[**theano学习指南1（翻译）**](http://www.cnblogs.com/xueliangliu/archive/2013/04/03/2997437.html)

theano学习指南，主要翻译官方文档

**基础知识**

本学习指南不是一份机器学习的教程，但是首先我们会对其中的概念做一个简单的回顾，以确保我们在相同的起跑线上。大家还需要下载几个数据库，以便于跑这个指南里面的程序。

**theano下载安装**

在学习每一个算法的时候，大家都需要下载安装相应的文件，如果你想要一次下载所有的文件，可以通过下面这种方式

|  |
| --- |
| git clone git://github.com/lisa-lab/DeepLearningTutorials.git |

**数据库**

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

MNIST数据集由手写的数字的图像组成，它分为了60,000训练数据和10,000个测试数据。在很多文献以及这个指南里面，官方的训练数据又进一步的分成50,000的训练数据和10,000的验证数据，以便于模型参数的选择。所有的图像都做了规范化的处理，每个图像的大小都是28\*28.在原始数据中，图像的像素存成常用的灰度图（灰度区间0~255）。

为了方便在python中调用改数据集，我们对其进行了序列化。序列化后的文件包括三个list，训练数据，验证数据和测试数据。list中的每一个元素都是由图像和相应的标注组成的。其中图像是一个784维（28\*28）的numpy数组，标注则是一个0-9之间的数字。下面的代码演示了如何使用这个数据集。

import cPickle, gzip, numpy

# Load the dataset

f = gzip.open('mnist.pkl.gz', 'rb')

train\_set, valid\_set, test\_set = cPickle.load(f)

f.close()

在使用这个数据集的时候，我们一般把它分成若干minibatch。我们也鼓励你吧数据集存成共享变量，并根据minibatch的索引来访问它。这样做是为了在GPU上运行代码的方便。当复制代码到GPU上时，数据会有很大的重叠。如果你按照程序请求来复制数据，而不是通过共享变量的方式，GPU上面的程序就不会比运行在CPU上面的快。如果你运用theano的共享数据，就使得theano可以通过一个调用复制所有数据到GPU上。（有些说明没翻译，对GPU的原理不是很理解-译者）

到目前为止，数据保存到了一个变量中，minibatch则是这个变量的一系列的切片，它最自然的定义方法是这个切片的位置和大小。在我们的设置汇总，每个块的大小都是固定的，所以函数只要通过切片的位置就可以访问每个minibatch。下面的代码演示了如果存储数据及minibatch。

[复制代码](javascript:void(0);)

def shared\_dataset(data\_xy):

""" Function that loads the dataset into shared variables

The reason we store our dataset in shared variables is to allow

Theano to copy it into the GPU memory (when code is run on GPU).

Since copying data into the GPU is slow, copying a minibatch everytime

is needed (the default behaviour if the data is not in a shared

variable) would lead to a large decrease in performance.

"""

data\_x, data\_y = data\_xy

shared\_x = theano.shared(numpy.asarray(data\_x, dtype=theano.config.floatX))

shared\_y = theano.shared(numpy.asarray(data\_y, dtype=theano.config.floatX))

# When storing data on the GPU it has to be stored as floats

# therefore we will store the labels as ``floatX`` as well

# (``shared\_y`` does exactly that). But during our computations

# we need them as ints (we use labels as index, and if they are

# floats it doesn't make sense) therefore instead of returning

# ``shared\_y`` we will have to cast it to int. This little hack

# lets us get around this issue

return shared\_x, T.cast(shared\_y, 'int32')

test\_set\_x, test\_set\_y = shared\_dataset(test\_set)

valid\_set\_x, valid\_set\_y = shared\_dataset(valid\_set)

train\_set\_x, train\_set\_y = shared\_dataset(train\_set)

batch\_size = 500 # size of the minibatch

# accessing the third minibatch of the training set

data = train\_set\_x[2 \* 500: 3 \* 500]

label = train\_set\_y[2 \* 500: 3 \* 500]

[复制代码](javascript:void(0);)

**符号**

**数据集符号**

首先，我们用 $\mathbf{D}$来表示数据集，为了区分的方便，训练，验证和测试数据可以分别用$\mathbf{D\_{train}}$，$\mathbf{D\_{valid}}$， $\mathbf{D\_{test}}$来表示。

本指南着眼于分类问题，对于每一个数据集，都有一些数据对（$x^{(i)},y^{(i)}$）组成。其中$x^{(i)} \in R^D$为特征向量，$y^{(i)} \in (0~L)$  表示了数据$x^{(i)}$的类别。

对于其他符号，如无特殊说明，做如下约定，

* $\mathbf{W}$ 大写符号表示矩阵
* $W\_{ij}$ 矩阵第i行，第j列的元素
* $W\_{i.}$ 行向量
* $W\_{.j}$ 列向量
* $b$  向量
* $b\_i$  向量的元素

符号和函数的定义列表如下

* $D$  输入向量的维度
* $D\_h^{i}$  第i层隐变量的个数
* $f\_\theta(x), f(x)$ 分类函数
* L  标注的个数
* $L(\theta,D)$ 模型似然函数的对数形式
* $l(\theta,D)$  预测函数的经验损失
* NLL    负的以对数表示的似然函数
* $\theta$  模型的参数集合

 Python名字空间

本指南的程序一般引用如下名字空间

import theano

import theano.tensor as T

import numpy

**监督优化问题入门**

在深度学习中，深度网络的无监督学习得到了广泛的应用。但是监督学习仍然扮演着重要角色。本章节简单的回顾一下分类问题的监督学习模型，并且介绍在theano下面随机梯度下降算法的实现。

**分类器的学习**

**0-1损失**

在本指南中介绍的方法也常常用于一般的分类问题中。训练一个分类器的目的是最小化预测函数在测试实例上面的错误。这种错误最简单的表示方法是0-1损失。如果预测函数定义为$f: \mathbf{R^D} -> {0,...,\mathbf{L}}$，那么损失函数可以表示为：

l0,1=∑i=0|D|If(xi≠yi)

这里，$D$ 可以是训练过程中的训练数据，或者和训练数据没有任何交集，以避免验证或测试过程中的偏差。 指标函数$I$定义为：

Ix={1ifxisTrue0else

在本指南中，预测函数定于为：

f(x)=argmaxkP(Y=k|x,θ)

在python中，结合Theano，该函数的实现如下：

# zero\_one\_loss is a Theano variable representing a symbolic

# expression of the zero one loss ; to get the actual value this

# symbolic expression has to be compiled into a Theano function (see

# the Theano tutorial for more details)

zero\_one\_loss = T.sum(T.neq(T.argmax(p\_y\_given\_x), y))

**负对数似然损失**

因为0-1损失函数是不可微的，在一个含有几千甚至几万个参数的复杂问题中，模型的求解变得非常困难。因此我们最大化分类器的对数似然函数：

L(θ,D)=∑i=0|D|logP(Y=yi|xi,θ)

 正确类别的似然，并不和正确预测的数目完全一致，但是，从随机初始化的分类器的角度看，他们是非常类似的。但是请记住，似然函数和0-1损失函数是不同的，你应该看到他们的在验证数据上面的正相关性，但是有时候又是负相关。（这段是不是很明白）

既然我们可以最小化损失函数，那么学习的过程，也就是最小化负的对数似然函数的过程：

NLL(θ,D)=∑i=0|D|logP(Y=yi|xi,θ)

NLL函数其实是0-1损失函数的一种可以微分的替代，这样我们就可以用它在训练集合的梯度来训练分类器。相应的代码如下：

[复制代码](javascript:void(0);)

# NLL is a symbolic variable ; to get the actual value of NLL, this symbolic

# expression has to be compiled into a Theano function (see the Theano

# tutorial for more details)

NLL = -T.sum(T.log(p\_y\_given\_x)[T.arange(y.shape[0]), y])

# note on syntax: T.arange(y.shape[0]) is a vector of integers [0,1,2,...,len(y)].

# Indexing a matrix M by the two vectors [0,1,...,K], [a,b,...,k] returns the

# elements M[0,a], M[1,b], ..., M[K,k] as a vector. Here, we use this

# syntax to retrieve the log-probability of the correct labels, y.

[复制代码](javascript:void(0);)

**随机梯度下降算法**

什么是一般的梯度下降呢？如果我们定义了损失函数，这种方法在错误平面上面，重复地小幅的向下移动参数，以达到最优化的目的。通过梯度下降，训练数据在损失函数上面达到极值，相应的伪代码如下：

[复制代码](javascript:void(0);)

# GRADIENT DESCENT

while True:

loss = f(params)

d\_loss\_wrt\_params = ... # compute gradient

params -= learning\_rate \* d\_loss\_wrt\_params

if <stopping condition is met>:

return params

[复制代码](javascript:void(0);)

随机梯度下降（SGD）也遵从类似的原理，但是它每次估计梯度的时候，只采用一小部分训练数据，因而处理速度更快，相应的伪代码如下：

[复制代码](javascript:void(0);)

# STOCHASTIC GRADIENT DESCENT

for (x\_i,y\_i) in training\_set:

# imagine an infinite generator

# that may repeat examples (if there is only a finite training set)

loss = f(params, x\_i, y\_i)

d\_loss\_wrt\_params = ... # compute gradient

params -= learning\_rate \* d\_loss\_wrt\_params

if <stopping condition is met>:

return params

[复制代码](javascript:void(0);)

当在深度学习中采用minibatch的时候，SGD稍微有一点变化。在minibatch SGD中，我们每次用多个训练数据来估计梯度。这种技术减少了估计的梯度方差，也充分的利用了现在计算机体系结构中的内存的层次化组织技术。

[复制代码](javascript:void(0);)

for (x\_batch,y\_batch) in train\_batches:

# imagine an infinite generator

# that may repeat examples

loss = f(params, x\_batch, y\_batch)

d\_loss\_wrt\_params = ... # compute gradient using theano

params -= learning\_rate \* d\_loss\_wrt\_params

if <stopping condition is met>:

return params

[复制代码](javascript:void(0);)

以上的伪代码描述了算法是如何工作的，在Theano平台下的具体实现为：

[复制代码](javascript:void(0);)

# Minibatch Stochastic Gradient Descent

# assume loss is a symbolic description of the loss function given

# the symbolic variables params (shared variable), x\_batch, y\_batch;

# compute gradient of loss with respect to params

d\_loss\_wrt\_params = T.grad(loss, params)

# compile the MSGD step into a theano function

updates = [(params, params - learning\_rate \* d\_loss\_wrt\_params)]

MSGD = theano.function([x\_batch,y\_batch], loss, updates=updates)

for (x\_batch, y\_batch) in train\_batches:

# here x\_batch and y\_batch are elements of train\_batches and

# therefore numpy arrays; function MSGD also updates the params

print('Current loss is ', MSGD(x\_batch, y\_batch))

if stopping\_condition\_is\_met:

return params

[复制代码](javascript:void(0);)

**规则化**

机器学习要优化复杂一些。我们从一些数据上面训练模型的目的，是要把它应用到新的数据上面。但是前面的训练算法并没有考虑这一点，这有可能引起训练过度的问题。一种解决训练过度的办法是规则化，有几种技术可以实现，这里我们主要介绍L1/L2规则化，以及提前结束训练的技术。

**L1/L2规则化**

这种技术主要是在损失函数上面添加一项，从而达到对相关的参数的惩罚的目的。假设我们的损失函数为：

NLL(θ,D)=∑i=0|D|logP(Y=yi|xi,θ)

那么规则化的后的损失函数可以定义为：

E(θ,D)=NLL(θ,D)+λR(θ)

在我们的问题，函数可以具体定义为：

E(θ,D)=NLL(θ,D)+λ||θpp||

这里，

$$||\theta||\_p = (\sum\_{j=0}^{|\theta|}{|\theta\_j|^p})^{\frac{1}{p}}$$

为参数$\theta$的$L\_p$范数。通常p的取值为1或者2。当p=2的是，规范化又称权衰减。

应该注意的是，这种简单的方法并不一定意味着模型的泛化。在实际应用过程中，人们发现在神经网络中应用这种技术有助于泛化，特别是小数据集上面。下面的代码演示了如何应用这种技术。

[复制代码](javascript:void(0);)

# symbolic Theano variable that represents the L1 regularization term

L1 = T.sum(abs(param))

# symbolic Theano variable that represents the squared L2 term

L2\_sqr = T.sum(param \*\* 2)

# the loss

loss = NLL + lambda\_1 \* L1 + lambda\_2 \* L2

[复制代码](javascript:void(0);)

**提前结束训练**

提前结束训练是另一种处理训练过度的办法，它的解决思路是监测模型在验证数据上的表现。验证数据在训练过程中，可以用来做测试数据。如果模型的性能在验证数据中改进很小，真是变差，那么就应该放弃进一步的优化。

停止优化的判别有很多方法，在这个指南中，我们用一种基于patience(???)几何增长的策略。

[复制代码](javascript:void(0);)

# early-stopping parameters

patience = 5000 # look as this many examples regardless

patience\_increase = 2 # wait this much longer when a new best is

# found

improvement\_threshold = 0.995 # a relative improvement of this much is

# considered significant

validation\_frequency = min(n\_train\_batches, patience/2)

