)=(500，800),作为layer2的输入。
78. #(500，800)表示有500个样本，每一行代表一个样本。layer2的输出大小是(batch\_size,n\_out)=(500,500)
79. layer2\_input = layer1.output.flatten(2)
80. layer2 = HiddenLayer(
81. rng,
82. input=layer2\_input,
83. n\_in=nkerns[1] \* 4 \* 4,
84. n\_out=500,
85. activation=T.tanh
86. )
88. #最后一层layer3是分类层，用的是逻辑回归中定义的LogisticRegression，
89. #layer3的输入是layer2的输出(500,500)，layer3的输出就是(batch\_size,n\_out)=(500,10)
90. layer3 = LogisticRegression(input=layer2.output, n\_in=500, n\_out=10)
92. #代价函数NLL
93. cost = layer3.negative\_log\_likelihood(y)
95. # test\_model计算测试误差，x、y根据给定的index具体化，然后调用layer3，
96. #layer3又会逐层地调用layer2、layer1、layer0，故test\_model其实就是整个CNN结构，
97. #test\_model的输入是x、y，输出是layer3.errors(y)的输出，即误差。
98. test\_model = theano.function(
99. [index],
100. layer3.errors(y),
101. givens={
102. x: test\_set\_x[index \* batch\_size: (index + 1) \* batch\_size],
103. y: test\_set\_y[index \* batch\_size: (index + 1) \* batch\_size]
104. }
105. )
106. #validate\_model，验证模型，分析同上。
107. validate\_model = theano.function(
108. [index],
109. layer3.errors(y),
110. givens={
111. x: valid\_set\_x[index \* batch\_size: (index + 1) \* batch\_size],
112. y: valid\_set\_y[index \* batch\_size: (index + 1) \* batch\_size]
113. }
114. )
116. #下面是train\_model，涉及到优化算法即SGD，需要计算梯度、更新参数
117. #参数集
118. params = layer3.params + layer2.params + layer1.params + layer0.params
120. #对各个参数的梯度
121. grads = T.grad(cost, params)
123. #因为参数太多，在updates规则里面一个一个具体地写出来是很麻烦的，所以下面用了一个for..in..,自动生成规则对(param\_i, param\_i - learning\_rate \* grad\_i)
124. updates = [
125. (param\_i, param\_i - learning\_rate \* grad\_i)
126. **for** param\_i, grad\_i **in** zip(params, grads)
127. ]
129. #train\_model，代码分析同test\_model。train\_model里比test\_model、validation\_model多出updates规则
130. train\_model = theano.function(
131. [index],
132. cost,
133. updates=updates,
134. givens={
135. x: train\_set\_x[index \* batch\_size: (index + 1) \* batch\_size],
136. y: train\_set\_y[index \* batch\_size: (index + 1) \* batch\_size]
137. }
138. )

141. ###############
142. #   开始训练  #
143. ###############
144. **print** '... training'
145. patience = 10000
146. patience\_increase = 2
147. improvement\_threshold = 0.995
149. validation\_frequency = min(n\_train\_batches, patience / 2)
150. #这样设置validation\_frequency可以保证每一次epoch都会在验证集上测试。
152. best\_validation\_loss = numpy.inf   #最好的验证集上的loss，最好即最小
153. best\_iter = 0                      #最好的迭代次数，以batch为单位。比如best\_iter=10000，说明在训练完第10000个batch时，达到best\_validation\_loss
154. test\_score = 0.
155. start\_time = time.clock()
157. epoch = 0
158. done\_looping = False
160. #下面就是训练过程了，while循环控制的时步数epoch，一个epoch会遍历所有的batch，即所有的图片。
161. #for循环是遍历一个个batch，一次一个batch地训练。for循环体里会用train\_model(minibatch\_index)去训练模型，
162. #train\_model里面的updatas会更新各个参数。
163. #for循环里面会累加训练过的batch数iter，当iter是validation\_frequency倍数时则会在验证集上测试，
164. #如果验证集的损失this\_validation\_loss小于之前最佳的损失best\_validation\_loss，
165. #则更新best\_validation\_loss和best\_iter，同时在testset上测试。
166. #如果验证集的损失this\_validation\_loss小于best\_validation\_loss\*improvement\_threshold时则更新patience。
167. #当达到最大步数n\_epoch时，或者patience<iter时，结束训练
168. **while** (epoch < n\_epochs) **and** (**not** done\_looping):
169. epoch = epoch + 1
170. **for** minibatch\_index **in** xrange(n\_train\_batches):
172. iter = (epoch - 1) \* n\_train\_batches + minibatch\_index
174. **if** iter % 100 == 0:
175. **print** 'training @ iter = ', iter
176. cost\_ij = train\_model(minibatch\_index)
177. #cost\_ij 没什么用，后面都没有用到,只是为了调用train\_model，而train\_model有返回值
178. **if** (iter + 1) % validation\_frequency == 0:
180. # compute zero-one loss on validation set
181. validation\_losses = [validate\_model(i) **for** i
182. **in** xrange(n\_valid\_batches)]
183. this\_validation\_loss = numpy.mean(validation\_losses)
184. **print**('epoch %i, minibatch %i/%i, validation error %f %%' %
185. (epoch, minibatch\_index + 1, n\_train\_batches,
186. this\_validation\_loss \* 100.))

