**LINEAR EQUATION**

**REPORT**



**School or Department:** Wu Han University

**Grade and Specialty:** Excellent engineer 2016

**Name:** Guo Yang, Kong Chuishun, Tang Rui

**Advisor:** Fangling Pu

May 2 2018

【**Abstract**】In this lab, we consider vector-valued linear and aﬃne functions, and systems of linear equations. And we use a python program to familiar it. In the application, we introduce a method to train Binarized Neural Networks (BNNs) - neural networks with binary weights and activations.

**【Key words】** linear equation; BNN, Multidimensional linear function model

**Table of Contents**

[Abstract 1](#_Toc324279571)

[Chapter 1 Linear and aﬃne functions 1](#_Toc324279574)

[1.1 Linear and aﬃne functions 1](#_Toc324279574)

[1.1.1 Vector-valued functions of vectors 1](#_Toc324279574)

[1.1.2 The matrix-vector product function 3](#_Toc324279574)

[1.1.3 Examples 1](#_Toc324279574)

[1.1.4 Aﬃne functions 1](#_Toc324279574)

[1.2 Systems of linear equations 4](#_Toc324279575)

[Chapter 2 Our experiment 7](#_Toc324279576)

[2.1 Brief Introduction 1](#_Toc324279574)

[2.2 Specific operation 1](#_Toc324279574)

[2.3 Analysis about Results 1](#_Toc324279574)

[Bibliography 7](#_Toc324279590)

[Appendix 8](#_Toc324279591)

**Chapter 1 Introduction**

* 1. **Linear and aﬃne functions**

1.1.1Vector-valued functions of vectors.

The notation f : Rn → Rm means that *f* is a function that maps real n-vectors to real m-vectors. The value of the function *f*, evaluated at an n-vector x, is an m-vector f(x) = (*f1(x),f2(x),...,fm(x)*). Each of the components *fi* of *f* is itself a scalar-valued function of *x*. As with scalar-valued functions, we sometimes write *f(x) = f(x1,x2,...,xn)* to emphasize that f is a function of n scalar arguments. We use the same notation for each of the components of f, writing *fi(x)* = *fi(x1,x2,...,xn)* to emphasize that fi is a function mapping the scalar arguments x1,...,xn into a scalar.

**1.1.2 The matrix-vector product function**

Suppose A is an m×n matrix. We can deﬁne a function f : Rn → Rm by *f(x) = Ax*. The inner product function *f : Rn → R*, deﬁned as *f(x) = aTx*, , is the special case with m = 1.

**1.1.3 Superposition and linearity.**

The function *f : Rn → Rm*, deﬁned by *f(x) = Ax*, is linear, i.e., it satisﬁes the superposition property:

*f(αx + βy) = αf(x) + βf(y)*  (1.1)

holds for all n-vectors x and y and all scalars *α* and *β*. It is a good exercise to parse this simple looking equation, since it involves overloading of notation. On the left-hand side, the scalar-vector multiplications *αx* and *βy* involve n-vectors, and the sum αx+βy is the sum of two n-vectors. The function f maps n-vectors to m-vectors, so *f(αx+βy)* is an m-vector. On the right-hand side, the scalar-vector multiplications and the sum are those for m-vectors. Finally, the equality sign is equality between two m-vectors.

We can verify that superposition holds for f using properties of matrix-vector and scalar-vector multiplication:

*f(αx + βy) = A(αx + βy)*

*= A(αx) + A(βy)*

*= α(Ax) + β(Ay)*

*= αf(x) + βf(y)*

Thus we can associate with every matrix *A* a linear function *f(x) = Ax.*

The converse is also true. Suppose f is a function that maps n-vectors to m vectors, and is linear, i.e., (1.1) holds for all n-vectors x and y and all scalars *α* and *β*. Then there exists an m×n matrix A such that *f(x) = Ax* for all x. This can be shown in the same way as for scalar-valued functions, by showing that if f is linear, then

*f(x) = x1f(e1) + x2f(e2) +···+ xnf(en)  1.2*

where ek is the kth unit vector of size n. The right-hand side can also be written as a matrix-vector product Ax, with

*A = [ f(e1) f(e2) ··· f(en) ]*

It is easily shown that the matrix-vector representation of a linear function is unique. If f : Rn → Rm is a linear function, then there exists exactly one matrix A such that f(x) = Ax for all x.

