[Theano学习笔记（一）——代数](http://blog.csdn.net/ycheng_sjtu/article/details/38898697)

分类： [Python](http://blog.csdn.net/u012428391/article/category/2509039)2014-08-28 15:29 2008人阅读 [评论](http://blog.csdn.net/ycheng_sjtu/article/details/38898697#comments)(5) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/ycheng_sjtu/article/details/38898697#report)

[numpy](http://www.csdn.net/tag/numpy)[python](http://www.csdn.net/tag/python)[深度学习](http://www.csdn.net/tag/%e6%b7%b1%e5%ba%a6%e5%ad%a6%e4%b9%a0)[Theano](http://www.csdn.net/tag/Theano)[gpu](http://www.csdn.net/tag/gpu)

标量相加

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/38898697)[在CODE上查看代码片](https://code.csdn.net/snippets/459900)

1. **import** theano.tensor as T
2. **from** theano **import** function
3. x = T.dscalar('x')
4. y = T.dscalar('y')
5. z = x + y
6. f = function([x, y], z)

输入定义两个符号变量来代替数值，输出是一个0维的numpy.ndarray数组。

矩阵相加

把输入类型换一下就行了，矩阵如果维数不同，会遵循NumPy的广播规则。

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/38898697)[在CODE上查看代码片](https://code.csdn.net/snippets/459900)

1. **import** theano.tensor as T
2. **from** theano **import** function
3. x = T.dmatrix('x')
4. y = T.dmatrix('y')
5. z = x + y
6. f = function([x, y], z)

定义一个公式如：a \*\* 2 + b \*\* 2 + 2 \* a\* b

这里每个变量都需要单独申明。

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/38898697)[在CODE上查看代码片](https://code.csdn.net/snippets/459900)

1. **import** theano
2. a = theano.tensor.vector()
3. b = theano.tensor.vector()
4. out = a \*\* 2 + b \*\* 2 + 2 \* a \* b
5. f = theano.function([a,b],out)
6. **print** f([0, 1],[1,2])
7. >>>
8. [ 1. 9.]

支持多输出

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/38898697)[在CODE上查看代码片](https://code.csdn.net/snippets/459900)

1. **import** theano.tensor as T
2. **from** theano **import** function
3. a, b = T.dmatrices('a', 'b')
4. diff = a - b
5. abs\_diff = abs(diff)
6. diff\_squared = diff\*\*2
7. f = function([a, b], [diff, abs\_diff,diff\_squared])
8. **print** f([[1, 1], [1, 1]], [[0, 1], [2,3]])
9. >>>
10. [array([[ 1.,  0.],
11. [-1., -2.]]), array([[ 1.,  0.],
12. [ 1.,  2.]]), array([[ 1.,  0.],
13. [ 1.,  4.]])]

设置默认参数

和标准Python一样，缺省参数必须在非缺省之后，也可以定义缺省变量名。

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/38898697)[在CODE上查看代码片](https://code.csdn.net/snippets/459900)

1. **import** theano.tensor as T
2. **from** theano **import** function
3. **from** theano **import** Param
4. x, y = T.dscalars('x', 'y')
5. z = x + y
6. f = function([x, Param(y, default=1,name='by\_name')],z)
7. **print** f(33)
8. **print** f(33, 2)
9. **print** f(33,by\_name=3)
10. >>>
11. 34.0
12. 35.0
13. 36.0

共享变量

为了在GPU上更好的性能，引入共享变量，以累加器为例。

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/38898697)[在CODE上查看代码片](https://code.csdn.net/snippets/459900)

1. **import** theano.tensor as T
2. **from** theano **import** function
3. **from** theano **import** shared
4. state = shared(0)
5. inc = T.iscalar('inc')
6. accumulator = function([inc], state,updates=[(state, state+inc)])
7. **print** state.get\_value()
8. accumulator(1)
9. **print** state.get\_value()
10. accumulator(300)
11. **print** state.get\_value()
12. state.set\_value(-1)
13. **print** accumulator(3)
14. **print** state.get\_value()
15. >>>
16. 0
17. 1
18. 301
19. -1
20. 2

state的值在调用函数之后才刷新。而且可以定义多个函数共用同一个共享变量，例如这个减法器。

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/38898697)[在CODE上查看代码片](https://code.csdn.net/snippets/459900)

1. decrementor = function([inc], state,updates=[(state, state-inc)])
2. **print** decrementor(2)
3. **print** state.get\_value()
4. >>>
5. 2
6. 0

如果在某个函数中，共用了这个共享变量，但是又不想变动它的值，那么可以使用given参数替代这个变量。而旧的state不发生变化。

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/38898697)[在CODE上查看代码片](https://code.csdn.net/snippets/459900)

1. fn\_of\_state = state \* 2 + inc
2. foo = T.scalar(dtype=state.dtype)
3. skip\_shared = function([inc, foo],fn\_of\_state,
4. givens=[(state,foo)])
5. **print** skip\_shared(1, 3)
6. **print** state.get\_value()
7. >>>
8. 7
9. 0

产生随机数

和C中的srand()一样，都是伪随机数。

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/38898697)[在CODE上查看代码片](https://code.csdn.net/snippets/459900)

1. **from** theano **import** function
2. **from** theano.tensor.shared\_randomstreamsimport RandomStreams
3. srng = RandomStreams(seed=234)#种子
4. rv\_u = srng.uniform((2,2))#均匀分布
5. rv\_n = srng.normal((2,2))#正态分布
6. f = function([], rv\_u)#每次调用，每次都会更新
7. g = function([], rv\_n,no\_default\_updates=True)#如果以后一直用这组随机数，就不再更新
8. nearly\_zeros = function([], rv\_u + rv\_u- 2 \* rv\_u)
9. **print** nearly\_zeros()#函数每次执行只获得一个随机数，即使表达式里面有3个随机数

种子流：上述2个随机变量，可以全局设定同一个种子，也可以是分别设定。

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/38898697)[在CODE上查看代码片](https://code.csdn.net/snippets/459900)

1. #分别设置，使用.rng.set\_value()函数
2. rng\_val =rv\_u.rng.get\_value(borrow=True) # Get the rng for rv\_u
3. rng\_val.seed(89234) # seeds thegenerator
4. rv\_u.rng.set\_value(rng\_val,borrow=True)
5. #全局设置，使用.seed()函数
6. srng.seed(902340)

函数间共享流

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/38898697)[在CODE上查看代码片](https://code.csdn.net/snippets/459900)