189. **if** this\_validation\_loss < best\_validation\_loss:

192. **if** this\_validation\_loss < best\_validation\_loss \*  \
193. improvement\_threshold:
194. patience = max(patience, iter \* patience\_increase)

197. best\_validation\_loss = this\_validation\_loss
198. best\_iter = iter

201. test\_losses = [
202. test\_model(i)
203. **for** i **in** xrange(n\_test\_batches)
204. ]
205. test\_score = numpy.mean(test\_losses)
206. **print**(('     epoch %i, minibatch %i/%i, test error of '
207. 'best model %f %%') %
208. (epoch, minibatch\_index + 1, n\_train\_batches,
209. test\_score \* 100.))
211. **if** patience <= iter:
212. done\_looping = True
213. **break**
215. end\_time = time.clock()
216. **print**('Optimization complete.')
217. **print**('Best validation score of %f %% obtained at iteration %i, '
218. 'with test performance %f %%' %
219. (best\_validation\_loss \* 100., best\_iter + 1, test\_score \* 100.))
220. **print** >> sys.stderr, ('The code for file ' +
221. os.path.split(\_\_file\_\_)[1] +
222. ' ran for %.2fm' % ((end\_time - start\_time) / 60.))

文章完，经详细注释的代码和原始代码：[放在我的github地址上，可下载](https://github.com/wepe/MachineLearning-Demo/tree/master/DeepLearning%20Tutorials/cnn_LeNet)。

[DeepLearning tutorial（5）CNN卷积神经网络应用于人脸识别（详细流程+代码实现）](http://blog.csdn.net/u012162613/article/details/43277187)

@author：wepon

@blog：<http://blog.csdn.net/u012162613/article/details/43277187>

本文代码下载地址：[我的github](https://github.com/wepe/MachineLearning-Demo/tree/master/DeepLearning%20Tutorials/FaceRecognition_CNN(olivettifaces))

本文主要讲解将CNN应用于人脸识别的流程，程序基于python+numpy+theano+PIL开发，采用类似LeNet5的CNN模型，应用于olivettifaces人脸数据库，实现人脸识别的功能，模型的误差降到了5%以下。本程序只是个人学习过程的一个toy implement，样本很小，模型随时都会过拟合。

但是，本文意在理清程序开发CNN模型的具体步骤，特别是针对图像识别，从拿到图像数据库，到实现一个针对这个图像数据库的CNN模型，我觉得本文对这些流程的实现具有参考意义。

《本文目录》

一、olivettifaces人脸数据库介绍

二、CNN的基本“构件”（LogisticRegression、HiddenLayer、LeNetConvPoolLayer）

三、组建CNN模型，设置优化算法，应用于Olivetti Faces进行人脸识别

四、训练结果以及参数设置的讨论

五、利用训练好的参数初始化模型

六、一些需要说明的

**一、olivettifaces人脸数据库介绍**

[Olivetti Faces](http://www.cs.nyu.edu/~roweis/data/olivettifaces.gif)是纽约大学的一个比较小的人脸库，由40个人的400张图片构成，即每个人的人脸图片为10张。每张图片的灰度级为8位，每个像素的灰度大小位于0-255之间，每张图片大小为64×64。如下图，这个图片大小是1190\*942，一共有20\*20张人脸，故每张人脸大小是（1190/20）\*（942/20）即57\*47=2679：



本文所用的训练数据就是这张图片，400个样本，40个类别，乍一看样本好像比较小，用CNN效果会好吗？先别下结论，请往下看。

要运行CNN算法，这张图片必须先转化为数组（或者说矩阵），这个用到python的图像库PIL，几行代码就可以搞定，具体的方法我之前刚好写过一篇文章，也是用这张图，考虑到文章冗长，就不复制过来了，链接在此：[《利用Python PIL、cPickle读取和保存图像数据库》](http://blog.csdn.net/u012162613/article/details/43226127)。

训练机器学习算法，我们一般将原始数据分成训练数据（training\_set）、验证数据(validation\_set)、测试数据(testing\_set)。本程序将training\_set、validation\_set、testing\_set分别设置为320、40、40个样本。它们的label为0～39，对应40个不同的人。这部分的代码如下：

**[python]** [view plaincopy](http://blog.csdn.net/u012162613/article/details/43277187)

1. """
2. 加载图像数据的函数,dataset\_path即图像olivettifaces的路径
3. 加载olivettifaces后，划分为train\_data,valid\_data,test\_data三个数据集
4. 函数返回train\_data,valid\_data,test\_data以及对应的label
5. """
6. **def** load\_data(dataset\_path):
7. img = Image.open(dataset\_path)
8. img\_ndarray = numpy.asarray(img, dtype='float64')/256
9. faces=numpy.empty((400,2679))
10. **for** row **in** range(20):
11. **for** column **in** range(20):
12. faces[row\*20+column]=numpy.ndarray.flatten(img\_ndarray [row\*57:(row+1)\*57,column\*47:(column+1)\*47])
14. label=numpy.empty(400)
15. **for** i **in** range(40):
16. label[i\*10:i\*10+10]=i
17. label=label.astype(numpy.int)
19. #分成训练集、验证集、测试集，大小如下
20. train\_data=numpy.empty((320,2679))
21. train\_label=numpy.empty(320)
22. valid\_data=numpy.empty((40,2679))
23. valid\_label=numpy.empty(40)
24. test\_data=numpy.empty((40,2679))
25. test\_label=numpy.empty(40)
27. **for** i **in** range(40):
28. train\_data[i\*8:i\*8+8]=faces[i\*10:i\*10+8]
29. train\_label[i\*8:i\*8+8]=label[i\*10:i\*10+8]
30. valid\_data[i]=faces[i\*10+8]
31. valid\_label[i]=label[i\*10+8]
32. test\_data[i]=faces[i\*10+9]
33. test\_label[i]=label[i\*10+9]
35. #将数据集定义成shared类型，才能将数据复制进GPU，利用GPU加速程序。
36. **def** shared\_dataset(data\_x, data\_y, borrow=True):
37. shared\_x = theano.shared(numpy.asarray(data\_x,
38. dtype=theano.config.floatX),
39. borrow=borrow)
40. shared\_y = theano.shared(numpy.asarray(data\_y,
41. dtype=theano.config.floatX),
42. borrow=borrow)
43. **return** shared\_x, T.cast(shared\_y, 'int32')