**1.1.4 Examples of linear functions**

In the examples below we deﬁne functions *f* that map n-vectors *x* to *n*-vectors *f(x)*. Each function is described in words, in terms of its eﬀect on an arbitrary x. In each case we give the associated matrix multiplication representation.

• Negation. f changes the sign of *x: f(x) = −x.*

Negation can be expressed as *f(x) = Ax* with *A* = *−I*.

• Reversal. f reverses the order of the elements of *x: f(x) = (xn,xn−1,...,x1).* The reversal function can be expressed as *f(x) = Ax* with

*A* =    

(This is the n×n identity matrix with the order of its columns reversed.

**1.1.5 Aﬃne functions.**

A vector-valued function f : Rn → Rm is called aﬃne if it can be expressed as f(x) = Ax + b, where A is an m×n matrix and b is an m-vector. It can be shown that a function f : Rn → Rm is aﬃne if and only if

*f(αx + βy) = αf(x) + βf(y)*

holds for all n-vectors x, y, and all scalars α, β that satisfy *α + β = 1*. In other words, superposition holds for aﬃne combinations of vectors. (For linear functions, superposition holds for any linear combinations of vectors.) The matrix A and the vector b in the representation of an aﬃne function as f(x) = Ax + b are unique. These parameters can be obtained by evaluating f at the vectors 0, *e1,...,en*, where *ek* is the kth unit vector in Rn. We have

A =[ f(e1)−f(0) f(e2)−f(0) ··· f(en)−f(0) ] , b = f(0).

Just like aﬃne scalar-valued functions, aﬃne vector-valued functions are often called linear, even though they are linear only when the vector b is zero.

**1.2 Systems of linear equations**

Consider a set (also called a system) of *m* linear equations in *n* variables or unknowns,

.

.

.

The numbers are called the coefficients in the linear equations, and the numbers are called the right-hand sides (since by tradition, they appear on the right-hand side of the equation). These equations can be written succinctly in matrix notation as

In this context, the *m × n* matrix A is called the *coefficient matrix,* and the *m*-vector b is called the *right-hand* side. An *n*-vector x is called a *solution* of the linear equations if holds. A set of linear equations can have no solutions, one solution, or multiple solutions.

The set of linear equations is called over-determined if *m > n*, under-determined if *m < n*, and square if *m = n*; these correspond to the coefficient matrix being tall, wide, and square, respectively. When the system of linear equations is over-determined, there are more equations than variables or unknowns. When the system of linear equations is under-determined, there are more unknowns than equations. When the system of linear equations is square, the numbers of unknowns and equations is the same. A set of equations with zero right-hand side, *Ax =* 0, is called a homogeneous set of equations. Any homogeneous set of equations has *x =* 0 as a solution.

Base on the basic conception, we have several usages. For example, *Leontief input-output model*.

We consider an economy with n different industrial sectors. We let be the economic activity level, or total production output, of sector *i*, for *i =* 1*, …, n*, measured in a common unit, such as (billions of) dollars. The output of each sector flows to other sectors, to support their production, and also to consumers. We denote the total consumer demand for sector *i* as , for *i* = 1, …, *n*. Supporting the output level xj for sector j requires output for sector *i*.

We refer to as the sector i input that flows to sector *i*. (We can have 0; for example, it requires some energy to support the production of energy.) Thus, is the total sector *i* output required by, or flowing into, the n industrial sectors. The matrix *A* is called the input-output matrix of the economy, since it describes the flows of sector outputs to the inputs of itself and other sectors. The vector gives the sector outputs required to support the production levels given by *x*.