# go through this many

# minibatches before checking the network

# on the validation set; in this case we

# check every epoch

best\_params = None

best\_validation\_loss = numpy.inf

test\_score = 0.

start\_time = time.clock()

done\_looping = False

epoch = 0

while (epoch < n\_epochs) and (not done\_looping):

# Report "1" for first epoch, "n\_epochs" for last epoch

epoch = epoch + 1

for minibatch\_index in xrange(n\_train\_batches):

d\_loss\_wrt\_params = ... # compute gradient

params -= learning\_rate \* d\_loss\_wrt\_params # gradient descent

# iteration number. We want it to start at 0.

iter = (epoch - 1) \* n\_train\_batches + minibatch\_index

# note that if we do `iter % validation\_frequency` it will be

# true for iter = 0 which we do not want. We want it true for

# iter = validation\_frequency - 1.

if (iter + 1) % validation\_frequency == 0:

this\_validation\_loss = ... # compute zero-one loss on validation set

if this\_validation\_loss < best\_validation\_loss:

# improve patience if loss improvement is good enough

if this\_validation\_loss < best\_validation\_loss \* improvement\_threshold:

patience = max(patience, iter \* patience\_increase)

best\_params = copy.deepcopy(params)

best\_validation\_loss = this\_validation\_loss

if patience <= iter:

done\_looping = True

break

# POSTCONDITION:

# best\_params refers to the best out-of-sample parameters observed during the optimization

[**theano学习指南2（翻译）-对数回归分类器**](http://www.cnblogs.com/xueliangliu/archive/2013/04/07/3006014.html)

在本章节中，我们会学习如何用Theano实现最基本的对数回归分类器。首先，我们会简单的复习一个这个模型，在这个过程中，大家可以进一步的了解如何把数学表达式和Theano的图模型结合起来。

**数学模型**

对数回归模型是试过线性概率分类器，它有两个参数，权重矩阵$W$和偏移向量$b$.分类的过程是把数据投影到一组高维超平面上，数据和平面的距离反应了它属于这个类别的概率。这个模型的数学公式可以表示为：

P(Y=i|x,W,b)=softmaxi(Wx+b)=eWix+bi∑jeWjx+bj

模型的输出即为预测的结果，它的值为：

ypred=argmaxiP(Y=i|x,W,b)

在Theano中，通过以下函数实现如下功能

[复制代码](javascript:void(0);)

# generate symbolic variables for input (x and y represent a

# minibatch)

x = T.fmatrix('x')

y = T.lvector('y')

# allocate shared variables model params

b = theano.shared(numpy.zeros((10,)), name='b')

W = theano.shared(numpy.zeros((784, 10)), name='W')

# symbolic expression for computing the vector of

# class-membership probabilities

p\_y\_given\_x = T.nnet.softmax(T.dot(x, W) + b)

# compiled Theano function that returns the vector of class-membership

# probabilities

get\_p\_y\_given\_x = theano.function(inputs=[x], outputs=p\_y\_given\_x)

# print the probability of some example represented by x\_value

# x\_value is not a symbolic variable but a numpy array describing the

# datapoint

print 'Probability that x is of class %i is %f' % (i, get\_p\_y\_given\_x(x\_value)[i])

# symbolic description of how to compute prediction as class whose probability

# is maximal

y\_pred = T.argmax(p\_y\_given\_x, axis=1)

# compiled theano function that returns this value

classify = theano.function(inputs=[x], outputs=y\_pred)

[复制代码](javascript:void(0);)

 在以上代码中，首先定义了输入变量$x$,$y$. 因为模型在训练过程中要保持一个稳定的状态，模型参数$W$,$b$定义成共享变量，这种定义不仅可以声明变量，还会初始化他们的值。接下来，点乘和softmax操作用来计算模型输出$P(Y|x,W,b)$. 结果保存在变量p\_y\_given\_x中。

到目前为止，我们仅仅订了Theano运行的计算图模型。为了得到真实的$P(Y|x,W,b)$值，我们需要创建函数 get\_p\_y\_given\_x, 它以x为参数，输出值为p\_y\_given\_x。我们可以遍历它的值，并得到数据属于每一个类别的概率。

现在，让我们结束Theano图的创建。为了得到模型的预测结果，我们用T.argmax操作符，这个操作返回p\_y\_given\_x中做大值得索引。

类似的，为了得到给定输入的预测结果，我们定义函数classify。该函数以模型输入矩阵$x$为参数，输出为列向量，表示了每个实例的预测类别。

当然，这个模型还没有任何用途，因为模型参数还处于初始状态。下面的章节中，我们将学习如何训练模型 。

**损失函数**

模型的训练过程也就是最小化损失函数的过程。 在多类别的对数回归模型中，通常采用负对数似然函数作为模型的参数。这相当于在以$\theta$为参数的模型中，最大化训练数据的似然。如果我们定义似然和损失函数如下：

L(θ={W,b},D)=∑i=0|D|log(P(Y=y(i)|x(i),W,b))ℓ(θ={W,b},D)=−L(θ={W,b},D)

下面的代码演示了如何计算一个minbatch的损失

loss = -T.mean(T.log(p\_y\_given\_x)[T.arange(y.shape[0]), y])

# note on syntax: T.arange(y.shape[0]) is a vector of integers [0,1,2,...,len(y)].

# Indexing a matrix M by the two vectors [0,1,...,K], [a,b,...,k] returns the

# elements M[0,a], M[1,b], ..., M[K,k] as a vector. Here, we use this

# syntax to retrieve the log-probability of the correct labels, y.

**创建LogisticRegression类**

现在我们已经有了LogisticRegression类的所有功能。该类的代码如下，这些代码涵盖了我们之前学习的所有功能。

[复制代码](javascript:void(0);)

class LogisticRegression(object):

def \_\_init\_\_(self, input, n\_in, n\_out):

""" Initialize the parameters of the logistic regression

:type input: theano.tensor.TensorType

:param input: symbolic variable that describes the input of the

architecture (e.g., one minibatch of input images)

:type n\_in: int

:param n\_in: number of input units, the dimension of the space in

which the datapoint lies

:type n\_out: int

:param n\_out: number of output units, the dimension of the space in

which the target lies

"""

# initialize with 0 the weights W as a matrix of shape (n\_in, n\_out)

self.W = theano.shared(value=numpy.zeros((n\_in, n\_out),

dtype=theano.config.floatX), name='W' )

# initialize the baises b as a vector of n\_out 0s

self.b = theano.shared(value=numpy.zeros((n\_out,),

dtype=theano.config.floatX), name='b' )

# compute vector of class-membership probabilities in symbolic form

self.p\_y\_given\_x = T.nnet.softmax(T.dot(input, self.W) + self.b)

# compute prediction as class whose probability is maximal in

# symbolic form

self.y\_pred=T.argmax(self.p\_y\_given\_x, axis=1)

def negative\_log\_likelihood(self, y):

"""Return the mean of the negative log-likelihood of the prediction

of this model under a given target distribution.

.. math::

\frac{1}{|\mathcal{D}|} \mathcal{L} (\theta=\{W,b\}, \mathcal{D}) =

\frac{1}{|\mathcal{D}|} \sum\_{i=0}^{|\mathcal{D}|} \log(P(Y=y^{(i)}|x^{(i)}, W,b)) \\

\ell (\theta=\{W,b\}, \mathcal{D})

:param y: corresponds to a vector that gives for each example the

correct label;

Note: we use the mean instead of the sum so that

the learning rate is less dependent on the batch size

"""

return -T.mean(T.log(self.p\_y\_given\_x)[T.arange(y.shape[0]), y])

[复制代码](javascript:void(0);)

这个类可以通过以下方式实例化：

[复制代码](javascript:void(0);)

# allocate symbolic variables for the data

x = T.fmatrix() # the data is presented as rasterized images (each being a 1-D row vector in x)

y = T.lvector() # the labels are presented as 1D vector of [long int] labels

# construct the logistic regression class

classifier = LogisticRegression(

input=x.reshape((batch\_size, 28 \* 28)), n\_in=28 \* 28, n\_out=10)

[复制代码](javascript:void(0);)

最后，定义损失函数：

cost = classifier.negative\_log\_likelihood(y)

**模型的训练**

为了在编程语言里面实现MSGD，我们需要手动计算模型的微分。如果模型比较复杂的话，计算过程会变得非常困难。

在Theano中，这个工作可以通过函数自动的完成，实例代码如下：

# compute the gradient of cost with respect to theta = (W,b)

g\_W = T.grad(cost, classifier.W)

g\_b = T.grad(cost, classifier.b)

g\_W和g\_b是符号变量，他们可以用在计算图模型中。下面代码演示了执行一步梯度下降算法的过程：

[复制代码](javascript:void(0);)

# compute the gradient of cost with respect to theta = (W,b)

g\_W = T.grad(cost=cost, wrt=classifier.W)

g\_b = T.grad(cost=cost, wrt=classifier.b)

# specify how to update the parameters of the model as a list of

# (variable, update expression) pairs

updates = [(classifier.W, classifier.W - learning\_rate \* g\_W),

(classifier.b, classifier.b - learning\_rate \* g\_b)]

# compiling a Theano function `train\_model` that returns the cost, but in

# the same time updates the parameter of the model based on the rules

# defined in `updates`

train\_model = theano.function(inputs=[index],

outputs=cost,

updates=updates,

givens={

x: train\_set\_x[index \* batch\_size: (index + 1) \* batch\_size],

y: train\_set\_y[index \* batch\_size: (index + 1) \* batch\_size]})

[复制代码](javascript:void(0);)

update这列表里面包含了对每一个变量的随机梯度算法下面的更新操作。givens 字典里面包含数据和计算图模型中变量的映射关系。整个train\_model定义了：

* 输入：为通过index索引的mini-batch，其数据定义为$x$，相应的label表示为$y$.
* 返回值，为相应的损失
* 每次函数调用的时候，首先通过index检索相应的参数 $x$， $y$， 然后计算在这个minbatch上面的函数损失，并应用定义在updates 列表中的操作更新参数。

函数 train\_model(index) 调用的时候，它会计算并返回近似的损失，并执行一步MSGD操作。整个学习过程包括一系列的在该数据集上的循环，也就是是一个反复的调用这个函数的过程

**模型的测试**

正如第一节介绍的，我们对模型的测试主要是关心它的错误分类的数据的数量，而不仅仅是似然函数。因此类 LogisticRegression 中需要一个成员函数，用于建立返回测试数据上面的误分数据的数目符号图（symbolic graph）。 代码如下：

[复制代码](javascript:void(0);)

class LogisticRegression(object):

def errors(self, y):

"""Return a float representing the number of errors in the minibatch

over the total number of examples of the minibatch ; zero

one loss over the size of the minibatch

"""

return T.mean(T.neq(self.y\_pred, y))

[复制代码](javascript:void(0);)

接下来我们定义函数 test\_model和validte\_model， 以便于得到这个函数的值。 validate\_model是实现前期结束的关键（见前一节）。两个函数的功能都是对已一个给定的batch，计算其误分类的实例的数目。两个函数的区别在于，它们一个运行在测试数据上，一个运行在验证数据上。相应的函数代码如下：

[复制代码](javascript:void(0);)

# compiling a Theano function that computes the mistakes that are made by

# the model on a minibatch

test\_model = theano.function(inputs=[index],

outputs=classifier.errors(y),

givens={

x: test\_set\_x[index \* batch\_size: (index + 1) \* batch\_size],

y: test\_set\_y[index \* batch\_size: (index + 1) \* batch\_size]})

validate\_model = theano.function(inputs=[index],

outputs=classifier.errors(y),

givens={

x: valid\_set\_x[index \* batch\_size: (index + 1) \* batch\_size],

y: valid\_set\_y[index \* batch\_size: (index + 1) \* batch\_size]})

[复制代码](javascript:void(0);)

**综合所有功能**

如果把以上所有的功能整合在一起，就得到如下的代码：

[复制代码](javascript:void(0);)

"""

This tutorial introduces logistic regression using Theano and stochastic

gradient descent.

Logistic regression is a probabilistic, linear classifier. It is parametrized

by a weight matrix :math:`W` and a bias vector :math:`b`. Classification is

done by projecting data points onto a set of hyperplanes, the distance to

which is used to determine a class membership probability.

Mathematically, this can be written as:

.. math::

P(Y=i|x, W,b) &= softmax\_i(W x + b) \\

&= \frac {e^{W\_i x + b\_i}} {\sum\_j e^{W\_j x + b\_j}}

The output of the model or prediction is then done by taking the argmax of

the vector whose i'th element is P(Y=i|x).

.. math::

y\_{pred} = argmax\_i P(Y=i|x,W,b)

This tutorial presents a stochastic gradient descent optimization method

suitable for large datasets, and a conjugate gradient optimization method

that is suitable for smaller datasets.