47. train\_set\_x, train\_set\_y = shared\_dataset(train\_data,train\_label)
48. valid\_set\_x, valid\_set\_y = shared\_dataset(valid\_data,valid\_label)
49. test\_set\_x, test\_set\_y = shared\_dataset(test\_data,test\_label)
50. rval = [(train\_set\_x, train\_set\_y), (valid\_set\_x, valid\_set\_y),
51. (test\_set\_x, test\_set\_y)]
52. **return** rval

**二、CNN的基本“构件”（LogisticRegression、HiddenLayer、LeNetConvPoolLayer）**

卷积神经网络（CNN）的基本结构就是输入层、卷积层（conv）、子采样层（pooling）、全连接层、输出层（分类器）。  卷积层+子采样层一般都会有若干个，本程序实现的CNN模型参考LeNet5，有两个“卷积+子采样层”LeNetConvPoolLayer。全连接层相当于MLP（多层感知机）中的隐含层HiddenLayer。输出层即分类器，一般采用softmax回归（也有人直接叫逻辑回归，其实就是多类别的logistics regression），本程序也直接用LogisticRegression表示。

总结起来，要组建CNN模型，必须先定义LeNetConvPoolLayer、HiddenLayer、LogisticRegression这三种layer，这一点在我上一篇文章介绍CNN算法时讲得很详细，包括代码注解，因为太冗长，这里给出链接：[《DeepLearning tutorial（4）CNN卷积神经网络原理简介+代码详解》](http://blog.csdn.net/u012162613/article/details/43225445)。

代码太长，就不贴具体的了，只给出框架，具体可以下载我的代码看看：

**[python]** [view plaincopy](http://blog.csdn.net/u012162613/article/details/43277187)

1. #分类器，即CNN最后一层，采用逻辑回归（softmax）
2. **class** LogisticRegression(object):
3. **def** \_\_init\_\_(self, input, n\_in, n\_out):
4. self.W = ....
5. self.b = ....
6. self.p\_y\_given\_x = ...
7. self.y\_pred = ...
8. self.params = ...
9. **def** negative\_log\_likelihood(self, y):
10. **def** errors(self, y):
12. #全连接层，分类器前一层
13. **class** HiddenLayer(object):
14. **def** \_\_init\_\_(self, rng, input, n\_in, n\_out, W=None, b=None,activation=T.tanh):
15. self.input = input
16. self.W = ...
17. self.b = ...
18. lin\_output = ...
19. self.params = [self.W, self.b]
21. #卷积+采样层（conv+maxpooling）
22. **class** LeNetConvPoolLayer(object):
23. **def** \_\_init\_\_(self, rng, input, filter\_shape, image\_shape, poolsize=(2, 2)):
24. self.input = input
25. self.W = ...
26. self.b = ...
27. # 卷积
28. conv\_out = ...
29. # 子采样
30. pooled\_out =...
31. self.output = ...
32. self.params = [self.W, self.b]

**三、组建CNN模型，设置优化算法，应用于Olivetti Faces进行人脸识别**

上面定义好了CNN的几个基本“构件”，现在我们使用这些构件来组建CNN模型，本程序的CNN模型参考LeNet5，具体为：input+layer0(LeNetConvPoolLayer)+layer1(LeNetConvPoolLayer)+layer2(HiddenLayer)+layer3(LogisticRegression)

这是一个串联结构，代码也很好写，直接用第二部分定义好的各种layer去组建就行了，上一layer的输出接下一layer的输入，具体可以看看代码evaluate\_olivettifaces函数中的“建立CNN模型”部分。

CNN模型组建好了，就剩下用优化算法求解了，优化算法采用批量随机梯度下降算法（MSGD），所以要先定义MSGD的一些要素，主要包括：代价函数，训练、验证、测试model、参数更新规则（即梯度下降）。这部分的代码在evaluate\_olivettifaces函数中的“定义优化算法的一些基本要素”部分。

优化算法的基本要素也定义好了，接下来就要运用人脸图像数据集来训练这个模型了，训练过程有训练步数（n\_epoch）的设置，每个epoch会遍历所有的训练数据（training\_set），本程序中也就是320个人脸图。还有迭代次数iter，一次迭代遍历一个batch里的所有样本，具体为多少要看所设置的batch\_size。关于参数的设定我在下面会讨论。这一部分的代码在evaluate\_olivettifaces函数中的“训练CNN阶段”部分。

代码很长，只贴框架，具体可以下载我的代码看看：

**[python]** [view plaincopy](http://blog.csdn.net/u012162613/article/details/43277187)

1. **def** evaluate\_olivettifaces(learning\_rate=0.05, n\_epochs=200,
2. dataset='olivettifaces.gif',
3. nkerns=[5, 10], batch\_size=40):
5. #随机数生成器，用于初始化参数....
6. #加载数据.....
7. #计算各数据集的batch个数....
8. #定义几个变量，x代表人脸数据，作为layer0的输入......
10. ######################
11. #建立CNN模型:
12. #input+layer0(LeNetConvPoolLayer)+layer1(LeNetConvPoolLayer)+layer2(HiddenLayer)+layer3(LogisticRegression)
13. ######################
14. ...
15. ....
16. ......
18. #########################
19. # 定义优化算法的一些基本要素：代价函数，训练、验证、测试model、参数更新规则（即梯度下降）
20. #########################
21. ...
22. ....
23. ......
25. #########################
26. # 训练CNN阶段，寻找最优的参数。
27. ########################
28. ...
29. .....
30. .......

