# theano学习指南

**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" \o "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。我们也鼓励你把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);)

复制代码

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))

# 往GPU中保存的数据必须是floats格式的，因此上面的数据都被转变成floatX格式的（data\_x和  
 # data\_y都是同样的操作），y被处理后还要作为index，因此还要把y变为整数（见下面的操作）。

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

[复制代码](javascript:void(0);)

**符号**

**数据集符号**

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)∈RD为特征向量，y(i)∈(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，

* W 大写符号表示矩阵
* Wij 矩阵第i行，第j列的元素
* Wi. 行向量
* W.j 列向量
* b  向量
* bi  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
* Dih  第i层隐变量的个数
* fθ(x),f(x) 分类函数
* L  标注的个数
* L(θ,D) 模型似然函数的对数形式
* l(θ,D)  预测函数的经验损失
* NLL    负的以对数表示的似然函数
* θ  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:RD−>0,...,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={1 if x is True0 else

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);)

复制代码

# 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

[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);)

复制代码

# 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

[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);)

复制代码

# 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

[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);)

复制代码

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

[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);)

复制代码

# 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

[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，

为参数θ的Lp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);)

复制代码

# 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

[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);)

复制代码

# 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

复制代码

[复制代码](javascript:void(0);)