References:

- textbooks: "Pattern Recognition and Machine Learning" -

Christopher M. Bishop, section 4.3.2

"""

\_\_docformat\_\_ = 'restructedtext en'

import cPickle

import gzip

import os

import sys

import time

import numpy

import theano

import theano.tensor as T

class LogisticRegression(object):

"""Multi-class Logistic Regression Class

The logistic regression is fully described by a weight matrix :math:`W`

and bias vector :math:`b`. Classification is done by projecting data

points onto a set of hyperplanes, the distance to which is used to

determine a class membership probability.

"""

def \_\_init\_\_(self, input, n\_in, n\_out):

""" Initialize the parameters of the logistic regression

:type input: theano.tensor.TensorType

:param input: symbolic variable that describes the input of the

architecture (one minibatch)

:type n\_in: int

:param n\_in: number of input units, the dimension of the space in

which the datapoints lie

:type n\_out: int

:param n\_out: number of output units, the dimension of the space in

which the labels lie

"""

# initialize with 0 the weights W as a matrix of shape (n\_in, n\_out)

self.W = theano.shared(value=numpy.zeros((n\_in, n\_out),

dtype=theano.config.floatX),

name='W', borrow=True)

# initialize the baises b as a vector of n\_out 0s

self.b = theano.shared(value=numpy.zeros((n\_out,),

dtype=theano.config.floatX),

name='b', borrow=True)

# compute vector of class-membership probabilities in symbolic form

self.p\_y\_given\_x = T.nnet.softmax(T.dot(input, self.W) + self.b)

# compute prediction as class whose probability is maximal in

# symbolic form

self.y\_pred = T.argmax(self.p\_y\_given\_x, axis=1)

# parameters of the model

self.params = [self.W, self.b]

def negative\_log\_likelihood(self, y):

"""Return the mean of the negative log-likelihood of the prediction

of this model under a given target distribution.

.. math::

\frac{1}{|\mathcal{D}|} \mathcal{L} (\theta=\{W,b\}, \mathcal{D}) =

\frac{1}{|\mathcal{D}|} \sum\_{i=0}^{|\mathcal{D}|} \log(P(Y=y^{(i)}|x^{(i)}, W,b)) \\

\ell (\theta=\{W,b\}, \mathcal{D})

:type y: theano.tensor.TensorType

:param y: corresponds to a vector that gives for each example the

correct label

Note: we use the mean instead of the sum so that

the learning rate is less dependent on the batch size

"""

# y.shape[0] is (symbolically) the number of rows in y, i.e.,

# number of examples (call it n) in the minibatch

# T.arange(y.shape[0]) is a symbolic vector which will contain

# [0,1,2,... n-1] T.log(self.p\_y\_given\_x) is a matrix of

# Log-Probabilities (call it LP) with one row per example and

# one column per class LP[T.arange(y.shape[0]),y] is a vector

# v containing [LP[0,y[0]], LP[1,y[1]], LP[2,y[2]], ...,

# LP[n-1,y[n-1]]] and T.mean(LP[T.arange(y.shape[0]),y]) is

# the mean (across minibatch examples) of the elements in v,

# i.e., the mean log-likelihood across the minibatch.

return -T.mean(T.log(self.p\_y\_given\_x)[T.arange(y.shape[0]), y])

def errors(self, y):

"""Return a float representing the number of errors in the minibatch

over the total number of examples of the minibatch ; zero one

loss over the size of the minibatch

:type y: theano.tensor.TensorType

:param y: corresponds to a vector that gives for each example the

correct label

"""

# check if y has same dimension of y\_pred

if y.ndim != self.y\_pred.ndim:

raise TypeError('y should have the same shape as self.y\_pred',

('y', target.type, 'y\_pred', self.y\_pred.type))

# check if y is of the correct datatype

if y.dtype.startswith('int'):

# the T.neq operator returns a vector of 0s and 1s, where 1

# represents a mistake in prediction

return T.mean(T.neq(self.y\_pred, y))

else:

raise NotImplementedError()

def load\_data(dataset):

''' Loads the dataset

:type dataset: string

:param dataset: the path to the dataset (here MNIST)

'''

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

# LOAD DATA #

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

# Download the MNIST dataset if it is not present

data\_dir, data\_file = os.path.split(dataset)

if (not os.path.isfile(dataset)) and data\_file == 'mnist.pkl.gz':

import urllib

origin = 'http://www.iro.umontreal.ca/~lisa/deep/data/mnist/mnist.pkl.gz'

print 'Downloading data from %s' % origin

urllib.urlretrieve(origin, dataset)

print '... loading data'

# Load the dataset

f = gzip.open(dataset, 'rb')

train\_set, valid\_set, test\_set = cPickle.load(f)

f.close()

#train\_set, valid\_set, test\_set format: tuple(input, target)

#input is an numpy.ndarray of 2 dimensions (a matrix)

#witch row's correspond to an example. target is a

#numpy.ndarray of 1 dimensions (vector)) that have the same length as

#the number of rows in the input. It should give the target

#target to the example with the same index in the input.

def shared\_dataset(data\_xy, borrow=True):

""" Function that loads the dataset into shared variables

The reason we store our dataset in shared variables is to allow

Theano to copy it into the GPU memory (when code is run on GPU).

Since copying data into the GPU is slow, copying a minibatch everytime

is needed (the default behaviour if the data is not in a shared

variable) would lead to a large decrease in performance.

"""

data\_x, data\_y = data\_xy

shared\_x = theano.shared(numpy.asarray(data\_x,

dtype=theano.config.floatX),

borrow=borrow)

shared\_y = theano.shared(numpy.asarray(data\_y,

dtype=theano.config.floatX),

borrow=borrow)

# When storing data on the GPU it has to be stored as floats

# therefore we will store the labels as ``floatX`` as well

# (``shared\_y`` does exactly that). But during our computations

# we need them as ints (we use labels as index, and if they are

# floats it doesn't make sense) therefore instead of returning

# ``shared\_y`` we will have to cast it to int. This little hack

# lets ous get around this issue

return shared\_x, T.cast(shared\_y, 'int32')

test\_set\_x, test\_set\_y = shared\_dataset(test\_set)

valid\_set\_x, valid\_set\_y = shared\_dataset(valid\_set)

train\_set\_x, train\_set\_y = shared\_dataset(train\_set)

rval = [(train\_set\_x, train\_set\_y), (valid\_set\_x, valid\_set\_y),

(test\_set\_x, test\_set\_y)]

return rval

def sgd\_optimization\_mnist(learning\_rate=0.13, n\_epochs=1000,

dataset='../data/mnist.pkl.gz',

batch\_size=600):

"""

Demonstrate stochastic gradient descent optimization of a log-linear

model

This is demonstrated on MNIST.

:type learning\_rate: float

:param learning\_rate: learning rate used (factor for the stochastic

gradient)

:type n\_epochs: int

:param n\_epochs: maximal number of epochs to run the optimizer

:type dataset: string

:param dataset: the path of the MNIST dataset file from

http://www.iro.umontreal.ca/~lisa/deep/data/mnist/mnist.pkl.gz

"""

datasets = load\_data(dataset)

train\_set\_x, train\_set\_y = datasets[0]

valid\_set\_x, valid\_set\_y = datasets[1]

test\_set\_x, test\_set\_y = datasets[2]

# compute number of minibatches for training, validation and testing

n\_train\_batches = train\_set\_x.get\_value(borrow=True).shape[0] / batch\_size

n\_valid\_batches = valid\_set\_x.get\_value(borrow=True).shape[0] / batch\_size

n\_test\_batches = test\_set\_x.get\_value(borrow=True).shape[0] / batch\_size

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

# BUILD ACTUAL MODEL #

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

print '... building the model'

# allocate symbolic variables for the data

index = T.lscalar() # index to a [mini]batch

x = T.matrix('x') # the data is presented as rasterized images

y = T.ivector('y') # the labels are presented as 1D vector of

# [int] labels

# construct the logistic regression class

# Each MNIST image has size 28\*28

classifier = LogisticRegression(input=x, n\_in=28 \* 28, n\_out=10)

# the cost we minimize during training is the negative log likelihood of

# the model in symbolic format

cost = classifier.negative\_log\_likelihood(y)

# compiling a Theano function that computes the mistakes that are made by

# the model on a minibatch

test\_model = theano.function(inputs=[index],

outputs=classifier.errors(y),

givens={

x: test\_set\_x[index \* batch\_size: (index + 1) \* batch\_size],

y: test\_set\_y[index \* batch\_size: (index + 1) \* batch\_size]})

validate\_model = theano.function(inputs=[index],

outputs=classifier.errors(y),

givens={

x: valid\_set\_x[index \* batch\_size:(index + 1) \* batch\_size],

y: valid\_set\_y[index \* batch\_size:(index + 1) \* batch\_size]})

# compute the gradient of cost with respect to theta = (W,b)

g\_W = T.grad(cost=cost, wrt=classifier.W)

g\_b = T.grad(cost=cost, wrt=classifier.b)

# specify how to update the parameters of the model as a list of

# (variable, update expression) pairs.

updates = [(classifier.W, classifier.W - learning\_rate \* g\_W),

(classifier.b, classifier.b - learning\_rate \* g\_b)]

# compiling a Theano function `train\_model` that returns the cost, but in

# the same time updates the parameter of the model based on the rules

# defined in `updates`

train\_model = theano.function(inputs=[index],

outputs=cost,

updates=updates,

givens={

x: train\_set\_x[index \* batch\_size:(index + 1) \* batch\_size],

y: train\_set\_y[index \* batch\_size:(index + 1) \* batch\_size]})

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

# TRAIN MODEL #

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

print '... training the model'

# early-stopping parameters

patience = 5000 # look as this many examples regardless

patience\_increase = 2 # wait this much longer when a new best is

# found

improvement\_threshold = 0.995 # a relative improvement of this much is

# considered significant

validation\_frequency = min(n\_train\_batches, patience / 2)

# go through this many

# minibatche before checking the network

# on the validation set; in this case we

# check every epoch

best\_params = None

best\_validation\_loss = numpy.inf

test\_score = 0.

start\_time = time.clock()

done\_looping = False

epoch = 0

while (epoch < n\_epochs) and (not done\_looping):

epoch = epoch + 1

for minibatch\_index in xrange(n\_train\_batches):

minibatch\_avg\_cost = train\_model(minibatch\_index)

# iteration number

iter = (epoch - 1) \* n\_train\_batches + minibatch\_index

if (iter + 1) % validation\_frequency == 0:

# compute zero-one loss on validation set

validation\_losses = [validate\_model(i)

for i in xrange(n\_valid\_batches)]

this\_validation\_loss = numpy.mean(validation\_losses)

print('epoch %i, minibatch %i/%i, validation error %f %%' % \

(epoch, minibatch\_index + 1, n\_train\_batches,

this\_validation\_loss \* 100.))

# if we got the best validation score until now

if this\_validation\_loss < best\_validation\_loss:

#improve patience if loss improvement is good enough

if this\_validation\_loss < best\_validation\_loss \* \

improvement\_threshold:

patience = max(patience, iter \* patience\_increase)

best\_validation\_loss = this\_validation\_loss

# test it on the test set

test\_losses = [test\_model(i)

for i in xrange(n\_test\_batches)]

test\_score = numpy.mean(test\_losses)

print((' epoch %i, minibatch %i/%i, test error of best'

' model %f %%') %

(epoch, minibatch\_index + 1, n\_train\_batches,

test\_score \* 100.))

if patience <= iter:

done\_looping = True

break

end\_time = time.clock()

print(('Optimization complete with best validation score of %f %%,'

'with test performance %f %%') %

(best\_validation\_loss \* 100., test\_score \* 100.))

print 'The code run for %d epochs, with %f epochs/sec' % (

epoch, 1. \* epoch / (end\_time - start\_time))

print >> sys.stderr, ('The code for file ' +

os.path.split(\_\_file\_\_)[1] +

' ran for %.1fs' % ((end\_time - start\_time)))

if \_\_name\_\_ == '\_\_main\_\_':

sgd\_optimization\_mnist()

[复制代码](javascript:void(0);)

这段程序采用SGD逻辑回归算法学习分类器，在DeepLearningTutorials文件夹中，可以通过以下命令调用：

python code/logistic\_sgd.py

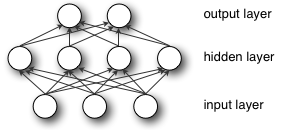
## [theano学习指南3（翻译）-多层感知器模型](http://www.cnblogs.com/xueliangliu/archive/2013/04/26/3044636.html)

本节要用Theano实现的结构是一个隐层的多层感知器模型（MLP）。MLP可以看成一种对数回归器，其中输入通过非线性转移矩阵$\Phi$做一个变换处理，以便于把输入数据投影到一个线性可分的空间上。MLP的中间层一般称为隐层。单一的隐层便可以确保MLP全局近似。然而，我们稍后还会看到多隐层的好处，比如在深度学习中的应用。

（本节只要介绍了MLP的实现，对神经网络的背景知识介绍不多，感兴趣的朋友可以进一步阅读相应教程 - 译者注）

# MLP模型

MLP模型可以用以下的图来表示：



单隐层的MLP定义了一个映射：

f:RD→RL

，其中 $D$和$L$为输入向量和输出向量$f(x)$的大小。

$f(x)$的数学表达式为：

f(x)=G(b(2)+W(2)(s(b(1)+W(1)x)))

其中$b^{1)}$,$b^{(2)}$为偏差向量，$W^{(1)}$,$W^{(2)}$为权重向量，$G$和$s$为激活函数

向量 $h(x) = \Phi(x) = s(b^{(1)} + W^{(1)} x)$ 定义了隐层。 $W^{(1)} \in R^{D \times D\_h}$为连接输入向量和隐层的权重矩阵。其中每一列表示了输入神经元和一个隐层神经元权重。$s$函数的经典选择包括 tanh, $tanh(a)=(e^a-e^{-a})/(e^a+e^{-a})$，或者符号函数 sigmod， $sigmoid(a)=1/(1+e^{-a})$ 。

模型的输出向量为 $o(x) = G(b^{(2)} + W^{(2)} h(x))$.读者应该记得，该形式在上一节中用过。和之前一样，如果把$G$定义为 softmax函数，输出为类的归属概率。

为了训练MLP模型，我们用随机梯度下降算法学习所有参数，包括  $\theta = \{W^{(2)},b^{(2)},W^{(1)},b^{(1)}\}$。梯度$\partial{\ell}/\partial{\theta}$可以通过BP算法( **backpropagation algorithm**)计算。幸运的是，Theano可以自动的计算差分，再次我们不需要操心此细节。

# 从对数回归模型到多层感知器

本节我们专注于单层的MLP模型。在此，我们首先实现一个表示隐层的类。为了构建MLP模型，我们需要在此之上构建一个对数回归层。

[复制代码](javascript:void(0);)

class HiddenLayer(object):

def \_\_init\_\_(self, rng, input, n\_in, n\_out, activation=T.tanh):

"""

Typical hidden layer of a MLP: units are fully-connected and have

sigmoidal activation function. Weight matrix W is of shape (n\_in,n\_out)

and the bias vector b is of shape (n\_out,).