1. state\_after\_v0 =rv\_u.rng.get\_value().get\_state()#保存调用前的state
2. nearly\_zeros()       # this affects rv\_u's generator
3. v1 = f()#第一个调用，之后state会变化
4. rng = rv\_u.rng.get\_value(borrow=True)
5. rng.set\_state(state\_after\_v0)#为其state还原
6. rv\_u.rng.set\_value(rng, borrow=True)
7. v2 = f()             # v2 != v1输出更新后state对应的随机数
8. v3 = f()             # v3 == v1再次更新又还原成原来的state了

在2张Theano图间复制状态

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/38898697)[在CODE上查看代码片](https://code.csdn.net/snippets/459900)

1. **import** theano
2. **import** numpy
3. **import** theano.tensor as T
4. **from** theano.sandbox.rng\_mrg importMRG\_RandomStreams
5. **from** theano.tensor.shared\_randomstreamsimport RandomStreams
7. **class** Graph():
8. **def** \_\_init\_\_(self, seed=123):
9. self.rng = RandomStreams(seed)
10. self.y = self.rng.uniform(size=(1,))
12. g1 = Graph(seed=123)
13. f1 = theano.function([], g1.y)
15. g2 = Graph(seed=987)
16. f2 = theano.function([], g2.y)
18. **print** 'By default, the two functionsare out of sync.'
19. **print** 'f1() returns ', f1()
20. **print** 'f2() returns ', f2()
21. #输出不同的随机值
22. **def** copy\_random\_state(g1, g2):
23. **if** isinstance(g1.rng, MRG\_RandomStreams):
24. #类型判断：其第一个参数为对象，第二个为类型名或类型名的一个列表。其返回值为布尔型。
25. g2.rng.rstate = g1.rng.rstate
26. **for** (su1, su2) **in** zip(g1.rng.state\_updates, g2.rng.state\_updates):#打包
27. su2[0].set\_value(su1[0].get\_value())#赋值
29. **print** 'We now copy the state of thetheano random number generators.'
30. copy\_random\_state(g1, g2)
31. **print** 'f1() returns ', f1()
32. **print** 'f2() returns ', f2()
33. #输出相同的随机值
34. >>>
35. By default, the two functions are outof sync.
36. f1() returns  [ 0.72803009]
37. f2() returns  [ 0.55056769]
38. We now copy the state of the theanorandom number generators.
39. f1() returns  [ 0.59044123]
40. f2() returns  [ 0.59044123]

[Theano学习笔记（二）——逻辑回归函数解析](http://blog.csdn.net/ycheng_sjtu/article/details/38904849)

分类： [Python](http://blog.csdn.net/u012428391/article/category/2509039)2014-08-28 21:46 1215人阅读 [评论](http://blog.csdn.net/ycheng_sjtu/article/details/38904849#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/ycheng_sjtu/article/details/38904849#report)

[python](http://www.csdn.net/tag/python)[numpy](http://www.csdn.net/tag/numpy)[Deep Learning](http://www.csdn.net/tag/Deep%20Learning)[Theano](http://www.csdn.net/tag/Theano)[深度学习](http://www.csdn.net/tag/%e6%b7%b1%e5%ba%a6%e5%ad%a6%e4%b9%a0)

有了前面的准备，可以用Theano实现一个逻辑回归程序，逻辑回归是典型的有监督学习。

为了形象，这里我们假设分类任务是区分人与狗的照片。

首先是生成随机数对象

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/38904849)

1. importnumpy
2. importtheano
3. importtheano.tensor as T
4. rng= numpy.random

数据初始化

有400张照片，这些照片不是人的就是狗的。

每张照片是28\*28=784的维度。

D[0]是训练集，是个400\*784的矩阵，每一行都是一张照片。

D[1]是每张照片对应的标签，用来记录这张照片是人还是狗。

training\_steps是迭代上限。

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/38904849)

1. N= 400
2. feats= 784
3. D= (rng.randn(N, feats), rng.randint(size=N, low=0, high=2))
4. training\_steps= 10000

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/38904849)

1. #Declare Theano symbolic variables
2. x= T.matrix("x")
3. y= T.vector("y")
4. w= theano.shared(rng.randn(feats), name="w")
5. b= theano.shared(0., name="b")
6. **print**"Initial model:"
7. printw.get\_value(), b.get\_value()

x是输入的训练集，是个矩阵，把D[0]赋值给它。

y是标签，是个列向量，400个样本所以有400维。把D[1]赋给它。

w是权重列向量，维数为图像的尺寸784维。

b是偏倚项向量，初始值都是0，这里没写成向量是因为之后要广播形式。

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/38904849)

1. #Construct Theano expression graph
2. p\_1= 1 / (1 + T.exp(-T.dot(x, w) - b))   #Probability that target = 1
3. prediction= p\_1 > 0.5                    # Theprediction thresholded
4. xent= -y \* T.log(p\_1) - (1-y) \* T.log(1-p\_1) # Cross-entropy loss function
5. cost= xent.mean() + 0.01 \* (w \*\* 2).sum()# The cost to minimize
6. gw,gb = T.grad(cost, [w, b])             #Compute the gradient of the cost
7. # (we shall return to this in a
8. #following section of this tutorial)

这里是函数的主干部分，涉及到3个公式

1.判定函数

http://latex.codecogs.com/gif.latex?%7bh_\theta%20%7d\left(%20x%20\right)%20=%20\frac%7b1%7d%7b%7bI%20+%20%7be%5e%7b%20-%20%7b\theta%20%5eT%7dX%7d%7d%7d%7d

2.代价函数

3.总目标函数

第二项是权重衰减项，减小权重的幅度，用来防止过拟合的。

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/38904849)

1. #Compile
2. train= theano.function(
3. inputs=[x,y],
4. outputs=[prediction, xent],
5. updates=((w, w - 0.1 \* gw), (b, b -0.1 \* gb)))
6. predict= theano.function(inputs=[x], outputs=prediction)

构造预测和训练函数。

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/38904849)

1. #Train
2. fori **in** range(training\_steps):
3. pred,err = train(D[0], D[1])
4. **print**"Final model:"
5. printw.get\_value(), b.get\_value()
6. **print**"target values for D:", D[1]
7. **print**"prediction on D:", predict(D[0])

这里算过之后发现，经过10000次训练，预测结果与标签已经完全相同了。

[Theano学习笔记（三）——图结构](http://blog.csdn.net/ycheng_sjtu/article/details/38920473)