**Chapter 2 Our experiment**

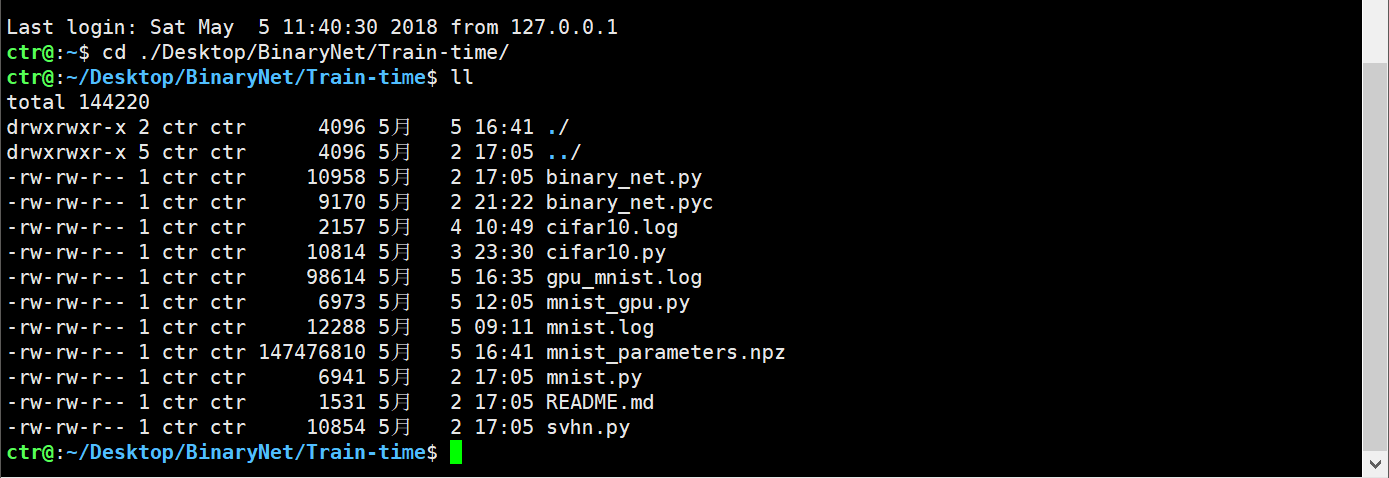
**2.1 Brief Introduction**

Based on what we have learned and what we three had done in DSP Laboratory, we want to establish a multidimensional linear function model to train *Cifar10* dataset using binarized neural network.

*Cifar10* is a collection of data sets for universal object recognition. It is an advanced science project research institute led by the Canadian government. *Cifar10* consists of 60, 000 32\*32 RGB color images in 10 categories. 50,000 training, 10,000 tests (cross validation). The biggest feature of this dataset is that it migrates to universal objects and applies to multiple categories.

**2.2 Specific operation**

As follow figure, we login DSP laboratory by Xshell5 and have installed our lab environment including annocoda2.7, pylearn2, lasagna, Theano, Cuda and Pycuda. It cost me two days to config all environment well and that tortures me a lot.