另外，值得一提的是，在训练CNN阶段，我们必须定时地保存模型的参数，这是在训练机器学习算法时一个经常会做的事情，这一部分的详细介绍我之前写过一篇文章[《DeepLearning tutorial（2）机器学习算法在训练过程中保存参数》](http://blog.csdn.net/u012162613/article/details/43169019)。简单来说，我们要保存CNN模型中layer0、layer1、layer2、layer3的参数，所以在“训练CNN阶段”这部分下面，有一句代码：

**[python]** [view plaincopy](http://blog.csdn.net/u012162613/article/details/43277187)

1. save\_params(layer0.params,layer1.params,layer2.params,layer3.params)

这个函数具体定义为：

**[python]** [view plaincopy](http://blog.csdn.net/u012162613/article/details/43277187)

1. #保存训练参数的函数
2. **def** save\_params(param1,param2,param3,param4):
3. **import** cPickle
4. write\_file = open('params.pkl', 'wb')
5. cPickle.dump(param1, write\_file, -1)
6. cPickle.dump(param2, write\_file, -1)
7. cPickle.dump(param3, write\_file, -1)
8. cPickle.dump(param4, write\_file, -1)
9. write\_file.close()

如果在其他算法中，你要保存的参数有五个六个甚至更多，那么改一下这个函数的参数就行啦。

**四、训练结果以及参数设置的讨论**

ok，上面基本介绍完了CNN模型的构建，以及模型的训练，我将它们的代码都放在train\_CNN\_olivettifaces.py这个源文件中，将[Olivetti Faces](http://www.cs.nyu.edu/~roweis/data/olivettifaces.gif)这张图片跟这个代码文件放在同个目录下，运行这个文件，几分钟就可以训练完模型，并且在同个目录下得到一个params.pkl文件，这个文件保存的就是最后的模型的参数，方便你以后直接使用这个模型。

好了，现在讨论一下怎么设置参数，具体来说，程序中可以设置的参数包括：学习速率learning\_rate、batch\_size、n\_epochs、nkerns、poolsize。下面逐一讨论调节它们时对模型的影响。

* 调节learning\_rate

学习速率learning\_rate就是运用SGD算法时梯度前面的系数，非常重要，设得太大的话算法可能永远都优化不了，设得太小会使算法优化得太慢，而且可能还会掉入局部最优。可以形象地将learning\_rate比喻成走路时步子的大小，想象一下要从一个U形的山谷的一边走到山谷最低点，如果步子特别大，像巨人那么大，那会直接从一边跨到另一边，然后又跨回这边，如此往复。如果太小了，可能你走着走着就掉入了某些小坑，因为山路总是凹凸不平的（局部最优），掉入这些小坑后，如果步子还是不变，就永远走不出那个坑。

好，回到本文的模型，下面是我使用时的记录，固定其他参数，调节learning\_rate：

（1）kerns=[20, 50], batch\_size=40，poolsize=（2，2），learning\_rate=0.1时，validation-error一直是97.5%，没降下来，分析了一下，觉得应该是学习速率太大，跳过了最优。  
  
（2）nkerns=[20, 50], batch\_size=40，poolsize=（2，2），learning\_rate=0.01时，训练到epoch 60多时，validation-error降到5%，test-error降到15%  
  
（3）nkerns=[20, 50], batch\_size=40，poolsize=（2，2），learning\_rate=0.05时，训练到epoch 36时，validation-error降到2.5%，test-error降到5%

注意，验证集和测试集都只有40张图片，也就是说只有一两张识别错了，还是不错的，数据集再大点，误差率可以降到更小。最后我将learning\_rate设置为0.05。  
  
PS：学习速率应该自适应地减小，是有专门的一些算法的，本程序没有实现这个功能，有时间再研究一下。

* 调节batch\_size

因为我们采用minibatch SGD算法来优化，所以是一个batch一个batch地将数据输入CNN模型中，然后计算这个batch的所有样本的平均损失，即代价函数是所有样本的平均。而batch\_size就是一个batch的所包含的样本数，显然batch\_size将影响到模型的优化程度和速度。

回到本文的模型，首先因为我们train\_dataset是320，valid\_dataset和test\_dataset都是40，所以batch\_size最好都是40的因子，也就是能让40整除，比如40、20、10、5、2、1，否则会浪费一些样本，比如设置为30，则320/30=10，余数时20，这20个样本是没被利用的。并且，如果batch\_size设置为30，则得出的validation-error和test-error只是30个样本的错误率，并不是全部40个样本的错误率。这是设置batch\_size要注意的。特别是样本比较少的时候。