NOTE : The nonlinearity used here is tanh

Hidden unit activation is given by: tanh(dot(input,W) + b)

:type rng: numpy.random.RandomState

:param rng: a random number generator used to initialize weights

:type input: theano.tensor.dmatrix

:param input: a symbolic tensor of shape (n\_examples, n\_in)

:type n\_in: int

:param n\_in: dimensionality of input

:type n\_out: int

:param n\_out: number of hidden units

:type activation: theano.Op or function

:param activation: Non linearity to be applied in the hidden

layer

"""

self.input = input

[复制代码](javascript:void(0);)

隐层权重的初始值需要从一个和激活函数相关的对称区间上面均匀采样得到。对于tanh函数，采样区间应该为$[-\sqrt{\frac{6}{fan\_{in}+fan\_{out}}},\sqrt{\frac{6}{fan\_{in}+fan\_{out}}}]$ [[Xavier10]](http://deeplearning.net/tutorial/references.html#xavier10). 这里$fan\_{in}$和$fan\_{out}$分别为第(i-1) 和 i层的神经元的数目. 对于sigmoid函数，采样区间为：$[-4\sqrt{\frac{6}{fan\_{in}+fan\_{out}}},4\sqrt{\frac{6}{fan\_{in}+fan\_{out}}}]$。初始化操作能够保证在训练的前期，每个神经元在激活函数的作用下，信息可以更容易地向下向上两个方向进行传播。

[复制代码](javascript:void(0);)

W\_values = numpy.asarray(rng.uniform(

low=-numpy.sqrt(6. / (n\_in + n\_out)),

high=numpy.sqrt(6. / (n\_in + n\_out)),

size=(n\_in, n\_out)), dtype=theano.config.floatX)

if activation == theano.tensor.nnet.sigmoid:

W\_values \*= 4

self.W = theano.shared(value=W\_values, name='W')

b\_values = numpy.zeros((n\_out,), dtype=theano.config.floatX)

self.b = theano.shared(value=b\_values, name='b')

[复制代码](javascript:void(0);)

这里我们要注意到，隐层的激活函数为一个非线性函数。函数缺省为 tanh，但是很多情况下，我们可能用下面的函数

self.output = activation(T.dot(input, self.W) + self.b)

# parameters of the model

self.params = [self.W, self.b]

结合理论知识，这里其实是计算了隐层的输出：$h(x) = \Phi(x) = s(b^{(1)} + W^{(1)} x)$。如果你把这个值当做LogisticRegression类的输入，正好是上节对数回归分类的内容，而且此时的输出正好是MLP的输出。所以一个MLP的简单实现如下：

[复制代码](javascript:void(0);)

class MLP(object):

"""Multi-Layer Perceptron Class

A multilayer perceptron is a feedforward artificial neural network model

that has one layer or more of hidden units and nonlinear activations.

Intermediate layers usually have as activation function tanh or the

sigmoid function (defined here by a ``HiddenLayer`` class) while the

top layer is a softamx layer (defined here by a ``LogisticRegression``

class).

"""

def \_\_init\_\_(self, rng, input, n\_in, n\_hidden, n\_out):

"""Initialize the parameters for the multilayer perceptron

:type rng: numpy.random.RandomState

:param rng: a random number generator used to initialize weights

:type input: theano.tensor.TensorType

:param input: symbolic variable that describes the input of the

architecture (one minibatch)

:type n\_in: int

:param n\_in: number of input units, the dimension of the space in

which the datapoints lie

:type n\_hidden: int

:param n\_hidden: number of hidden units

:type n\_out: int

:param n\_out: number of output units, the dimension of the space in

which the labels lie

"""

# Since we are dealing with a one hidden layer MLP, this will

# translate into a Hidden Layer connected to the LogisticRegression

# layer

self.hiddenLayer = HiddenLayer(rng = rng, input = input,

n\_in = n\_in, n\_out = n\_hidden,

activation = T.tanh)

# The logistic regression layer gets as input the hidden units

# of the hidden layer

self.logRegressionLayer = LogisticRegression(

input=self.hiddenLayer.output,

n\_in=n\_hidden,

n\_out=n\_out)

[复制代码](javascript:void(0);)

在本节中，我们仍然采用$L\_1$和$L\_2$规则化，因此需要计算两层的权重矩阵的规范化的结果。

[复制代码](javascript:void(0);)

# L1 norm ; one regularization option is to enforce L1 norm to

# be small

self.L1 = abs(self.hiddenLayer.W).sum() \

+ abs(self.logRegressionLayer.W).sum()

# square of L2 norm ; one regularization option is to enforce

# square of L2 norm to be small

self.L2\_sqr = (self.hiddenLayer.W \*\* 2).sum() \

+ (self.logRegressionLayer.W \*\* 2).sum()

# negative log likelihood of the MLP is given by the negative

# log likelihood of the output of the model, computed in the

# logistic regression layer

self.negative\_log\_likelihood = self.logRegressionLayer.negative\_log\_likelihood

# same holds for the function computing the number of errors

self.errors = self.logRegressionLayer.errors

# the parameters of the model are the parameters of the two layer it is

# made out of

self.params = self.hiddenLayer.params + self.logRegressionLayer.params

[复制代码](javascript:void(0);)

和之前一样，我们用在mini-batch上面的随机梯度下降算法训练模型。这里的区别在于，我们修改损失函数并包括规范化项。 L1\_reg 和 L2\_reg 为超参数，用以控制规范化项在整个损失函数中的比重。计算损失的函数如下：

# the cost we minimize during training is the negative log likelihood of

# the model plus the regularization terms (L1 and L2); cost is expressed

# here symbolically

cost = classifier.negative\_log\_likelihood(y) \

+ L1\_reg \* L1 \

+ L2\_reg \* L2\_sqr

接下来，模型参数通过梯度更新。这段代码和之前的基本上一样，除了参数多少的差别。

[复制代码](javascript:void(0);)

# compute the gradient of cost with respect to theta (stored in params)

# the resulting gradients will be stored in a list gparams

gparams = []

for param in classifier.params:

gparam = T.grad(cost, param)

gparams.append(gparam)

# specify how to update the parameters of the model as a list of

# (variable, update expression) pairs

updates = []

# given two list the zip A = [a1, a2, a3, a4] and B = [b1, b2, b3, b4] of

# same length, zip generates a list C of same size, where each element

# is a pair formed from the two lists :

# C = [(a1, b1), (a2, b2), (a3, b3) , (a4, b4)]

for param, gparam in zip(classifier.params, gparams):

updates.append((param, param - learning\_rate \* gparam))

# compiling a Theano function `train\_model` that returns the cost, butx

# in the same time updates the parameter of the model based on the rules

# defined in `updates`

train\_model = theano.function(inputs=[index], outputs=cost,

updates=updates,

givens={

x: train\_set\_x[index \* batch\_size:(index + 1) \* batch\_size],

y: train\_set\_y[index \* batch\_size:(index + 1) \* batch\_size]})

[复制代码](javascript:void(0);)

# 功能综合

基于以上基本概念，写一个MLP的类变成了一件非常容易的事情。下面的代码演示了它是如何运作的，其原理和我们之前的对数回归分类器基本一致。

[复制代码](javascript:void(0);)

"""

This tutorial introduces the multilayer perceptron using Theano.

A multilayer perceptron is a logistic regressor where

instead of feeding the input to the logistic regression you insert a

intermediate layer, called the hidden layer, that has a nonlinear

activation function (usually tanh or sigmoid) . One can use many such

hidden layers making the architecture deep. The tutorial will also tackle

the problem of MNIST digit classification.

.. math::

f(x) = G( b^{(2)} + W^{(2)}( s( b^{(1)} + W^{(1)} x))),

References:

- textbooks: "Pattern Recognition and Machine Learning" -

Christopher M. Bishop, section 5

"""

\_\_docformat\_\_ = 'restructedtext en'

import cPickle

import gzip

import os

import sys

import time

import numpy

import theano

import theano.tensor as T

from logistic\_sgd import LogisticRegression, load\_data

class HiddenLayer(object):

def \_\_init\_\_(self, rng, input, n\_in, n\_out, W=None, b=None,

activation=T.tanh):

"""

Typical hidden layer of a MLP: units are fully-connected and have

sigmoidal activation function. Weight matrix W is of shape (n\_in,n\_out)

and the bias vector b is of shape (n\_out,).

NOTE : The nonlinearity used here is tanh

Hidden unit activation is given by: tanh(dot(input,W) + b)

:type rng: numpy.random.RandomState

:param rng: a random number generator used to initialize weights

:type input: theano.tensor.dmatrix

:param input: a symbolic tensor of shape (n\_examples, n\_in)

:type n\_in: int

:param n\_in: dimensionality of input

:type n\_out: int

:param n\_out: number of hidden units

:type activation: theano.Op or function

:param activation: Non linearity to be applied in the hidden

layer

"""

self.input = input

# `W` is initialized with `W\_values` which is uniformely sampled

# from sqrt(-6./(n\_in+n\_hidden)) and sqrt(6./(n\_in+n\_hidden))

# for tanh activation function

# the output of uniform if converted using asarray to dtype

# theano.config.floatX so that the code is runable on GPU

# Note : optimal initialization of weights is dependent on the

# activation function used (among other things).

# For example, results presented in [Xavier10] suggest that you

# should use 4 times larger initial weights for sigmoid

# compared to tanh

# We have no info for other function, so we use the same as

# tanh.

if W is None:

W\_values = numpy.asarray(rng.uniform(

low=-numpy.sqrt(6. / (n\_in + n\_out)),

high=numpy.sqrt(6. / (n\_in + n\_out)),

size=(n\_in, n\_out)), dtype=theano.config.floatX)

if activation == theano.tensor.nnet.sigmoid:

W\_values \*= 4

W = theano.shared(value=W\_values, name='W', borrow=True)

if b is None:

b\_values = numpy.zeros((n\_out,), dtype=theano.config.floatX)

b = theano.shared(value=b\_values, name='b', borrow=True)

self.W = W

self.b = b

lin\_output = T.dot(input, self.W) + self.b

self.output = (lin\_output if activation is None

else activation(lin\_output))

# parameters of the model

self.params = [self.W, self.b]

class MLP(object):

"""Multi-Layer Perceptron Class

A multilayer perceptron is a feedforward artificial neural network model

that has one layer or more of hidden units and nonlinear activations.

Intermediate layers usually have as activation function thanh or the

sigmoid function (defined here by a ``SigmoidalLayer`` class) while the

top layer is a softamx layer (defined here by a ``LogisticRegression``

class).