分类： [Python](http://blog.csdn.net/u012428391/article/category/2509039)2014-08-29 10:13 1515人阅读 [评论](http://blog.csdn.net/ycheng_sjtu/article/details/38920473#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/ycheng_sjtu/article/details/38920473#report)

[Theano](http://www.csdn.net/tag/Theano)[python](http://www.csdn.net/tag/python)[深度学习](http://www.csdn.net/tag/%e6%b7%b1%e5%ba%a6%e5%ad%a6%e4%b9%a0)[Deep Learning](http://www.csdn.net/tag/Deep%20Learning)[图结构](http://www.csdn.net/tag/%e5%9b%be%e7%bb%93%e6%9e%84)

图结构（Graph Structures）是了解Theano内在工作原理的基础。

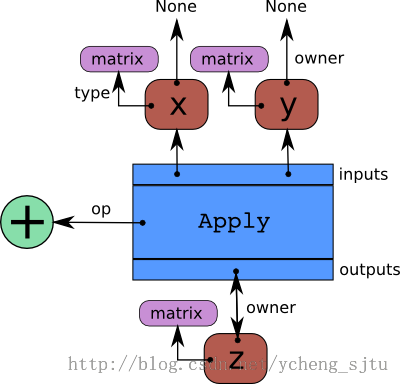
Theano编程的核心是用符号占位符把数学关系表示出来。

图结构的组成部分

如图实现了这段代码：

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/38920473)

1. importtheano.tensor as T
2. x= T.matrix('x')
3. y= T.matrix('y')
4. z= x + y



变量节点（variable nodes）

红色表示。变量节点都有owner，其中x与y的owner为none。z的owner为apply。

操作节点（op nodes）

绿色表示。表示各个变量之间的运算（例如+, -, \*\*, sum(),tanh()等等）。

应用节点（apply nodes）

蓝色表示。其他节点都连在上面。

分析nodes对应属性

对于以下代码，分析其节点属性。

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/38920473)

1. importtheano.tensor as T
2. x= T.dmatrix('x')
3. y= x \* 2.
4. >>>y.owner.op.name
5. 'Elemwise{mul,no\_inplace}'#y的owner是apply而apply的op是'Elemwise{mul,no\_inplace}'
6. >>>len(y.owner.inputs)
7. 2#两个输入
8. >>>y.owner.inputs[0]
9. x#第一个输入是x矩阵
10. >>>y.owner.inputs[1]
11. InplaceDimShuffle{x,x}.0

#注意这里第二个输入并不是2，而是和x同样大小的矩阵框架，因为等会要广播才能相乘

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/38920473)

1. >>>type(y.owner.inputs[1])
2. <**class**'theano.tensor.basic.TensorVariable'>
3. >>>type(y.owner.inputs[1].owner)
4. <**class**'theano.gof.graph.Apply'>
5. >>>y.owner.inputs[1].owner.op
6. <**class**'theano.tensor.elemwise.DimShuffle object at 0x14675f0'>#用DimShuffle把2广播出来
7. >>>y.owner.inputs[1].owner.inputs
8. [2.0]#矩阵框架的owner才是2

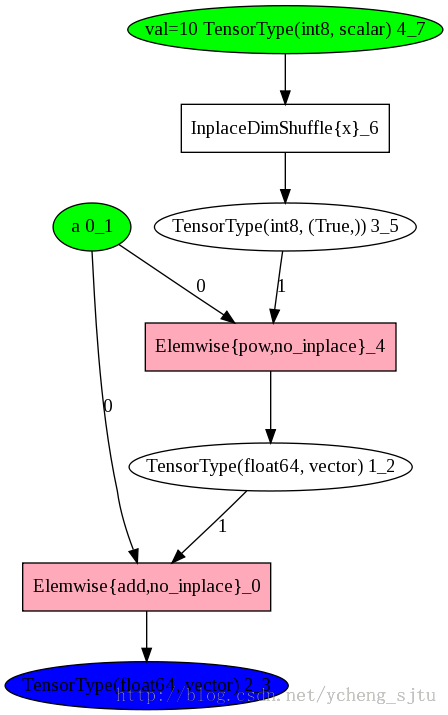
自动优化

编译Theano其实是编译了一张图。这张图从输入变量开始贯穿全图直到输出变量。Theano可以检测关键子图，来进行替换，防止重复，以达到优化的目的。比如用x替换xy/y。

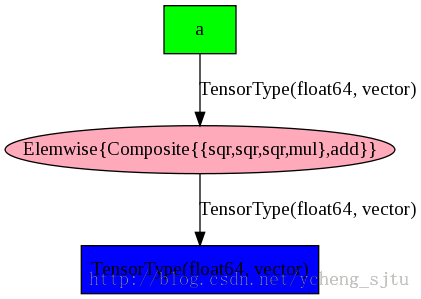
举个例子

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/38920473)

1. >>>**import** theano
2. >>>a = theano.tensor.vector("a")     # declare symbolic variable
3. >>>b = a + a \*\* 10                    #build symbolic expression
4. >>>f = theano.function([a], b)        #compile function
5. >>>**print** f([0, 1, 2])                 #prints `array([0,2,1026])`



优化前



优化后

[Theano学习笔记（四）——导数](http://blog.csdn.net/ycheng_sjtu/article/details/38929057)

分类： [Python](http://blog.csdn.net/u012428391/article/category/2509039)2014-08-29 18:34 1348人阅读 [评论](http://blog.csdn.net/ycheng_sjtu/article/details/38929057#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/ycheng_sjtu/article/details/38929057#report)

[深度学习](http://www.csdn.net/tag/%e6%b7%b1%e5%ba%a6%e5%ad%a6%e4%b9%a0)[python](http://www.csdn.net/tag/python)[Theano](http://www.csdn.net/tag/Theano)[导数](http://www.csdn.net/tag/%e5%af%bc%e6%95%b0)[Deep Learning](http://www.csdn.net/tag/Deep%20Learning)

导数使用T.grad计算。

这里使用pp()打印梯度的符号表达式。

第3行输出是打印了经过优化器简化的符号梯度表达式，与第1个输出相比确实简单多了。

fill((x\*\* TensorConstant{2}), TensorConstant{1.0})指创建一个x\*\*2大小的矩阵，并填充1。

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/38929057)