下面是我实验时的记录，固定其他参数，改变batch\_size:  
batch\_size=1、2、5、10、20时，validation-error一直是97.5%，没降下来。我觉得可能是样本类别覆盖率过小，因为我们的数据是按类别排的，每个类别10个样本是连续排在一起的，batch\_size等于20时其实只包含了两个类别，这样优化会很慢。  
  
因此最后我将batch\_size设为40，也就是valid\_dataset和test\_dataset的大小了，没办法，原始数据集样本太少了。一般我们都不会让batch\_size达到valid\_dataset和test\_dataset的大小的。

* 关于n\_epochs

n\_epochs也就是最大的训练步数，比如设为200，那训练过程最多遍历你的数据集200遍，当遍历了200遍你的dataset时，程序会停止。n\_epochs就相当于一个停止程序的控制参数，并不会影响CNN模型的优化程度和速度，只是一个控制程序结束的参数。

* nkerns=[20, 50]

20表示第一个卷积层的卷积核的个数，50表示第二个卷积层的卷积核的个数。这个我也是瞎调的，暂时没什么经验可以总结。  
不过从理论上来说，卷积核的个数其实就代表了特征的个数，你提取的特征越多，可能最后分类就越准。但是，特征太多（卷积核太多），会增加参数的规模，加大了计算复杂度，而且有时候卷积核也不是越多越好，应根据具体的应用对象来确定。所以我觉得，CNN虽号称自动提取特征，免去复杂的特征工程，但是很多参数比如这里的nkerns还是需要去调节的，还是需要一些“人工”的。

下面是我的实验记录，固定batch\_size=40，learning\_rate=0.05，poolsize=（2，2）：  
  
（1）nkerns=[20, 50]时，训练到epoch 36时，validation-error降到2.5%，test-error降到5%  
  
（2）nkerns=[10, 30]时，训练到epoch 46时，validation-error降到5%，test-error降到5%  
  
（3）nkerns=[5, 10]时，训练到epoch 38时，validation-error降到5%，test-error降到7.5%

* poolsize=(2, 2)

poolzize在本程序中是设置为(2,2)，即从一个2\*2的区域里maxpooling出1个像素，说白了就算4和像素保留成1个像素。本例程中人脸图像大小是57\*47，对这种小图像来说，(2,2)时比较合理的。如果你用的图像比较大，可以把poolsize设的大一点。

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++分割线+++++++++++++++++++++++++++++++++++++++++++

上面部分介绍完了CNN模型构建以及模型训练的过程，代码都在train\_CNN\_olivettifaces.py里面，训练完可以得到一个params.pkl文件，这个文件保存的就是最后的模型的参数，方便你以后直接使用这个模型。以后只需利用这些保存下来的参数来初始化CNN模型，就得到一个可以使用的CNN系统，将人脸图输入这个CNN系统，预测人脸图的类别。

接下来就介绍怎么使用训练好的参数的方法，这部分的代码放在use\_CNN\_olivettifaces.py文件中。

**五、利用训练好的参数初始化模型**

在train\_CNN\_olivettifaces.py中的LeNetConvPoolLayer、HiddenLayer、LogisticRegression是用随机数生成器去随机初始化的，我们将它们定义为可以用参数来初始化的版本，其实很简单，代码只需要做稍微的改动，只需要在LogisticRegression、HiddenLayer、LeNetConvPoolLayer这三个class的\_\_init\_\_()函数中加两个参数params\_W,params\_b，然后将params\_W,params\_b赋值给这三个class里的W和b：

**[python]** [view plaincopy](http://blog.csdn.net/u012162613/article/details/43277187)

1. self.W = params\_W
2. self.b = params\_b

params\_W,params\_b就是从params.pkl文件中读取来的，读取的函数：

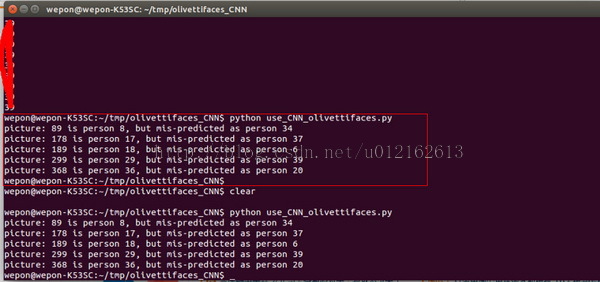
**[python]** [view plaincopy](http://blog.csdn.net/u012162613/article/details/43277187)

1. #读取之前保存的训练参数
2. #layer0\_params~layer3\_params都是包含W和b的,layer\*\_params[0]是W，layer\*\_params[1]是b
3. **def** load\_params(params\_file):
4. f=open(params\_file,'rb')
5. layer0\_params=cPickle.load(f)
6. layer1\_params=cPickle.load(f)
7. layer2\_params=cPickle.load(f)
8. layer3\_params=cPickle.load(f)
9. f.close()
10. **return** layer0\_params,layer1\_params,layer2\_params,layer3\_params

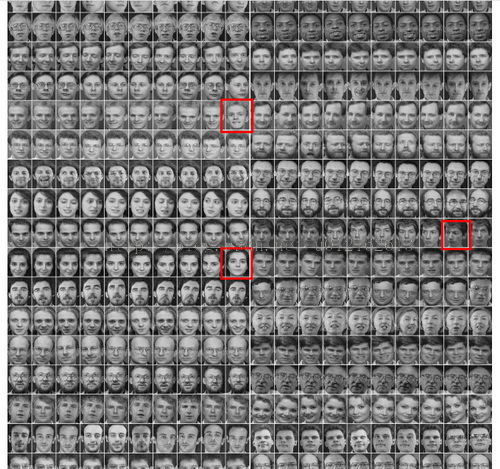
ok，可以用参数初始化的CNN定义好了，那现在就将需要测试的人脸图输入该CNN，测试其类别。同样的，需要写一个读图像的函数load\_data()，代码就不贴了。将图像数据输入，CNN的输出便是该图像的类别，这一部分的代码在use\_CNN()函数中，代码很容易看懂。