"""

def \_\_init\_\_(self, rng, input, n\_in, n\_hidden, n\_out):

"""Initialize the parameters for the multilayer perceptron

:type rng: numpy.random.RandomState

:param rng: a random number generator used to initialize weights

:type input: theano.tensor.TensorType

:param input: symbolic variable that describes the input of the

architecture (one minibatch)

:type n\_in: int

:param n\_in: number of input units, the dimension of the space in

which the datapoints lie

:type n\_hidden: int

:param n\_hidden: number of hidden units

:type n\_out: int

:param n\_out: number of output units, the dimension of the space in

which the labels lie

"""

# Since we are dealing with a one hidden layer MLP, this will

# translate into a TanhLayer connected to the LogisticRegression

# layer; this can be replaced by a SigmoidalLayer, or a layer

# implementing any other nonlinearity

self.hiddenLayer = HiddenLayer(rng=rng, input=input,

n\_in=n\_in, n\_out=n\_hidden,

activation=T.tanh)

# The logistic regression layer gets as input the hidden units

# of the hidden layer

self.logRegressionLayer = LogisticRegression(

input=self.hiddenLayer.output,

n\_in=n\_hidden,

n\_out=n\_out)

# L1 norm ; one regularization option is to enforce L1 norm to

# be small

self.L1 = abs(self.hiddenLayer.W).sum() \

+ abs(self.logRegressionLayer.W).sum()

# square of L2 norm ; one regularization option is to enforce

# square of L2 norm to be small

self.L2\_sqr = (self.hiddenLayer.W \*\* 2).sum() \

+ (self.logRegressionLayer.W \*\* 2).sum()

# negative log likelihood of the MLP is given by the negative

# log likelihood of the output of the model, computed in the

# logistic regression layer

self.negative\_log\_likelihood = self.logRegressionLayer.negative\_log\_likelihood

# same holds for the function computing the number of errors

self.errors = self.logRegressionLayer.errors

# the parameters of the model are the parameters of the two layer it is

# made out of

self.params = self.hiddenLayer.params + self.logRegressionLayer.params

def test\_mlp(learning\_rate=0.01, L1\_reg=0.00, L2\_reg=0.0001, n\_epochs=1000,

dataset='../data/mnist.pkl.gz', batch\_size=20, n\_hidden=500):

"""

Demonstrate stochastic gradient descent optimization for a multilayer

perceptron

This is demonstrated on MNIST.

:type learning\_rate: float

:param learning\_rate: learning rate used (factor for the stochastic

gradient

:type L1\_reg: float

:param L1\_reg: L1-norm's weight when added to the cost (see

regularization)

:type L2\_reg: float

:param L2\_reg: L2-norm's weight when added to the cost (see

regularization)

:type n\_epochs: int

:param n\_epochs: maximal number of epochs to run the optimizer

:type dataset: string

:param dataset: the path of the MNIST dataset file from

http://www.iro.umontreal.ca/~lisa/deep/data/mnist/mnist.pkl.gz

"""

datasets = load\_data(dataset)

train\_set\_x, train\_set\_y = datasets[0]

valid\_set\_x, valid\_set\_y = datasets[1]

test\_set\_x, test\_set\_y = datasets[2]

# compute number of minibatches for training, validation and testing

n\_train\_batches = train\_set\_x.get\_value(borrow=True).shape[0] / batch\_size

n\_valid\_batches = valid\_set\_x.get\_value(borrow=True).shape[0] / batch\_size

n\_test\_batches = test\_set\_x.get\_value(borrow=True).shape[0] / batch\_size

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

# BUILD ACTUAL MODEL #

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

print '... building the model'

# allocate symbolic variables for the data

index = T.lscalar() # index to a [mini]batch

x = T.matrix('x') # the data is presented as rasterized images

y = T.ivector('y') # the labels are presented as 1D vector of

# [int] labels

rng = numpy.random.RandomState(1234)

# construct the MLP class

classifier = MLP(rng=rng, input=x, n\_in=28 \* 28,

n\_hidden=n\_hidden, n\_out=10)

# the cost we minimize during training is the negative log likelihood of

# the model plus the regularization terms (L1 and L2); cost is expressed

# here symbolically

cost = classifier.negative\_log\_likelihood(y) \

+ L1\_reg \* classifier.L1 \

+ L2\_reg \* classifier.L2\_sqr

# compiling a Theano function that computes the mistakes that are made

# by the model on a minibatch

test\_model = theano.function(inputs=[index],

outputs=classifier.errors(y),

givens={

x: test\_set\_x[index \* batch\_size:(index + 1) \* batch\_size],

y: test\_set\_y[index \* batch\_size:(index + 1) \* batch\_size]})

validate\_model = theano.function(inputs=[index],

outputs=classifier.errors(y),

givens={

x: valid\_set\_x[index \* batch\_size:(index + 1) \* batch\_size],

y: valid\_set\_y[index \* batch\_size:(index + 1) \* batch\_size]})

# compute the gradient of cost with respect to theta (sotred in params)

# the resulting gradients will be stored in a list gparams

gparams = []

for param in classifier.params:

gparam = T.grad(cost, param)

gparams.append(gparam)

# specify how to update the parameters of the model as a list of

# (variable, update expression) pairs

updates = []

# given two list the zip A = [a1, a2, a3, a4] and B = [b1, b2, b3, b4] of

# same length, zip generates a list C of same size, where each element

# is a pair formed from the two lists :

# C = [(a1, b1), (a2, b2), (a3, b3), (a4, b4)]

for param, gparam in zip(classifier.params, gparams):

updates.append((param, param - learning\_rate \* gparam))

# compiling a Theano function `train\_model` that returns the cost, but

# in the same time updates the parameter of the model based on the rules

# defined in `updates`

train\_model = theano.function(inputs=[index], outputs=cost,

updates=updates,

givens={

x: train\_set\_x[index \* batch\_size:(index + 1) \* batch\_size],

y: train\_set\_y[index \* batch\_size:(index + 1) \* batch\_size]})

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

# TRAIN MODEL #

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

print '... training'

# early-stopping parameters

patience = 10000 # look as this many examples regardless

patience\_increase = 2 # wait this much longer when a new best is

# found

improvement\_threshold = 0.995 # a relative improvement of this much is

# considered significant

validation\_frequency = min(n\_train\_batches, patience / 2)

# go through this many

# minibatche before checking the network

# on the validation set; in this case we

# check every epoch

best\_params = None

best\_validation\_loss = numpy.inf

best\_iter = 0

test\_score = 0.

start\_time = time.clock()

epoch = 0

done\_looping = False

while (epoch < n\_epochs) and (not done\_looping):

epoch = epoch + 1

for minibatch\_index in xrange(n\_train\_batches):

minibatch\_avg\_cost = train\_model(minibatch\_index)

# iteration number

iter = (epoch - 1) \* n\_train\_batches + minibatch\_index

if (iter + 1) % validation\_frequency == 0:

# compute zero-one loss on validation set

validation\_losses = [validate\_model(i) for i

in xrange(n\_valid\_batches)]

this\_validation\_loss = numpy.mean(validation\_losses)

print('epoch %i, minibatch %i/%i, validation error %f %%' %

(epoch, minibatch\_index + 1, n\_train\_batches,

this\_validation\_loss \* 100.))

# if we got the best validation score until now

if this\_validation\_loss < best\_validation\_loss:

#improve patience if loss improvement is good enough

if this\_validation\_loss < best\_validation\_loss \* \

improvement\_threshold:

patience = max(patience, iter \* patience\_increase)

best\_validation\_loss = this\_validation\_loss

best\_iter = iter

# test it on the test set

test\_losses = [test\_model(i) for i

in xrange(n\_test\_batches)]

test\_score = numpy.mean(test\_losses)

print((' epoch %i, minibatch %i/%i, test error of '

'best model %f %%') %

(epoch, minibatch\_index + 1, n\_train\_batches,

test\_score \* 100.))

if patience <= iter:

done\_looping = True

break

end\_time = time.clock()

print(('Optimization complete. Best validation score of %f %% '

'obtained at iteration %i, with test performance %f %%') %

(best\_validation\_loss \* 100., best\_iter + 1, test\_score \* 100.))

print >> sys.stderr, ('The code for file ' +

os.path.split(\_\_file\_\_)[1] +

' ran for %.2fm' % ((end\_time - start\_time) / 60.))

if \_\_name\_\_ == '\_\_main\_\_':

test\_mlp()

[复制代码](javascript:void(0);)

[**theano学习指南4（翻译）- 卷积神经网络**](http://www.cnblogs.com/xueliangliu/archive/2013/06/09/3127197.html)

动机

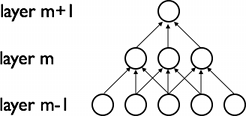
卷积神经网络是一种特殊的MLP,这个概念是从生物里面演化过来的. 根据Hubel和Wiesel早期在猫的视觉皮层上的工作 [[Hubel68]](http://deeplearning.net/tutorial/references.html#hubel68), 我们知道在视觉皮层上面存在一种细胞的复杂分布,这些细胞对一些局部输入是很敏感的,它们被成为感知野, 并通过这种特殊的组合方式来覆盖整个视野. 这些过滤器对输入空间是局部敏感的,因此能够更好得发觉自然图像中不同物体的空间相关性.

进一步讲, 视觉皮层存在两类不同的细胞,简单细胞S和复杂细胞C. 简单细胞尽可能得可视野中特殊的类似边缘这种结构进行相应.复杂细胞具有更大的感知范围,它们可以对刺激的空间位置进行精确的定位.

作为已知的最强大的视觉系统,视觉皮层也成为了科学研究的对象. 很多神经科学中提出的模型,都是基于对其进行的研究,比如, NeoCognitron [[Fukushima]](http://deeplearning.net/tutorial/references.html#fukushima), HMAX [[Serre07]](http://deeplearning.net/tutorial/references.html#serre07) 以及本文讨论的重点 LeNet-5 [[LeCun98]](http://deeplearning.net/tutorial/references.html#lecun98)

**稀疏连接性**

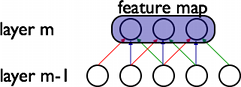
CNN通过增强相邻两层中神经元的局部的连接来发掘局部空间相关性. m层的隐输入单元和m-1层的一部分空间相邻,并具有连续可视野的神经元相连接. 它们的关系如下图所示:



我们可以假设m-1层为输入视网膜, 在它之上,m层的视觉神经元具有宽度为3的可视野,因此一个单元可以连接视网膜层的三个相邻的神经元. m层的神经元和m-1层具有类似的连接属性. 因此m+1层的神经元对于m层,仍具有宽度为3的可视野,但是相对于m-1层,可视野的宽度更大(结果为5). 这种结构把训练好的过滤器构建成一种局部空间模式. 如上图所示, 过滤器由多个感知层堆积而成,它变得更加地全局. 比如,m+1层的一个神经元可以对m-1层的宽度为5的特征进行编码.

**共享权重**

在CNN中,每一个稀疏的过滤器$h\_i$在整个可视野上是叠加的重复的. 这些重复的单元形成了一种特征图,它可以共享相同的参数,比如共同的权向量和偏差.



上图中, 属于同一个的特征图的三个隐单元,因为需要共享相同颜色的权重, 他们的被限制成相同的. 梯度下降算法,在进行了一个轻微的改动之后, 仍然可以用来学习这些共享的参数.  共享权重的梯度可以对共享参数的梯度进行简单的求和得到.

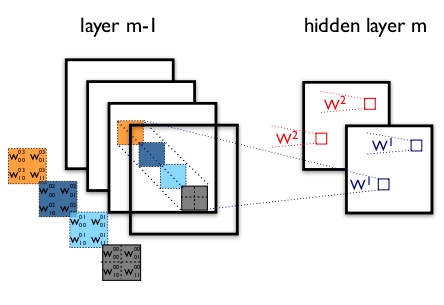
为什么要对共享权重感兴趣呢? 在这种方式中,重复单元可以检测特征,无论他们在可视野中的位置在什么地方. 而权重的共享为此提供了一种非常有效的方法, 因为这样做可以在很大程度上减少需要学习的参数. 通过控制模型的容量,CNN在视觉问题上达到了更好的泛化.

**具体细节**

从概念上讲,特征图通过对输入图像在一个线性滤波器上的卷积运算,增加一个便宜量,在结果上作用一个非线性函数得到.如果我们把某层的第k个的特征图记为$h^k$,其过滤器由权重$W$和偏移量$b\_k$决定, 那么,特征图可以通过下面的函数得到:

$$h^k\_{ij} = tanh ( (W^k \* x)\_{ij} + b\_k ) $$

为了更好的表达数据, 隐层由一系列的多个特征图构成${h^{(k)}, k= 0 .. K}$. 其权重$W$由四个参数决定: 目标特征图的索引,源特征图的索引,源水平位置索引和源垂直位置索引. 偏移量为一个向量,其中每一个元素对应目标特征图的一个索引. 其逻辑关系通过下图表示:



**Figure 1**: 卷积层实例 (这个图和下面的说明有点冲突,下面的特征权重表示成了$W^0$,$W^1$,图中是 $W^1$,$W^2$)

这里是一个两层的CNN,它有 m-1层的四个特征图和m层的两个特征图($h^0, h^1$)构成. 神经元在$h^0$和$h^1$的输出(蓝色和红色的框所示)是由m-1层落入其相应的2\*2的可视野的像素计算得到, 这里需要注意可视野如何地跨四个特征图.其权重为3D张量,分别表示了输入特征图的索引,以及像素的坐标.

整合以上概念, $W\_{ij}^{kl}$表示了连接m层第k个特征图的特征图上每一个像素的权重, 像素为m-1层的第l个特征图,其位置为 $(i,j)$.