1. importtheano.tensor as T
2. fromtheano **import** pp
3. fromtheano **import** function
4. x= T.dscalar('x')
5. y= x \*\* 2
6. gy= T.grad(y, x)
7. printpp(gy)
8. f= function([x], gy)
9. printf(4)
10. printpp(f.maker.fgraph.outputs[0])
11. >>>
12. ((fill((x\*\* TensorConstant{2}), TensorConstant{1.0}) \* TensorConstant{2}) \* (x \*\*(TensorConstant{2} - TensorConstant{1})))
13. 8.0
14. (TensorConstant{2.0}\* x)

T.grad的第1个参数必须是标量

例如计算逻辑函数sigmoid的导数：

http://latex.codecogs.com/gif.latex?\frac%7b%7bds\left(%20x%20\right)%7d%7d%7b%7bdx%7d%7d%20=%20s\left(%20x%20\right)%20\cdot%20\left(%20%7b1%20-%20s\left(%20x%20\right)%7d%20\right)

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/38929057)

1. importtheano.tensor as T
2. fromtheano **import** function
3. x= T.dmatrix('x')
4. s= T.sum(1 / (1 + T.exp(-x)))
5. gs= T.grad(s, x)
6. dlogistic= function([x], gs)
7. printdlogistic([[0, 1], [-1, -2]])
8. >>>
9. [[0.25        0.19661193]
10. [ 0.19661193 0.10499359]]

计算雅克比（Jacobian）矩阵

雅克比矩阵是向量的一阶偏导数：

用T.arrange生成从0到y.shape[0]的序列。循环计算。

scan可以提高创建符号循环效率。

lambda~是python内建的magicfunction.

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/38929057)

1. x= T.dvector('x')
2. y = x \*\* 2
3. J, updates = theano.scan(lambdai, y,x : T.grad(y[i], x), sequences=T.arange(y.shape[0]), non\_sequences=[y,x])
4. f = function([x], J,updates=updates)
5. f([4, 4])
6. >>>
7. [[ 8.  0.]
8. [ 0. 8.]]

计算海森（Hessian）矩阵

海森矩阵是多元函数的二阶偏导数方阵。

只要用T.grad(cost,x)替换雅克比矩阵的一些y即可。

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/38929057)

1. x= T.dvector('x')
2. y = x\*\* 2
3. cost= y.sum()
4. gy =T.grad(cost, x)
5. H,updates = theano.scan(**lambda** i, gy,x : T.grad(gy[i], x),sequences=T.arange(gy.shape[0]), non\_sequences=[gy, x])
6. f =function([x], H, updates=updates)
7. f([4,4])
8. >>>
9. [[2.  0.]
10. [ 0. 2.]]

雅克比右乘

x可以由向量扩展成矩阵。雅克比右乘使用Rop:

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/38929057)

1. W = T.dmatrix('W')
2. V =T.dmatrix('V')
3. x =T.dvector('x')
4. y =T.dot(x, W)
5. JV =T.Rop(y, W, V)
6. f =theano.function([W, V, x], JV)
7. printf([[1, 1], [1, 1]], [[2, 2], [2, 2]], [0,1])
8. >>>
9. [2.  2.]

雅克比左乘

雅克比左乘使用Lop:

http://latex.codecogs.com/gif.latex?\frac%7b%7b\partial%20f\left(%20x%20\right)%7d%7d%7b%7b\partial%20x%7d%7dv

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/38929057)

1. **import** theano
2. **import** theano.tensor as T
3. **from** theano **import** function
4. x = T.dvector('x')
5. v =T.dvector('v')
6. x =T.dvector('x')
7. y =T.dot(x, W)
8. VJ =T.Lop(y, W, v)
9. f =theano.function([v,x], VJ)
10. **print** f([2, 2], [0, 1])
11. >>>
12. [[0.  0.]
13. [ 2. 2.]]

海森矩阵乘以向量

可以使用Rop

http://latex.codecogs.com/gif.latex?v\frac%7b%7b\partial%20f\left(%20x%20\right)%7d%7d%7b%7b\partial%20x%7d%7d

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/38929057)

1. **import** theano
2. **import** theano.tensor as T
3. **from** theano **import** function
4. x= T.dvector('x')
5. v= T.dvector('v')
6. y= T.sum(x \*\* 2)
7. gy= T.grad(y, x)
8. Hv= T.Rop(gy, x, v)
9. f= theano.function([x, v], Hv)
10. printf([4, 4], [2, 2])
11. >>>
12. [4.  4.]

[Theano学习笔记（五）——配置设置与编译模型](http://blog.csdn.net/ycheng_sjtu/article/details/39010413)

分类： [Python](http://blog.csdn.net/u012428391/article/category/2509039)2014-09-02 19:33 2494人阅读 [评论](http://blog.csdn.net/ycheng_sjtu/article/details/39010413#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/ycheng_sjtu/article/details/39010413#report)

[Theano](http://www.csdn.net/tag/Theano)[深度学习](http://www.csdn.net/tag/%e6%b7%b1%e5%ba%a6%e5%ad%a6%e4%b9%a0)[python](http://www.csdn.net/tag/python)

配置

config模块包含了各种用于修改Theano的属性。在Theano导入时，许多属性都会被检查，而有些属性是只读模式。

一般约定，在用户代码内部config模块的属性不应当被修改。

Theano的这些属性都有默认值，但是你也可以在你的.theanorc文件里面修改，并且使用THEANO\_FLAGS的环境变量进行修改。

优先顺序是：

1. theano.config.<property>的赋值

2. THEANO\_FLAGS的赋值

3..theanorc（或者在THEANORC文件中表示）的赋值

通过打印theano.config可以展示当前的配置：

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/39010413)

1. python-c 'import theano; print theano.config' | less

例如，修改[笔记（二）](http://blog.csdn.net/ycheng_sjtu/article/details/38904849" \t "_blank)中的逻辑回归函数，设置精度为float32

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/39010413)