这一部分还涉及到theano.function的使用，我把一些笔记记在了use\_CNN\_olivettifaces.py代码的最后，因为跟代码相关，结合代码来看会比较好，所以下面就不讲这部分，有兴趣的看看代码。

最后说说测试的结果，我仍然以整副olivettifaces.gif作为输入，得出其类别后，跟真正的label对比，程序输出被错分的那些图像，运行结果如下：



错了五张，我标了三张：



**六、一些需要说明的**

首先是本文的严谨性：在文章开头我就说这只是一个toy implement，400张图片根本不适合用DL来做。

当然我写这篇文章，只是为了总结一下这个实现流程，这一点希望对读者也有参考意义。

最后，我的代码都放在这里：[github地址，可以下载](https://github.com/wepe/MachineLearning-Demo/tree/master/DeepLearning%20Tutorials/FaceRecognition_CNN(olivettifaces))

[DeepLearning tutorial（6）易用的深度学习框架Keras简介](http://blog.csdn.net/u012162613/article/details/45397033)

之前我一直在使用Theano，前面五篇Deeplearning相关的文章也是学习Theano的一些笔记，当时已经觉得Theano用起来略显麻烦，有时想实现一个新的结构，就要花很多时间去编程，所以想过将代码模块化，方便重复使用，但因为实在太忙没有时间去做。最近发现了一个叫做Keras的框架，跟我的想法不谋而合，用起来特别简单，适合快速开发。

1. Keras简介

Keras是基于Theano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。使用文档在这：<http://keras.io/>，这个框架貌似是刚刚火起来的，使用上的问题可以到github提issue:<https://github.com/fchollet/keras>

下面简单介绍一下怎么使用Keras，以Mnist数据库为例,编写一个CNN网络结构，你将会发现特别简单。

2. Keras里的模块介绍

* **Optimizers**

顾名思义，Optimizers包含了一些优化的方法，比如最基本的随机梯度下降SGD,另外还有Adagrad、Adadelta、RMSprop、Adam，一些新的方法以后也会被不断添加进来。

keras.optimizers.SGD(lr=0.01, momentum=0.9, decay=0.9, nesterov=False)

* + 1

上面的代码是SGD的使用方法，lr表示学习速率,momentum表示动量项，decay是学习速率的衰减系数(每个epoch衰减一次),Nesterov的值是False或者True，表示使不使用Nesterov momentum。其他的请参考文档。

* **Objectives**

这是目标函数模块，keras提供了mean\_squared\_error，mean\_absolute\_error   
，squared\_hinge，hinge，binary\_crossentropy，categorical\_crossentropy这几种目标函数。

这里binary\_crossentropy 和 categorical\_crossentropy也就是常说的logloss.

* **Activations**

这是激活函数模块，keras提供了linear、sigmoid、hard\_sigmoid、tanh、softplus、relu、softplus，另外softmax也放在Activations模块里(我觉得放在layers模块里更合理些）。此外，像LeakyReLU和PReLU这种比较新的激活函数，keras在keras.layers.advanced\_activations模块里提供。

* **Initializations**

这是参数初始化模块，在添加layer的时候调用init进行初始化。keras提供了uniform、lecun\_uniform、normal、orthogonal、zero、glorot\_normal、he\_normal这几种。

* **layers**

layers模块包含了core、convolutional、recurrent、advanced\_activations、normalization、embeddings这几种layer。

其中core里面包含了flatten(CNN的全连接层之前需要把二维特征图flatten成为一维的)、reshape（CNN输入时将一维的向量弄成二维的）、dense(就是隐藏层，dense是稠密的意思),还有其他的就不介绍了。convolutional层基本就是Theano的Convolution2D的封装。

* **Preprocessing**

这是预处理模块，包括序列数据的处理，文本数据的处理，图像数据的处理。重点看一下图像数据的处理，keras提供了ImageDataGenerator函数,实现data augmentation，数据集扩增，对图像做一些弹性变换，比如水平翻转，垂直翻转，旋转等。

* **Models**

这是最主要的模块，模型。上面定义了各种基本组件，model是将它们组合起来，下面通过一个实例来说明。

3.一个实例：用CNN分类Mnist

* **数据下载**

Mnist数据在其官网上有提供，但是不是图像格式的，因为我们通常都是直接处理图像，为了以后程序能复用，我把它弄成图像格式的，这里可以下载：<http://pan.baidu.com/s/1qCdS6>，共有42000张图片。

* **读取图片数据**

keras要求输入的数据格式是numpy.array类型（numpy是一个python的数值计算的库），所以需要写一个脚本来读入mnist图像，保存为一个四维的data，还有一个一维的label，代码：

#coding:utf-8

"""