**ConvOp**

Convop是Theano中实现卷积的函数, 它主要重复了scipy工具包中signal.convolve2d的函数功能. 总的来讲,ConvOp包含两个参数:

* 对应输入图像的mini-batch的4D张量. 其每个张量的大小为:[mini-batch的大小, 输入的特征图的数量, 图像的高度,图像的宽度]
* 对应权重矩阵$W$的4D张量,其每个张量的大小为:[m层的特征图的数量,m-1层的特征图的数量,过滤器的高度,过滤器的宽度].

下面的代码实现了一个类似图1里面的卷积层. 输入图像包括大小为120\*160的三个特征图(对应RGB). 我们可以用两个具有9\*9的可视野的卷积过滤器.

[复制代码](javascript:void(0);)

from theano.tensor.nnet import conv

rng = numpy.random.RandomState(23455)

# instantiate 4D tensor for input

input = T.tensor4(name='input')

# initialize shared variable for weights.

w\_shp = (2, 3, 9, 9)

w\_bound = numpy.sqrt(3 \* 9 \* 9)

W = theano.shared( numpy.asarray(

rng.uniform(

low=-1.0 / w\_bound,

high=1.0 / w\_bound,

size=w\_shp),

dtype=input.dtype), name ='W')

# initialize shared variable for bias (1D tensor) with random values

# IMPORTANT: biases are usually initialized to zero. However in this

# particular application, we simply apply the convolutional layer to

# an image without learning the parameters. We therefore initialize

# them to random values to "simulate" learning.

b\_shp = (2,)

b = theano.shared(numpy.asarray(

rng.uniform(low=-.5, high=.5, size=b\_shp),

dtype=input.dtype), name ='b')

# build symbolic expression that computes the convolution of input with filters in w

conv\_out = conv.conv2d(input, W)

# build symbolic expression to add bias and apply activation function, i.e. produce neural net layer output

# A few words on ``dimshuffle`` :

# ``dimshuffle`` is a powerful tool in reshaping a tensor;

# what it allows you to do is to shuffle dimension around

# but also to insert new ones along which the tensor will be

# broadcastable;

# dimshuffle('x', 2, 'x', 0, 1)

# This will work on 3d tensors with no broadcastable

# dimensions. The first dimension will be broadcastable,

# then we will have the third dimension of the input tensor as

# the second of the resulting tensor, etc. If the tensor has

# shape (20, 30, 40), the resulting tensor will have dimensions

# (1, 40, 1, 20, 30). (AxBxC tensor is mapped to 1xCx1xAxB tensor)

# More examples:

# dimshuffle('x') -> make a 0d (scalar) into a 1d vector

# dimshuffle(0, 1) -> identity

# dimshuffle(1, 0) -> inverts the first and second dimensions

# dimshuffle('x', 0) -> make a row out of a 1d vector (N to 1xN)

# dimshuffle(0, 'x') -> make a column out of a 1d vector (N to Nx1)

# dimshuffle(2, 0, 1) -> AxBxC to CxAxB

# dimshuffle(0, 'x', 1) -> AxB to Ax1xB

# dimshuffle(1, 'x', 0) -> AxB to Bx1xA

output = T.nnet.sigmoid(conv\_out + b.dimshuffle('x', 0, 'x', 'x'))

# create theano function to compute filtered images

f = theano.function([input], output)

[复制代码](javascript:void(0);)

首先我们用得到的函数f做点有意思的事情.

[复制代码](javascript:void(0);)

import pylab

from PIL import Image

# open random image of dimensions 639x516

img = Image.open(open('images/3wolfmoon.jpg'))

img = numpy.asarray(img, dtype='float64') / 256.

# put image in 4D tensor of shape (1, 3, height, width)

img\_ = img.swapaxes(0, 2).swapaxes(1, 2).reshape(1, 3, 639, 516)

filtered\_img = f(img\_)

# plot original image and first and second components of output

pylab.subplot(1, 3, 1); pylab.axis('off'); pylab.imshow(img)

pylab.gray();

# recall that the convOp output (filtered image) is actually a "minibatch",

# of size 1 here, so we take index 0 in the first dimension:

pylab.subplot(1, 3, 2); pylab.axis('off'); pylab.imshow(filtered\_img[0, 0, :, :])

pylab.subplot(1, 3, 3); pylab.axis('off'); pylab.imshow(filtered\_img[0, 1, :, :])

pylab.show()

[复制代码](javascript:void(0);)

运行代码，可以得到如下结果：



我们可以注意到，随机初始化的滤波器能够产生边缘检测算子的作用。另外，我们用和MLP中相同的权重对公式进行初始化。这些权重是从均匀分布[-1/fan-in, 1/fan-in]随机采样得到的。这里 fan-in是输入层到隐层单元的数量。对于MLP来说，这正是下一层的单元的数目。而对于CNNs，我们需要考虑到输入特征图的数量，以及可视野的大小。

**共用最大化**

CNN的另外一个重要特征是共用最大化，这其实是一种非线性向下采样的方法。共用最大化把输入图像分割成不重叠的矩形，然后对于每个矩形区域，输出最大化的结果。

这个技术在视觉上的好处主要有两个方面 （1）它降低了上层的计算复杂度 （2）它提供了一种变换不变量的。对于第二种益处，我们可以假设把一个共用最大化层和一个卷积层组合起来，对于单个像素，输入图像可以有8个方向的变换。如果共有最大层在2\*2的窗口上面实现，这8个可能的配置中，有3个可以准确的产生和卷积层相同的结果。如果窗口变成3\*3，则产生精确结果的概率变成了5／８．

可见，共有最大化对位置信息提供了附加的鲁棒性，它以一种非常聪明的方式减少了中间表示的维度。

在Theano中，这种技术通过函数 theano.tensor.signal.downsample.max\_pool\_2d 实现，这个函数的输入是一个N维张量（N>2), 和一个缩放因子来对这个张量进行共用最大化的变换。下面的例子说明了这个过程：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | from theano.tensor.signal import downsample    input = T.dtensor4('input')  maxpool\_shape = (2, 2)  pool\_out = downsample.max\_pool\_2d(input, maxpool\_shape, ignore\_border=True)  f = theano.function([input],pool\_out)    invals = numpy.random.RandomState(1).rand(3, 2, 5, 5)  print 'With ignore\_border set to True:'  print 'invals[0, 0, :, :] =\n', invals[0, 0, :, :]  print 'output[0, 0, :, :] =\n', f(invals)[0, 0, :, :]    pool\_out = downsample.max\_pool\_2d(input, maxpool\_shape, ignore\_border=False)  f = theano.function([input],pool\_out)  print 'With ignore\_border set to False:'  print 'invals[1, 0, :, :] =\n ', invals[1, 0, :, :]  print 'output[1, 0, :, :] =\n ', f(invals)[1, 0, :, :] |

　　这段代码的输出为类似下面的内容：

[复制代码](javascript:void(0);)

With ignore\_border set to True:

invals[0, 0, :, :] =

[[ 4.17022005e-01 7.20324493e-01 1.14374817e-04 3.02332573e-01 1.46755891e-01]

[ 9.23385948e-02 1.86260211e-01 3.45560727e-01 3.96767474e-01 5.38816734e-01]

[ 4.19194514e-01 6.85219500e-01 2.04452250e-01 8.78117436e-01 2.73875932e-02]

[ 6.70467510e-01 4.17304802e-01 5.58689828e-01 1.40386939e-01 1.98101489e-01]

[ 8.00744569e-01 9.68261576e-01 3.13424178e-01 6.92322616e-01 8.76389152e-01]]

output[0, 0, :, :] =

[[ 0.72032449 0.39676747]

[ 0.6852195 0.87811744]]

With ignore\_border set to False:

invals[1, 0, :, :] =

[[ 0.01936696 0.67883553 0.21162812 0.26554666 0.49157316]

[ 0.05336255 0.57411761 0.14672857 0.58930554 0.69975836]

[ 0.10233443 0.41405599 0.69440016 0.41417927 0.04995346]

[ 0.53589641 0.66379465 0.51488911 0.94459476 0.58655504]

[ 0.90340192 0.1374747 0.13927635 0.80739129 0.39767684]]

output[1, 0, :, :] =

[[ 0.67883553 0.58930554 0.69975836]

[ 0.66379465 0.94459476 0.58655504]

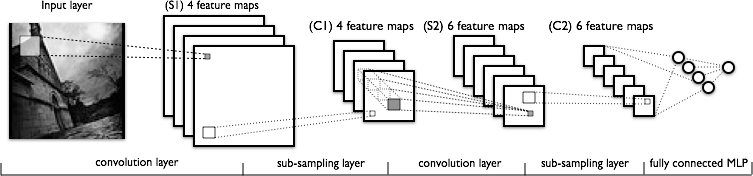
[ 0.90340192 0.80739129 0.39767684]]

[复制代码](javascript:void(0);)

注意到和大部分代码不同的是，这个函数max\_pool\_2d 在创建Theano图的时候，需要一个向下采样的因子ds (长度为2的tuple变量，表示了图像的宽和高的缩放. 这个可能在以后的版本中升级。

**LeNet模型**

稀疏，卷积层和共有最大化是LeNet的核心概念。因为模型的细节会有很大的变换，我们用下面的图来诠释LeNet的模型。



模型的低层由卷积和共有最大化层组成，高层是全连接的一个MLP 神经网络，它包含了隐层和对数回归。高层的输入是下层特征图的结合。

从实现的角度讲，这意味着低层操作了4D的张量，这个张量被压缩到了一个2D矩阵表示的光栅化的特征图上，以便于和前面的MLP的实现兼容。

**综合所有**

现在我们有了实现LeNet模型的所有细节，我们创建一个LeNetConvPoolLayer类，用了表示一个卷积和共有最大化层：

[复制代码](javascript:void(0);)

class LeNetConvPoolLayer(object):

def \_\_init\_\_(self, rng, input, filter\_shape, image\_shape, poolsize=(2, 2)):

"""

Allocate a LeNetConvPoolLayer with shared variable internal parameters.

:type rng: numpy.random.RandomState

:param rng: a random number generator used to initialize weights

:type input: theano.tensor.dtensor4

:param input: symbolic image tensor, of shape image\_shape

:type filter\_shape: tuple or list of length 4

:param filter\_shape: (number of filters, num input feature maps,

filter height,filter width)

:type image\_shape: tuple or list of length 4

:param image\_shape: (batch size, num input feature maps,

image height, image width)

:type poolsize: tuple or list of length 2

:param poolsize: the downsampling (pooling) factor (#rows,#cols)

"""

assert image\_shape[1] == filter\_shape[1]

self.input = input

# initialize weight values: the fan-in of each hidden neuron is

# restricted by the size of the receptive fields.

fan\_in = numpy.prod(filter\_shape[1:])

W\_values = numpy.asarray(rng.uniform(

low=-numpy.sqrt(3./fan\_in),

high=numpy.sqrt(3./fan\_in),

size=filter\_shape), dtype=theano.config.floatX)

self.W = theano.shared(value=W\_values, name='W')

# the bias is a 1D tensor -- one bias per output feature map

b\_values = numpy.zeros((filter\_shape[0],), dtype=theano.config.floatX)

self.b = theano.shared(value=b\_values, name='b')

# convolve input feature maps with filters

conv\_out = conv.conv2d(input, self.W,

filter\_shape=filter\_shape, image\_shape=image\_shape)

# downsample each feature map individually, using maxpooling

pooled\_out = downsample.max\_pool\_2d(conv\_out, poolsize, ignore\_border=True)

# add the bias term. Since the bias is a vector (1D array), we first

# reshape it to a tensor of shape (1, n\_filters, 1, 1). Each bias will thus

# be broadcasted across mini-batches and feature map width & height

self.output = T.tanh(pooled\_out + self.b.dimshuffle('x', 0, 'x', 'x'))

# store parameters of this layer

self.params = [self.W, self.b]

[复制代码](javascript:void(0);)

应该注意的是，在初始化权重的时候，fan-in是由感知野的大小和输入特征图的数目决定的。

最后，采用前面章节定义的LogisticRegression和HiddenLayer类，LeNet就可以工作了。

[复制代码](javascript:void(0);)

class LeNetConvPoolLayer(object):

def \_\_init\_\_(self, rng, input, filter\_shape, image\_shape, poolsize=(2, 2)):

"""

Allocate a LeNetConvPoolLayer with shared variable internal parameters.