1. #!/usr/bin/envpython
2. #Theano tutorial
3. #Solution to Exercise in section 'Configuration Settings and Compiling Modes'
5. importnumpy
6. importtheano
7. importtheano.tensor as tt
9. theano.config.floatX= 'float32'
11. rng= numpy.random
13. N= 400
14. feats= 784
15. D= (rng.randn(N, feats).astype(theano.config.floatX),
16. rng.randint(size=N,low=0, high=2).astype(theano.config.floatX))
17. training\_steps= 10000
19. #Declare Theano symbolic variables
20. x= tt.matrix("x")
21. y= tt.vector("y")
22. w= theano.shared(rng.randn(feats).astype(theano.config.floatX),name="w")
23. b= theano.shared(numpy.asarray(0., dtype=theano.config.floatX),name="b")
24. x.tag.test\_value= D[0]
25. y.tag.test\_value= D[1]
26. #print"Initial model:"
27. #printw.get\_value(), b.get\_value()
29. #Construct Theano expression graph
30. p\_1= 1 / (1 + tt.exp(-tt.dot(x, w) - b))  #Probability of having a one
31. prediction= p\_1 > 0.5  # The prediction that isdone: 0 or 1
32. xent= -y \* tt.log(p\_1) - (1 - y) \* tt.log(1 - p\_1) # Cross-entropy
33. cost= tt.cast(xent.mean(), 'float32') + \
34. 0.01 \* (w \*\* 2).sum()  # The cost to optimize
35. gw,gb = tt.grad(cost, [w, b])
37. #Compile expressions to functions
38. train= theano.function(
39. inputs=[x, y],
40. outputs=[prediction, xent],
41. updates={w: w - 0.01 \* gw, b: b -0.01 \* gb},
42. name="train")
43. predict= theano.function(inputs=[x], outputs=prediction,
44. name="predict")
46. ifany([x.op.\_\_class\_\_.\_\_name\_\_ **in** ['Gemv', 'CGemv', 'Gemm', 'CGemm'] **for** x **in**
47. train.maker.fgraph.toposort()]):
48. **print** 'Used the cpu'
49. elifany([x.op.\_\_class\_\_.\_\_name\_\_ **in** ['GpuGemm', 'GpuGemv'] **for** x **in**
50. train.maker.fgraph.toposort()]):
51. **print** 'Used the gpu'
52. **else**:
53. **print** 'ERROR, not able to tell if theanoused the cpu or the gpu'
54. **print** train.maker.fgraph.toposort()
56. fori **in** range(training\_steps):
57. pred, err = train(D[0], D[1])
58. #print"Final model:"
59. #printw.get\_value(), b.get\_value()
61. **print**"target values for D"
62. printD[1]
64. **print**"prediction on D"
65. printpredict(D[0])

用time python file.py运行，可得：

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/39010413)

1. real  0m15.055s
2. user 0m11.527s
3. sys   0m0.801s

Mode

每次调用theano.function时，Theano变量输入和输出的符号化关系都被优化和编译了。

而这些编辑都通过made参数的值来控制。

Theano定义以下mode：

FAST\_COMPILE：

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/39010413)

1. compile.mode.Mode(linker='py',optimizer='fast\_compile')

只应用少量的图优化并且只使用Python实现。

FAST\_RUN：

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/39010413)

1. compile.mode.Mode(linker='cvm',optimizer='fast\_run')

使用所有的优化并且在可能的情况下使用C实现。

DebugMode：

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/39010413)

1. compile.debugmode.DebugMode()

检查所有优化的正确性，并且比较C与Python实现。这种模式比别的模式耗时都长，但是可以识别出各种问题。

ProfileMode（不赞成使用）：

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/39010413)

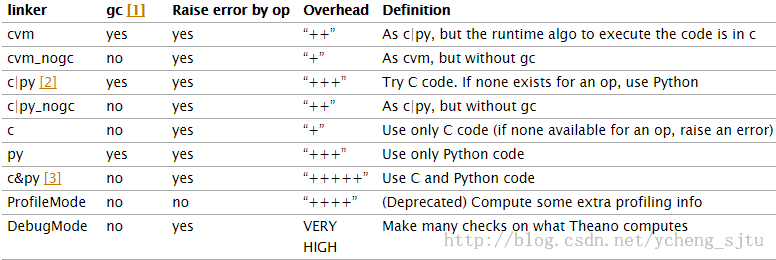
1. compile.profilemode.ProfileMode()

与FAST\_RUN相同的优化，但是打印出一些设置信息。

默认的模式是FAST\_RUN，但是通过传递关键字参数给theano.function，可以控制config.mode，从而改变模式。

Linkers

一个mode由2个部分组成：1个优化器和1个Linker。



[1]   gc指计算中间过程的碎片收集。否则在Theano函数调用之间，操作所使用的内存空间将被保存起来。为了不重新分配内存，降低开销（overhead），使其速度更快。

[2]   默认linker

[3]   不推荐使用

使用DebugMode

一般你应当使用FAST\_RUN 或者FAST\_COMPILE模式，当你定义新的类型的表达式或者优化方法时，先用DebugMode（mode='DebugMode'）运行是很有用的，DebugMode通过运行一些自检和判断程序来帮助诊断出将会导致错误输出的可能的编程错误。值得注意的是，DebugMode比FAST\_RUN或者 FAST\_COMPILE模式要慢得多，所以只在开发期使用。

举个例子：

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/39010413)

1. **import** theano
2. importtheano.tensor as T
3. x= T.dvector('x')
4. f= theano.function([x], 10 \* x, mode='DebugMode')
5. f([5])
6. f([0])
7. f([7])

运行后，如果有问题，输出会提示异常，如果依然不能解决，请联系本领域的专家。

但是DebugMode也不是万能的，因为有些错误只在特定的输入条件下才会出现。

如果你使用构造器而不是关键词DebugMode，就可以通过配构造器变量来配置。而关键词设置太严格了。

ProfileMode不推荐使用

检索时间信息

图编译好之后，运行就可以了。然后调用profmode.print\_summary()，返回各自时间信息，例如你的图大多数时间花在什么地方了等等。

还是以逻辑回归为例

生成ProfileMode实例

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/39010413)

1. fromtheano **import** ProfileMode
2. profmode= theano.ProfileMode(optimizer='fast\_run', linker=theano.gof.OpWiseCLinker())

在函数末尾声明一下

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/39010413)

1. train = theano.function(
2. inputs=[x,y],
3. outputs=[prediction,xent],
4. updates={w:w - 0.01 \* gw, b: b - 0.01 \* gb},
5. name="train",mode=profmode)
6. #如果是Module则这样声明：
7. # m = theano.Module()
8. # minst = m.make(mode=profmode)

取回时间信息

文件末尾添加

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/39010413)

1. profmode.print\_summary()

则运行效果是这样的

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/39010413)