Author:wepon

Source:https://github.com/wepe

file:data.py

"""

import os

from PIL import Image

import numpy as np

#读取文件夹mnist下的42000张图片，图片为灰度图，所以为1通道，

#如果是将彩色图作为输入,则将1替换为3，并且data[i,:,:,:] = arr改为data[i,:,:,:] = [arr[:,:,0],arr[:,:,1],arr[:,:,2]]

def load\_data():

data = np.empty((42000,1,28,28),dtype="float32")

label = np.empty((42000,),dtype="uint8")

imgs = os.listdir("./mnist")

num = len(imgs)

for i in range(num):

img = Image.open("./mnist/"+imgs[i])

arr = np.asarray(img,dtype="float32")

data[i,:,:,:] = arr

label[i] = int(imgs[i].split('.')[0])

return data,label

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 构建CNN，训练

短短二十多行代码，构建一个三个卷积层的CNN，直接读下面的代码吧，有注释，很容易读懂：

#导入各种用到的模块组件

from \_\_future\_\_ import absolute\_import

from \_\_future\_\_ import print\_function

from keras.preprocessing.image import ImageDataGenerator

from keras.models import Sequential

from keras.layers.core import Dense, Dropout, Activation, Flatten

from keras.layers.advanced\_activations import PReLU

from keras.layers.convolutional import Convolution2D, MaxPooling2D

from keras.optimizers import SGD, Adadelta, Adagrad

from keras.utils import np\_utils, generic\_utils

from six.moves import range

from data import load\_data

#加载数据

data, label = load\_data()

print(data.shape[0], ' samples')

#label为0~9共10个类别，keras要求格式为binary class matrices,转化一下，直接调用keras提供的这个函数

label = np\_utils.to\_categorical(label, 10)

###############

#开始建立CNN模型

###############

#生成一个model

model = Sequential()

#第一个卷积层，4个卷积核，每个卷积核大小5\*5。1表示输入的图片的通道,灰度图为1通道。

#border\_mode可以是valid或者full，具体看这里说明：http://deeplearning.net/software/theano/library/tensor/nnet/conv.html#theano.tensor.nnet.conv.conv2d

#激活函数用tanh

#你还可以在model.add(Activation('tanh'))后加上dropout的技巧: model.add(Dropout(0.5))

model.add(Convolution2D(4, 1, 5, 5, border\_mode='valid'))

model.add(Activation('tanh'))

#第二个卷积层，8个卷积核，每个卷积核大小3\*3。4表示输入的特征图个数，等于上一层的卷积核个数

#激活函数用tanh

#采用maxpooling，poolsize为(2,2)

model.add(Convolution2D(8,4, 3, 3, border\_mode='valid'))

model.add(Activation('tanh'))

model.add(MaxPooling2D(poolsize=(2, 2)))

#第三个卷积层，16个卷积核，每个卷积核大小3\*3

#激活函数用tanh

#采用maxpooling，poolsize为(2,2)

model.add(Convolution2D(16, 8, 3, 3, border\_mode='valid'))

model.add(Activation('tanh'))

model.add(MaxPooling2D(poolsize=(2, 2)))

#全连接层，先将前一层输出的二维特征图flatten为一维的。

#Dense就是隐藏层。16就是上一层输出的特征图个数。4是根据每个卷积层计算出来的：(28-5+1)得到24,(24-3+1)/2得到11，(11-3+1)/2得到4

#全连接有128个神经元节点,初始化方式为normal

model.add(Flatten())

model.add(Dense(16\*4\*4, 128, init='normal'))

model.add(Activation('tanh'))

#Softmax分类，输出是10类别

model.add(Dense(128, 10, init='normal'))

model.add(Activation('softmax'))

#############

#开始训练模型

##############

#使用SGD + momentum

#model.compile里的参数loss就是损失函数(目标函数)

sgd = SGD(l2=0.0,lr=0.05, decay=1e-6, momentum=0.9, nesterov=True)

model.compile(loss='categorical\_crossentropy', optimizer=sgd,class\_mode="categorical")

#调用fit方法，就是一个训练过程. 训练的epoch数设为10，batch\_size为100．

#数据经过随机打乱shuffle=True。verbose=1，训练过程中输出的信息，0、1、2三种方式都可以，无关紧要。show\_accuracy=True，训练时每一个epoch都输出accuracy。

#validation\_split=0.2，将20%的数据作为验证集。

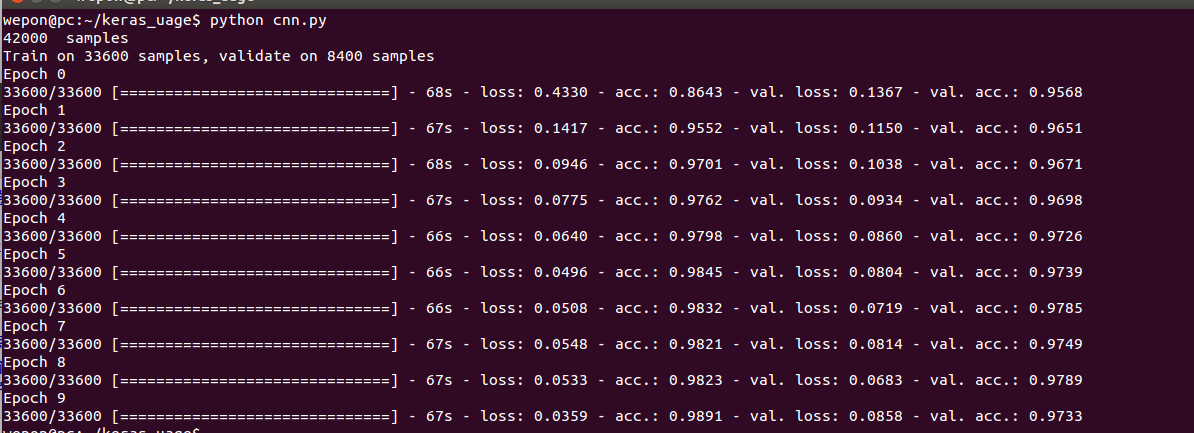
model.fit(data, label, batch\_size=100,nb\_epoch=10,shuffle=True,verbose=1,show\_accuracy=True,validation\_split=0.2)