:type rng: numpy.random.RandomState

:param rng: a random number generator used to initialize weights

:type input: theano.tensor.dtensor4

:param input: symbolic image tensor, of shape image\_shape

:type filter\_shape: tuple or list of length 4

:param filter\_shape: (number of filters, num input feature maps,

filter height,filter width)

:type image\_shape: tuple or list of length 4

:param image\_shape: (batch size, num input feature maps,

image height, image width)

:type poolsize: tuple or list of length 2

:param poolsize: the downsampling (pooling) factor (#rows,#cols)

"""

assert image\_shape[1] == filter\_shape[1]

self.input = input

# initialize weight values: the fan-in of each hidden neuron is

# restricted by the size of the receptive fields.

fan\_in = numpy.prod(filter\_shape[1:])

W\_values = numpy.asarray(rng.uniform(

low=-numpy.sqrt(3./fan\_in),

high=numpy.sqrt(3./fan\_in),

size=filter\_shape), dtype=theano.config.floatX)

self.W = theano.shared(value=W\_values, name='W')

# the bias is a 1D tensor -- one bias per output feature map

b\_values = numpy.zeros((filter\_shape[0],), dtype=theano.config.floatX)

self.b = theano.shared(value=b\_values, name='b')

# convolve input feature maps with filters

conv\_out = conv.conv2d(input, self.W,

filter\_shape=filter\_shape, image\_shape=image\_shape)

# downsample each feature map individually, using maxpooling

pooled\_out = downsample.max\_pool\_2d(conv\_out, poolsize, ignore\_border=True)

# add the bias term. Since the bias is a vector (1D array), we first

# reshape it to a tensor of shape (1, n\_filters, 1, 1). Each bias will thus

# be broadcasted across mini-batches and feature map width & height

self.output = T.tanh(pooled\_out + self.b.dimshuffle('x', 0, 'x', 'x'))

# store parameters of this layer

self.params = [self.W, self.b]

[复制代码](javascript:void(0);)

这里我们忽略了具体的训练和提前结束的代码，这些代码和前面MLP里面的是完全一样的。感兴趣的读者可以查阅DeeplearningTutoirals下面code目录的代码。

**运行算法**

算法运行很简单，通过一个命令：

python code/convolutional\_mlp.py

下面的结果为在i7-2600K CPU的机器上面，采用默认参数和‘floatX=float32’的输出

Optimization complete.

Best validation score of 0.910000 % obtained at iteration 17800,with test

performance 0.920000 %

The code for file convolutional\_mlp.py ran for 380.28m

在GeForce GTX 285的平台上面，结果略有不同

Optimization complete.

Best validation score of 0.910000 % obtained at iteration 15500,with test

performance 0.930000 %

The code for file convolutional\_mlp.py ran for 46.76m

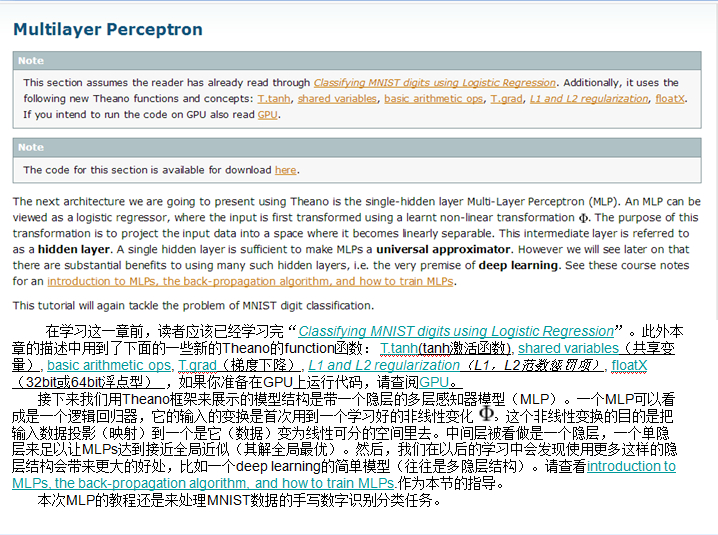
结果中的细小差别来自于不同硬件下不同的圆整机制，这些差别可以忽略。

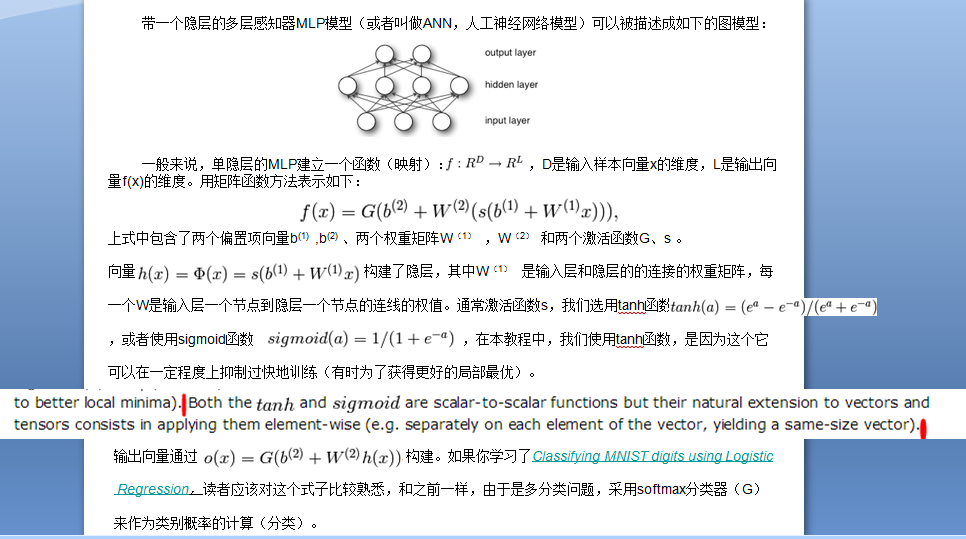
## [Deep learning with Theano 官方中文教程（翻译）（三）——多层感知机（MLP）](http://www.cnblogs.com/charleshuang/p/3648804.html)

关于更多的<http://deeplearning.net/tutorial/>的翻译还有学习笔记会陆续整理传到博客。

供大家相互交流和学习，本人水平有限，若有各种大小错误，还请巨牛大牛小牛微牛们立马拍砖，这样才能共同进步！若引用译文请注明出处<http://www.cnblogs.com/charleshuang/>。

下面。http://deeplearning.net/tutorial/mlp.html#mlp  的中文翻译。下面以PPT截图的方式给出，风格不好，还请见谅。



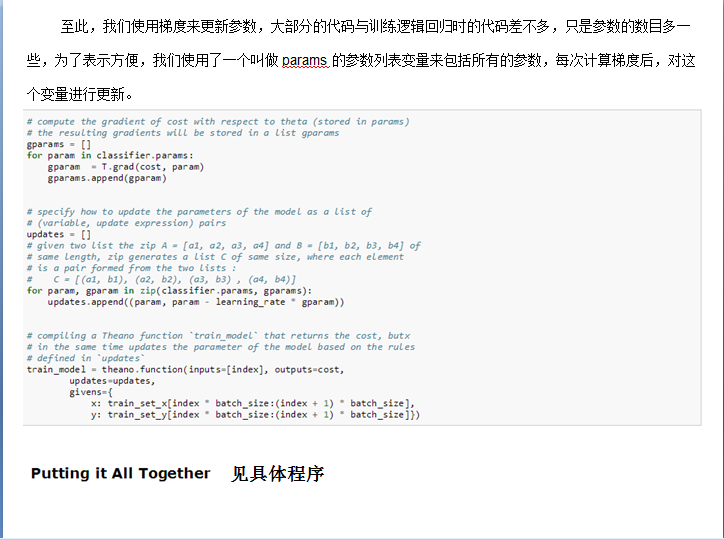






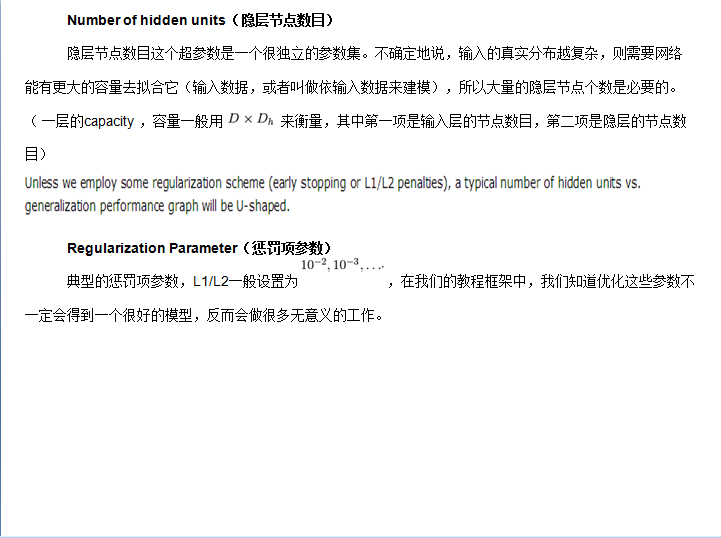












具体程序见 <http://deeplearning.net/tutorial/mlp.html#mlp>

## [Deep learning with Theano 官方中文教程（翻译）（四）—— 卷积神经网络（CNN）](http://www.cnblogs.com/charleshuang/p/3651843.html)

供大家相互交流和学习，本人水平有限，若有各种大小错误，还请巨牛大牛小牛微牛们立马拍砖，这样才能共同进步！若引用译文请注明出处<http://www.cnblogs.com/charleshuang/>。

 本文译自：<http://deeplearning.net/tutorial/lenet.html>

文章中的代码截图不是很清晰，可以去上面的原文网址去查看。

1、动机

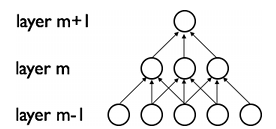
   卷积神经网络（CNN）是多层感知机（MLP）的一个变种模型，它是从生物学概念中演化而来的。从Hubel和Wiesel早期对猫的视觉皮层的研究工作，我们知道在视觉皮层存在一种细胞的复杂分布，，这些细胞对于外界的输入局部是很敏感的，它们被称为“感受野”（细胞），它们以某种方法来覆盖整个视觉域。这些细胞就像一些滤波器一样，它们对输入的图像是局部敏感的，因此能够更好地挖掘出自然图像中的目标的空间关系信息。

   此外，视觉皮层存在两类相关的细胞，S细胞（Simple Cell）和C（Complex Cell）细胞。S细胞在自身的感受野内最大限度地对图像中类似边缘模式的刺激做出响应，而C细胞具有更大的感受野，它可以对图像中产生刺激的模式的空间位置进行精准地定位。

   视觉皮层作为目前已知的最为强大的视觉系统，广受关注。学术领域出现了很多基于它的神经启发式模型。比如：NeoCognitron [[Fukushima]](http://www.cnblogs.com/charleshuang/p/3651843.html#fukushima), HMAX [[Serre07]](http://deeplearning.net/tutorial/references.html#serre07) 以及本教程要讨论的重点 LeNet-5 [[LeCun98]](http://www.cnblogs.com/charleshuang/p/3651843.html#lecun98)。

2、稀疏连接

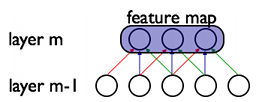
   CNNs通过加强神经网络中相邻层之间节点的局部连接模式（Local Connectivity Pattern）来挖掘自然图像（中的兴趣目标）的空间局部关联信息。第m层隐层的节点与第m-1层的节点的局部子集，并具有空间连续视觉感受野的节点（就是m-1层节点中的一部分，这部分节点在m-1层都是相邻的）相连。可以用下面的图来表示这种连接。



   假设，m-1层为视网膜输入层（接受自然图像）。根据上图的描述，在m-1层上面的m层的神经元节点都具有宽度为3的感受野，m层每一个节点连接下面的视网膜层的3个相邻的节点。m+1层的节点与它下面一层的节点有着相似的连接属性，所以m+1层的节点仍与m层中3个相邻的节点相连，但是对于输入层（视网膜层）连接数就变多了，在本图中是5。这种结构把训练好的滤波器（corresponding to the input producing the strongest response）构建成了一种空间局部模式（因为每个上层节点都只对感受野中的，连接的局部的下层节点有响应）。根据上面图，多层堆积形成了滤波器（不再是线性的了），它也变得更具有全局性了（如包含了一大片的像素空间）。比如，在上图中，第m+1层能够对宽度为5的非线性特征进行编码（就像素空间而言）。

3、权值共享

   在CNNs中，每一个稀疏滤波器hi在整个感受野中是重复叠加的，这些重复的节点形式了一种特征图（feature map）,这个特种图可以共享相同的参数，比如相同的权值矩阵和偏置向量。



   在上图中，属于同一个特征图的三个隐层节点，因为需要共享相同颜色的权重, 他们的被限制成相同的。在这里， 梯度下降算法仍然可以用来训练这些共享的参数，只需要在原算法的基础上稍作改动即可。共享权重的梯度可以对共享参数的梯度进行简单的求和得到。

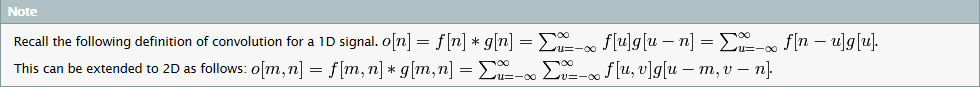
   为什么对权值共享如此感兴趣呢？无论重复单元在感受野的什么位置，他们都可以检测到特征。此外，权值共享提供了一种高效的方式来实现这个，因为这种方式大大减少了需要学习（训练）的参数数目。如果控制好模型的容量，CNN在解决计算机视觉问题上会有更好的泛化能力。