1. ProfileMode.print\_summary()
2. ---------------------------
4. Timesince **import** 6.183s
5. Theanocompile time: 0.000s (0.0% since **import**)
6. Optimization time: 0.000s
7. Linker time: 0.000s
8. Theanofct call 5.452s (88.2% since **import**)
9. Theano Op time 5.003s 80.9%(since **import**)91.8%(of fct call)
10. Theano function overhead **in** ProfileMode0.449s 7.3%(since **import**) 8.2%(of fct call)
11. 10000Theano fct call, 0.001s per call
12. Restof the time since **import** 0.730s 11.8%
14. Theanofct summary:
15. <%total fct time> <total time> <time per call> <nb call><fct name>
16. 100.0% 5.452s 5.45e-04s 10000 train
18. SingleOp-wise summary:
19. <%of local\_time spent on this kind of Op> <cumulative %> <selfseconds> <cumulative seconds> <time per call> [\*]<nb\_call> <nb\_op> <nb\_apply> <Op name>
20. 87.9%  87.9%  4.400s  4.400s 2.20e-04s \* 20000  1  2 <**class** 'theano.tensor.blas\_c.CGemv'>
21. 10.8%  98.8%  0.542s  4.942s 5.42e-06s \* 100000 10 10 <**class** 'theano.tensor.elemwise.Elemwise'>
22. 0.5%  99.3%  0.023s  4.966s 1.17e-06s \* 20000  1  2 <**class** 'theano.tensor.basic.Alloc'>
23. 0.4%  99.6%  0.018s  4.984s 6.05e-07s \* 30000  2  3 <**class**'theano.tensor.elemwise.DimShuffle'>
24. 0.3%  99.9%  0.013s  4.997s 1.25e-06s \* 10000  1  1 <**class** 'theano.tensor.elemwise.Sum'>
25. 0.1% 100.0%  0.007s  5.003s 3.35e-07s \* 20000  1  2 <**class** 'theano.compile.ops.Shape\_i'>
26. ... (remaining 0 single Op account for0.00%(0.00s) of the runtime)
27. (\*)Op **is** running a c implementation
29. Op-wisesummary:
30. <%of local\_time spent on this kind of Op> <cumulative %> <selfseconds> <cumulative seconds> <time per call> [\*]  <nb\_call> <nb apply> <Opname>
31. 87.9%  87.9%  4.400s  4.400s 2.20e-04s \* 20000  2CGemv{inplace}
32. 6.3%  94.3%  0.318s  4.718s 3.18e-05s \* 10000  1Elemwise{Composite{[Composite{[Composite{[sub(mul(i0, i1), neg(i2))]}(i0,scalar\_softplus(i1), mul(i2, i3))]}(i0, i1, i2, scalar\_softplus(i3))]}}
33. 2.1%  96.3%  0.103s  4.820s 1.03e-05s \* 10000  1Elemwise{Composite{[Composite{[Composite{[Composite{[mul(i0, add(i1, i2))]}(i0,neg(i1), true\_div(i2, i3))]}(i0, mul(i1, i2, i3), i4, i5)]}(i0, i1, i2,exp(i3), i4, i5)]}}[(0, 0)]
34. 1.6%  98.0%  0.082s  4.902s 8.16e-06s \* 10000  1Elemwise{ScalarSigmoid{output\_types\_preference=transfer\_type{0}}}[(0, 0)]
35. 0.5%  98.4%  0.023s  4.925s 1.17e-06s \* 20000  2 Alloc
36. 0.3%  98.7%  0.013s  4.938s 1.25e-06s \* 10000  1 Sum
37. 0.2%  98.9%  0.012s  4.950s 6.11e-07s \* 20000  2InplaceDimShuffle{x}
38. 0.2%  99.1%  0.008s  4.959s 8.44e-07s \* 10000  1Elemwise{gt,no\_inplace}
39. 0.1%  99.2%  0.007s  4.965s 6.80e-07s \* 10000  1Elemwise{sub,no\_inplace}
40. 0.1%  99.4%  0.007s  4.972s 3.35e-07s \* 20000  2 Shape\_i{0}
41. 0.1%  99.5%  0.006s  4.978s 6.11e-07s \* 10000  1Elemwise{Composite{[sub(neg(i0), i1)]}}[(0, 0)]
42. 0.1%  99.6%  0.006s  4.984s 5.93e-07s \* 10000  1InplaceDimShuffle{1,0}
43. 0.1%  99.7%  0.005s  4.989s 5.33e-07s \* 10000  1Elemwise{neg,no\_inplace}
44. 0.1%  99.8%  0.005s  4.994s 4.85e-07s \* 10000  1Elemwise{Cast{float32}}
45. 0.1%  99.9%  0.005s  4.999s 4.60e-07s \* 10000  1Elemwise{inv,no\_inplace}
46. 0.1% 100.0%  0.004s  5.003s 4.25e-07s \* 10000  1Elemwise{Composite{[sub(i0, mul(i1, i2))]}}[(0, 0)]
47. ... (remaining 0 Op account **for**   0.00%(0.00s) of the runtime)
48. (\*)Op **is** running a c implementation
50. Apply-wisesummary:
51. <%of local\_time spent at this position> <cumulative %%> <applytime> <cumulative seconds> <time per call> [\*] <nb\_call><Apply position> <Apply Op name>
52. 54.7%  54.7%  2.737s  2.737s 2.74e-04s  \* 10000  7 CGemv{inplace}(Alloc.0, TensorConstant{1.0}, x, w,TensorConstant{0.0})
53. 33.2%  87.9%  1.663s  4.400s 1.66e-04s  \* 10000 18 CGemv{inplace}(w, TensorConstant{-0.00999999977648}, x.T,Elemwise{Composite{[Composite{[Composite{[Composite{[mul(i0, add(i1, i2))]}(i0,neg(i1), true\_div(i2, i3))]}(i0, mul(i1, i2, i3), i4, i5)]}(i0, i1, i2,exp(i3), i4, i5)]}}[(0, 0)].0, TensorConstant{0.999800026417})
54. 6.3%  94.3%  0.318s  4.718s 3.18e-05s  \* 10000 13 Elemwise{Composite{[Composite{[Composite{[sub(mul(i0, i1),neg(i2))]}(i0, scalar\_softplus(i1), mul(i2, i3))]}(i0, i1, i2,scalar\_softplus(i3))]}}(y, Elemwise{Composite{[sub(neg(i0), i1)]}}[(0, 0)].0,Elemwise{sub,no\_inplace}.0, Elemwise{neg,no\_inplace}.0)
55. 2.1%  96.3%  0.103s  4.820s 1.03e-05s  \* 10000 16 Elemwise{Composite{[Composite{[Composite{[Composite{[mul(i0, add(i1,i2))]}(i0, neg(i1), true\_div(i2, i3))]}(i0, mul(i1, i2, i3), i4, i5)]}(i0, i1,i2, exp(i3), i4, i5)]}}[(0,0)](Elemwise{ScalarSigmoid{output\_types\_preference=transfer\_type{0}}}[(0,0)].0, Alloc.0, y, Elemwise{Composite{[sub(neg(i0), i1)]}}[(0, 0)].0,Elemwise{sub,no\_inplace}.0, Elemwise{Cast{float32}}.0)
56. 1.6%  98.0%  0.082s  4.902s 8.16e-06s  \* 10000 14 Elemwise{ScalarSigmoid{output\_types\_preference=transfer\_type{0}}}[(0,0)](Elemwise{neg,no\_inplace}.0)
57. 0.3%  98.3%  0.015s  4.917s 1.53e-06s  \* 10000 12 Alloc(Elemwise{inv,no\_inplace}.0, Shape\_i{0}.0)
58. 0.3%  98.5%  0.013s  4.930s 1.25e-06s  \* 10000 17 Sum(Elemwise{Composite{[Composite{[Composite{[Composite{[mul(i0,add(i1, i2))]}(i0, neg(i1), true\_div(i2, i3))]}(i0, mul(i1, i2, i3), i4,i5)]}(i0, i1, i2, exp(i3), i4, i5)]}}[(0, 0)].0)
59. 0.2%  98.7%  0.008s  4.938s 8.44e-07s  \* 10000 15Elemwise{gt,no\_inplace}(Elemwise{ScalarSigmoid{output\_types\_preference=transfer\_type{0}}}[(0,0)].0, TensorConstant{(1,) of 0.5})
60. 0.2%  98.9%  0.008s  4.946s 8.14e-07s  \* 10000  5 Alloc(TensorConstant{0.0}, Shape\_i{0}.0)
61. 0.1%  99.0%  0.007s  4.953s 6.80e-07s  \* 10000  4 Elemwise{sub,no\_inplace}(TensorConstant{(1,) of 1.0}, y)
62. 0.1%  99.1%  0.006s  4.959s 6.16e-07s  \* 10000  6 InplaceDimShuffle{x}(Shape\_i{0}.0)
63. 0.1%  99.2%  0.006s  4.965s 6.11e-07s  \* 10000  9 Elemwise{Composite{[sub(neg(i0), i1)]}}[(0, 0)](CGemv{inplace}.0,InplaceDimShuffle{x}.0)
64. 0.1%  99.4%  0.006s  4.972s 6.07e-07s  \* 10000  0 InplaceDimShuffle{x}(b)
65. 0.1%  99.5%  0.006s  4.977s 5.93e-07s  \* 10000  2 InplaceDimShuffle{1,0}(x)
66. 0.1%  99.6%  0.005s  4.983s 5.33e-07s  \* 10000 11 Elemwise{neg,no\_inplace}(Elemwise{Composite{[sub(neg(i0), i1)]}}[(0,0)].0)
67. ... (remaining 5 Apply instances account for0.41%(0.02s) of the runtime)
68. (\*)Op **is** running a c implementation