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 57
* 58
* 59
* 60
* 61
* 62
* 63
* 64
* 65
* 66
* 67
* 68
* 69
* 70
* 71
* **代码使用与结果**

代码放在我github的机器学习仓库里：<https://github.com/wepe/MachineLearning>,非github用户直接点右下的DownloadZip。

在/DeepLearning Tutorials/keras\_usage目录下包括data.py,cnn.py两份代码，下载Mnist数据后解压到该目录下，运行cnn.py这份文件即可。

结果如下所示，在Epoch 9达到了0.98的训练集识别率和0.97的验证集识别率：



[DeepLearning tutorial（7）深度学习框架Keras的使用-进阶](http://blog.csdn.net/u012162613/article/details/45581421)

[上一篇文章](http://blog.csdn.net/u012162613/article/details/45397033)总结了Keras的基本使用方法，相信用过的同学都会觉得不可思议，太简洁了。十多天前，我在github上发现这个框架的时候，关注Keras的人还比较少，这两天无论是github还是微薄，都看到越来越多的人关注和使用Keras。所以这篇文章就简单地再介绍一下Keras的使用,方便各位入门。

主要包括以下三个内容：

* 训练CNN并保存训练好的模型。
* 将CNN用于特征提取，用提取出来的特征训练SVM。
* 可视化CNN卷积层后的特征图。

仍然以Mnist为例，代码中用的Mnist数据到这里下载   
<http://pan.baidu.com/s/1qCdS6>,本文的代码在我的github上:[dive\_into \_keras](https://github.com/wepe/MachineLearning/tree/master/DeepLearning%20Tutorials)

1. 加载数据

数据是图片格式，利用pyhton的PIL模块读取，并转为numpy.array类型。这部分的代码在data.py里：

2. 训练CNN并保存训练好的CNN模型

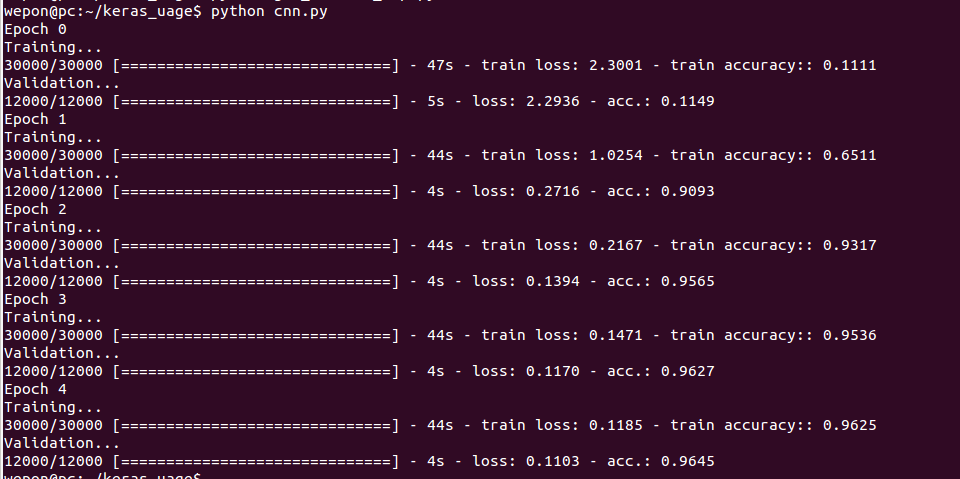
将上一步加载进来的数据分为训练数据（X\_train，30000个样本）和验证数据（X\_val，12000个样本），构建CNN模型并训练。训练过程中，每一个epoch得到的val-accuracy都不一样，我们保存达到最好的val-accuracy时的模型，利用Python的cPickle模块保持。（Keras的开发者最近在添加用hdf5保持模型的功能，我试了一下，没用成功，去github发了issue也没人回，估计还没完善，hdf5压缩率会更高，保存下来的文件会更小。）

这部分的代码在cnn.py里，运行:

python cnn.py

* 1

在第Epoch 4得到96.45%的validation accuracy,运行完后会得到model.pkl这份文件，保存的就是96.45%对应的模型：



3.将CNN用于特征提取，用提取出来的特征训练SVM

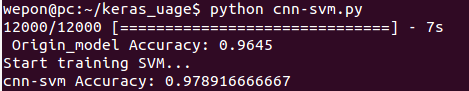
上一步得到了一个val-accuracy为96.45%的CNN模型，在一些论文中经常会看到用CNN的全连接层的输出作为特征，然后去训练其他分类器。这里我也试了一下，用全连接层的输出作为样本的特征向量，训练SVM。SVM用的是scikit learn里的算法。

这部分代码在cnn-svm.py,运行：

python cnn-svm.py

* 1

得到下图的输出，可以看到，cnn-svm的准确率提高到97.89%：



4.可视化CNN卷积层后的特征图

将卷积层和全连接层后的特征图、特征向量以图片形式展示出来，用到matplotlib这个库。这部分代码在get\_feature\_map.py里。运行：

python get\_feature\_map.py

* 1

得到全连接层的输出，以及第一个卷积层输出的4个特征图：