4、详细说明和标注说明

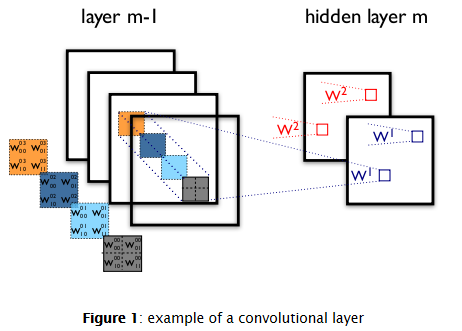
   从概念上来说，特征图是通过对输入图像在一个线性滤波器上做卷积，增加一个偏置项，在此结果上再作用一个非线性函数得到的。如果我们把某一层的第k个特征图记为hk，它的滤波器由权值Wk和偏置bk所决定，所以特征图的由下面公式定义（非线性函数取tanh）：

http://images.cnitblog.com/i/45621/201404/081346436532657.png

卷积说明：



为了更好地表示数据，隐层由多个特征图构成，{hk,k=1,2,3 ...K}.权值W由4个参数决定（目标特征图的索引、源特征图的索引、源垂直位置的索引、源水平位置的索引）（可以说W是一个4维的张量），偏置b为一个向量,向量中的每一个元素对应一个特征图的索引。我们用下图来表示：



上图是一个包含2层神经元节点的CNN，包括m-1层的4个特征图和m层的2个特征图（h0,h1）。神经元在h0,h1的输出（像素）是由m-1层中在其2\*2的感受野中的像素计算得到的。这里注意感受野是如何跨越4个特征图的，权值W0，W1是3维的张量（3D tensor），一个表示输入特征图的索引，另外两个表示像素坐标。总的来说，表示连接第m层第k个特征图的特征图上的每一个像素的权重，与之连接的是m-1层上第l个特征图中坐标为（i,j）的像素。

5、ConvOp

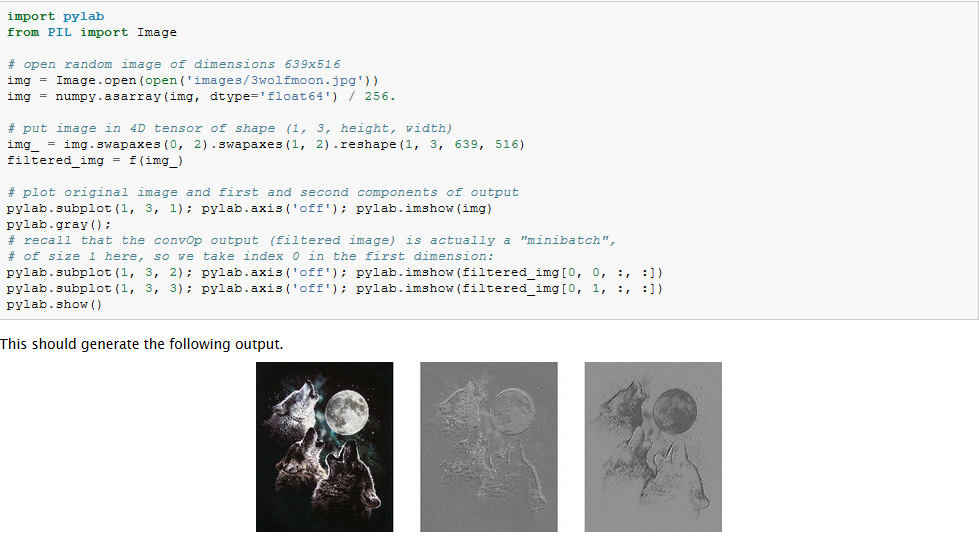
   ConvOp是Theano中对卷积层的一个实现。它重复了Scipy中scipy.signal.convolve2d的函数功能，总的来讲,ConvOp包含了两个输入（参数）：

   （1）对应输入图像的mini-batch的4D张量。每个张量的大小为：[mini-batch的大小，输入的特征图的数量，图像的高度，图像的宽度]。

   （2）对应于权值W的4D张量。每个张量的大小为：[m层的特征图数量，m-1层的特征图数量，滤波器的高度，滤波器的宽度]。

   下面代码实现了Figure 1中的卷积层，输入包括了大小为120\*160的3个特征图(对应RGB). 我们可以用两个具有9\*9的感受野的卷积过滤器。





我们发现，随机初始化的的滤波器能够产生边缘检测算子的作用。

另外，我们使用了相同的公式对权值进行初始化，这些权值都是从一个范围为[-1/fan-in, 1/fan-in]的均匀分布中采样得到的，fan-in是隐层的输入节点数目，在MLP中，这个fan-in就是下面那一层的节点数目，然而对于CNN来说，我们需要考虑到输入特征图的数量，以及感受野的大小。

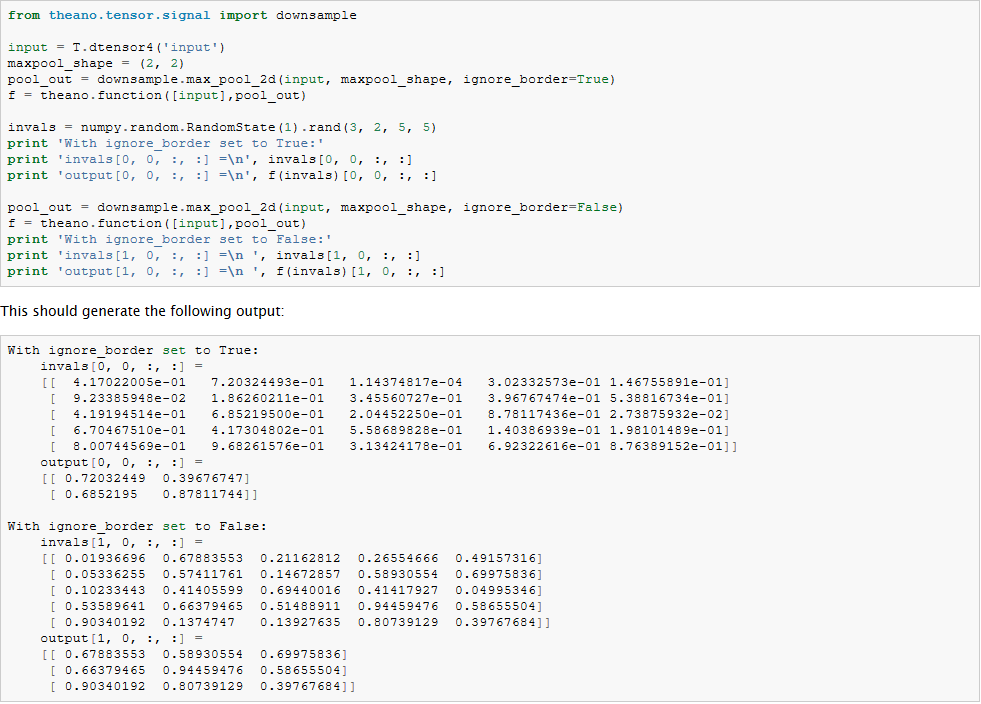
6、最大池化

   CNNs中另一个很重要的概念是最大池化（max-pooling），这是一种非线性的下采样的方法。最大池化把输入图像分割成为不重叠的矩阵，每一个子区域（矩形区域），都输出最大值。

   最大池化技术在视觉问题中是很有用的，原因有两个：（1）降低了上层的计算复杂度。（2）提出了一种变化的不变性形式。为了理解这种不变性，我们假设把最大池化层和一个卷基层结合起来，对于单个像素，有8个变换的方向，如果共有最大层在2\*2的窗口上面实现，这8个可能的配置中，有3个可以准确的产生和卷积层相同的结果。如果窗口变成3\*3，则产生精确结果的概率变成了5／８。

   因此，它对于位移变化有着不错的鲁棒性，最大池化用一种很灵活的方式降低了中间表示层的维度。

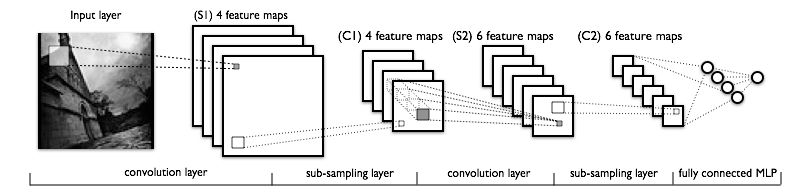
   最大池化在Theano中的theano.tensor.signal.downsample.max\_pool\_2d实现了。这个函数以一个N维的张量作为输入（N>2），和一个缩放因子用来对这个张量进行最大池化的变换。下面是例程代码：



注意到和大部分代码不同的是，这个函数max\_pool\_2d 在创建Theano图的时候，需要一个向下采样的因子ds (长度为2的tuple变量，表示了图像的宽和高的缩放. 这个可能在以后的版本中升级。

7、一个完整的CNN模型：LeNet

   稀疏性、卷积层和最大池化是LeNet系列模型的核心概念。犹豫模型细节变化较大，我们用下图来展示整个LeNet模型。



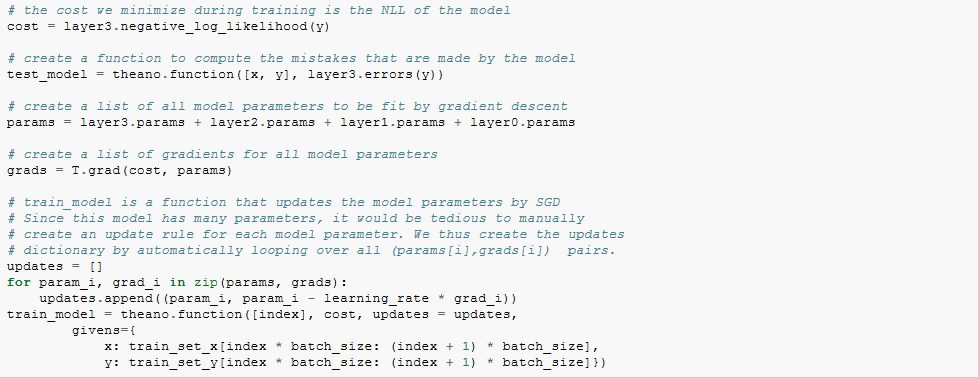
模型的低层由卷基层和最大池化曾组成，高层是一个全连接的MLP神经网络（隐层+逻辑回归，ANN），高层的输入是下层特征图的集合。

从实现的角度讲，这意味着低层操作了4D的张量，这个张量被压缩到了一个2D矩阵表示的光栅化的特征图上，以便于和前面的MLP的实现兼容。

8、全部代码







应该注意的是，在初始化权重的时候，fan-in是由感知野的大小和输入特征图的数目决定的。最后，采用前面章节定义的LogisticRegression和HiddenLayer类，LeNet就可以工作了。

9、注意要点和技巧

   超参数选择：由于CNNs比标准的MLP有着更多的超参数，所以CNNs的模型训练是很需要技巧的。不过传统的学习率和惩罚项仍然是需要使用的，下面说的的这些技巧在优化CNNs模型的过程中需要牢记。

  （1）滤波器的数量选择：在选定每一层的滤波器的数量的时候，要牢记计算一个卷积层滤波器的激活函数比计算传统的MLPs的激活函数的代价要高很多！假设第（i-1）层包含了Ki-1个特征图和M\*N个像素坐标（如坐标位置数目乘以特征图数目），在l层有Kl个m\*n的滤波器，所以计算特征图的代价为：（M-m）\*（N-n）\*m\*n\*Kl-1。整个代价是Kl乘级的。如果一层的所有特征图没有和前一层的所有的特征图全部连起来，情况可能会更加复杂一些。对于标准的MLP，这个代价为Kl \* Kl-1，Kl是第l层上的不同的节点。所以，CNNs中的特征图数目一般比MLPs中的隐层节点数目要少很多，这还取决于特征图的尺寸大小。因为特征图的尺寸随着层次深度的加大而变小，越靠近输入，所在层所包含的特征图越少，高层的特征图会越多。实际上，把每一次的计算平均一下，输出的特征图的的数目和像素位置的数目在各层是大致保持不变的。To preserve the information about the input would require keeping the total number of activations (number of feature maps times number of pixel positions) to be non-decreasing from one layer to the next (of course we could hope to get away with less when we are doing supervised learning).所以特征图的数量直接控制着模型的容量，它依赖于样本的数量和任务的复杂度。

  （2）滤波器的模型属性(shape)：一般来说，在论文中，由于所用的数据库不一样，滤波器的模型属性变化都会比较大。最好的CNNs的MNIST分类结果中，图像（28\*28）在第一层的输入用的5\*5的窗口（感受野），然后自然图像一般都使用更大的窗口，如12\*12,15\*15等。为了在给定数据库的情况下，获得某一个合适的尺度下的特征，需要找到一个合适的粒度等级。

  （3）最大池化的模型属性：典型的取值是2\*2或者不用最大池化。比较大的图像可以在CNNs的低层用4\*4的池化窗口。但是要需要注意的是，这样的池化在降维的同事也有可能导致信息丢失严重。

  （4）注意点：如果想在一些新的数据库上用CNN进行测试，可以对数据先进行白化处理（如用PCA），还有就是在每次训练迭代中减少学习率，这样可能会得到更好的实验效果。