[Theano学习笔记（六）——载入与保存、条件](http://blog.csdn.net/ycheng_sjtu/article/details/39060813)

分类： [Python](http://blog.csdn.net/u012428391/article/category/2509039)2014-09-04 22:26 1038人阅读 [评论](http://blog.csdn.net/ycheng_sjtu/article/details/39060813#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/ycheng_sjtu/article/details/39060813#report)

[Deep Learning](http://www.csdn.net/tag/Deep%20Learning)[Theano](http://www.csdn.net/tag/Theano)[python](http://www.csdn.net/tag/python)[深度学习](http://www.csdn.net/tag/%e6%b7%b1%e5%ba%a6%e5%ad%a6%e4%b9%a0)

载入与保存

Python标准的保存类别实体并重新载入它们的途径是pickle机制。许多Theano对象可以由此被序列化（或者反序列化），然而pickle的局限性在于，被序列化的类别实例的代码或者数据并没有被同时保存。因此重新载入先前版本的类可能会出问题。

因此，需要寻求基于预期保存和重新载入的耗时的不同机制。

对于短期（比如临时文件和网络转录），Theano的pickle是可行的。

对于长期（比如从实验中保存模型）不应当依赖于Theano的pickle对象。

推荐在任何其他Python项目的过程中的保存和载入底层共享对象。

Pickle基础

pickle和cPickle模块功能相似，但是cPickle用C编码，要更快一些。

可以用cPickle.dump把对象序列化（或者保存或者pickle）为一个文件。

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/39060813)[在CODE上查看代码片](https://code.csdn.net/snippets/464293)

1. importcPickle
2. f= file('obj.save', 'wb')
3. cPickle.dump(my\_obj,f, protocol=cPickle.HIGHEST\_PROTOCOL)
4. f.close()

使用了cPickle.HIGHEST\_PROTOCOL，使得保存对象的过程大大加快。

使用了’b’二进制模式，是为了在Unix和Windows系统之间保持可移植性。

使用cPickle.load把文件反序列化（或载入，或unpickle）

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/39060813)[在CODE上查看代码片](https://code.csdn.net/snippets/464293)

1. f= file('obj.save', 'rb')
2. loaded\_obj= cPickle.load(f)
3. f.close()

可以同时pickle多个对象到同一个文件：

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/39060813)[在CODE上查看代码片](https://code.csdn.net/snippets/464293)

1. f= file('objects.save', 'wb')
2. forobj **in** [obj1, obj2, obj3]:
3. cPickle.dump(obj, f,protocol=cPickle.HIGHEST\_PROTOCOL)
4. f.close()

也可以按照同样的顺序载入：

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/39060813)[在CODE上查看代码片](https://code.csdn.net/snippets/464293)

1. f= file('objects.save', 'rb')
2. loaded\_objects= []
3. fori **in** range(3):
4. loaded\_objects.append(cPickle.load(f))
5. f.close()

短期序列化

如果有信心，pickle整个模型是个好办法。

这种情况是指，你在项目中执行同样的保存和重载操作，或者这个类已经稳定运行很久了。

通过定义\_\_getstate\_\_ method和\_\_setstate\_\_可以控制从项目中保存何种pickle。

如果模型类包含了正在使用数据集的链接，而又不想pickle每个模型实例，上述控制方法会很实用。

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/39060813)[在CODE上查看代码片](https://code.csdn.net/snippets/464293)

1. def\_\_getstate\_\_(self):
2. state = dict(self.\_\_dict\_\_)
3. **del** state['training\_set']
4. **return** state
6. def\_\_setstate\_\_(self, d):
7. self.\_\_dict\_\_.update(d)
8. self.training\_set =cPickle.load(file(self.training\_set\_file, 'rb'))

长期序列化

如果想要保存的类运行不稳定，例如有函数创建或者删除、类成员重命名，应该只保存或载入类的不可变部分。

依然是使用定义\_\_getstate\_\_ method和\_\_setstate\_\_

例如只想要保存权重矩阵W和偏倚项b：

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/39060813)[在CODE上查看代码片](https://code.csdn.net/snippets/464293)

1. def\_\_getstate\_\_(self):
2. **return** (self.W, self.b)
4. def\_\_setstate\_\_(self, state):
5. W, b = state
6. self.W = W
7. self.b = b

如果更新了下列函数来表现变量名称的改变，那么即使W和b被重命名为weights和bias，之前的pickle文件依然是可用的：

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/39060813)[在CODE上查看代码片](https://code.csdn.net/snippets/464293)

1. def\_\_getstate\_\_(self):
2. **return** (self.weights, self.bias)
4. def\_\_setstate\_\_(self, state):
5. W, b = state
6. self.weights = W
7. self.bias = b

条件

-IfElse与Switch

-switch比ifelse更通用，因为switch是逐位操作。

-switch把2个输出变量都计算了，所以比ifelse要慢（只算1个）。

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/39060813)[在CODE上查看代码片](https://code.csdn.net/snippets/464293)

1. **from** theano **import** tensor as T
2. **from** theano.ifelse **import** ifelse
3. importtheano, time, numpy
5. a,b= T.scalars('a', 'b')
6. x,y= T.matrices('x', 'y')
8. z\_switch= T.switch(T.lt(a, b), T.mean(x), T.mean(y))
9. z\_lazy= ifelse(T.lt(a, b), T.mean(x), T.mean(y))
11. f\_switch= theano.function([a, b, x, y], z\_switch,
12. mode=theano.Mode(linker='vm'))
13. f\_lazyifelse= theano.function([a, b, x, y], z\_lazy,
14. mode=theano.Mode(linker='vm'))
16. val1= 0.
17. val2= 1.
18. big\_mat1= numpy.ones((10000, 1000))
19. big\_mat2= numpy.ones((10000, 1000))
21. n\_times= 10
23. tic= time.clock()
24. fori **in** xrange(n\_times):
25. f\_switch(val1, val2, big\_mat1, big\_mat2)
26. **print**'time spent evaluating both values %f sec' % (time.clock() - tic)
28. tic= time.clock()
29. fori **in** xrange(n\_times):
30. f\_lazyifelse(val1, val2, big\_mat1,big\_mat2)
31. **print**'time spent evaluating one value %f sec' % (time.clock() - tic)

测试结果

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/39060813)[在CODE上查看代码片](https://code.csdn.net/snippets/464293)

1. time spent evaluating both values 0.200000 sec

[让Caffe生成的数据集能在Theano上直接运行（一）——lmdb与protobuf](http://blog.csdn.net/ycheng_sjtu/article/details/40361947)

分类： [Deep Learning](http://blog.csdn.net/u012428391/article/category/2606893)2014-10-22 01:10 2925人阅读 [评论](http://blog.csdn.net/ycheng_sjtu/article/details/40361947#comments)(5) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/ycheng_sjtu/article/details/40361947#report)

[Theano](http://www.csdn.net/tag/Theano)[python](http://www.csdn.net/tag/python)[Caffe](http://www.csdn.net/tag/Caffe)[buffer](http://www.csdn.net/tag/buffer)

不论使用何种框架进行CNNs训练，共有3种数据集：

Training Set用于训练网络。

Validation Set用于训练时测试网络准确率。

Test Set用于测试网络训练完成后的最终正确率。

Caffe生成的数据分为2种格式：Lmdb和Leveldb。

它们都是键/值对（Key/Value Pair）嵌入式数据库管理系统编程库。

虽然lmdb的内存消耗是leveldb的1.1倍，但是lmdb的速度比leveldb快10%至15%，更重要的是lmdb允许多种训练模型同时读取同一组数据集。

因此lmdb取代了leveldb成为Caffe默认的数据集生成格式。

Google Protocol Buffer的安装

Protocol Buffer是一种类似于XML的用于序列化数据的自动机制。

首先在Protocol Buffers的中下载最新版本：

<https://developers.google.com/protocol-buffers/docs/downloads>

解压后运行：

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/40361947)[在CODE上查看代码片](https://code.csdn.net/snippets/492054)

1. ./configure
2. $ make
3. $ make check
4. $ make install
5. pip installprotobuf

添加动态链接库

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/40361947)[在CODE上查看代码片](https://code.csdn.net/snippets/492054)

1. exportLD\_LIBRARY\_PATH=/usr/local/lib:$LD\_LIBRARY\_PATH

Lmdb的安装

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/40361947)[在CODE上查看代码片](https://code.csdn.net/snippets/492054)

1. pip install lmdb

要parse（解析）一个protobuf类型数据，首先要告诉计算机你这个protobuf数据内部是什么格式（有哪些项，这些项各是什么数据类型的决定了占用多少字节，这些项可否重复，重复几次），安装protobuf这个module就可以用protobuf专用的语法来定义这些格式（这个是.proto文件）了，然后用protoc来编译这个.proto文件就可以生成你需要的目标文件。

想要定义自己的.proto文件请阅读：

<https://developers.google.com/protocol-buffers/docs/proto?hl=zh-cn>

编译.proto文件

**[python]** [view plaincopy](http://blog.csdn.net/ycheng_sjtu/article/details/40361947)[在CODE上查看代码片](https://code.csdn.net/snippets/492054)

1. protoc--proto\_path=IMPORT\_PATH --cpp\_out=DST\_DIR --java\_out=DST\_DIR--python\_out=DST\_DIR path/to/file.proto

--proto\_path 也可以简写成-I 是.proto所在的路径

输出路径：

--cpp\_out 要生成C++可用的头文件，分别是\*\*\*.pb.h（包含申明类）\*\*\*.pb.cc（包含可执行类），使用的时候只要include “\*\*\*.pb.h”

--java\_out 生成java可用的头文件

--python\_out 生成python可用的头文件，\*\*\_pb2.py，使用的时候import\*\*\_pb2.py即可

最后一个参数就是你的.proto文件完整路径。