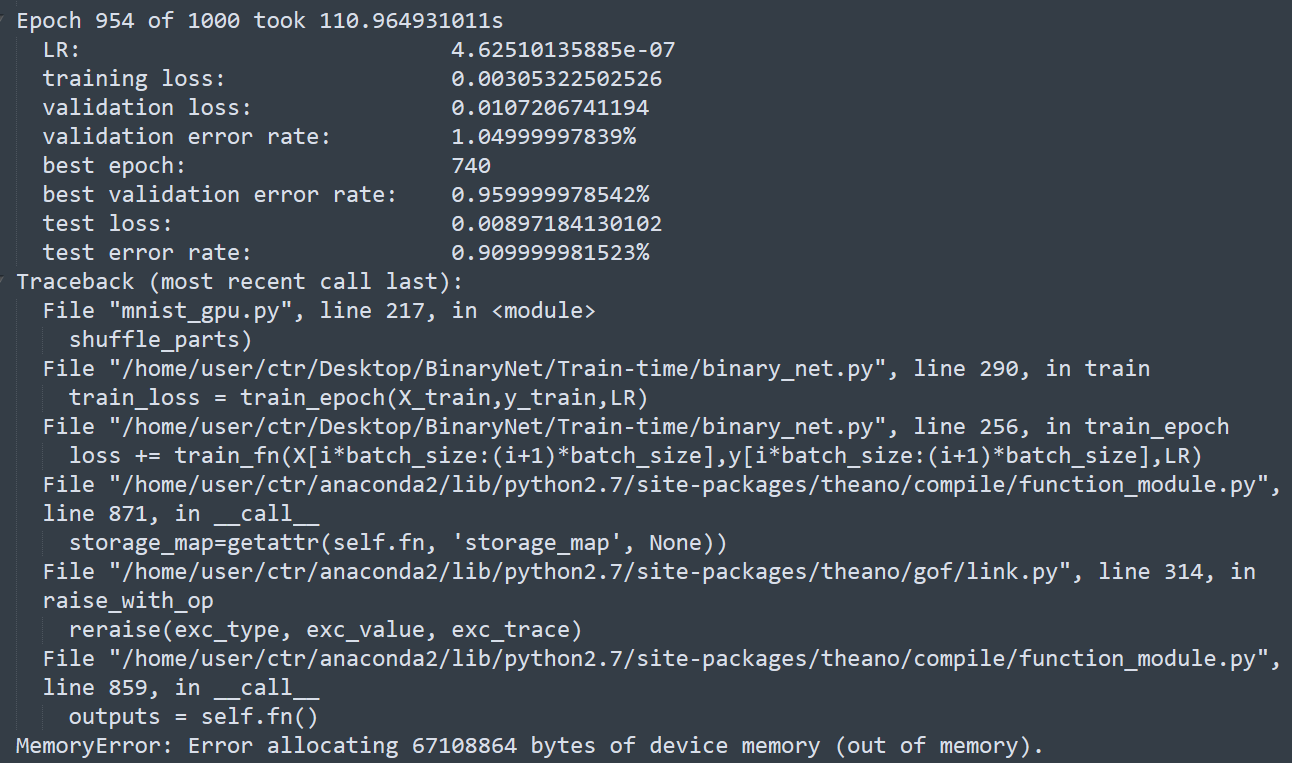
****Anyway, I successfully solve those problems. Finally we can run our machine learning code using the tesla k80 GPU. Following are our notes about this lab code(train cifar10 dataset using binarized neural network).

1. ##################################################
2. #
3. # 结构:
4. #
5. # 输入层
6. #   |
7. # 卷积层1 + BN层 + 激活层
8. #   |
9. # 卷积层2 + 最大值混合层2 + BN层 + 激活层   (binary)
10. #   |
11. # 卷积层3 + BN层 + 激活层
12. #   |
13. # 卷积层4 + 最大值混合层4 + BN层 + 激活层
14. #   |
15. # 卷积层5 + BN层 + 激活层
16. #   |
17. # 卷积层6 + 最大值混合层6 + BN层 + 激活层
18. #   |
19. # 全连接层1 + BN层 + 激活层  (binary)
20. #   |
21. # 全连接层2 + BN层 + 激活层
22. #   |
23. # 全连接层3 + BN层
24. #   |
25. # 输出
26. #
27. ##################################################
29. # theano.function(inputs, outputs=None, mode=None, updates=None, givens=None):
30. #   根据输入inputs计算数据outputs的函数，其中:
31. #   inputs:  列表类型，用来保存输入量
32. #   outputs: 列表或字典类型，用来保存输出量。输入量与输出量的映射关系通常在输出量的定义中体现。
33. #   updates: 一组可迭代更新的量(shared\_variable, new\_expression),
34. #       对其中的shared\_variable输入用new\_expression表达式更新
35. #
36. # lasagne.updates.adam(loss\_or\_grads, params, learning\_rate): 用于参数更新，
37. #   其中: loss\_or\_grads: 误差或梯度
38. #   params:             要更新的参数
39. #   learning\_rate:      更新速率(学习速率)
40. # lasagne.layers.get\_output(layer):            对指定网络，计算网络输出;
41. # lasagne.objectives.categorical\_crossentropy(predictions, targets): 计算分类结果与目标的交叉熵(误差);
42. # lasagne.layers.get\_all\_params(layer):        返回一个列表，包含该层参数的theano共享变量或表达式 ?
43. # bnn\_utils.compute\_grads(loss, network):      计算梯度;
44. # bnn\_utilsclipping\_scaling(updates, network): 该函数在参数更新后规范化;
45. # OrderedDict:                                 有序字典类;
46. # dict.items():                                返回字典的键值;
47. # tensor.nep():                                相当于"a != b";
48. # tensor.argmax():                             返回沿指定轴取得最大值的下标;
49. **from** \_\_future\_\_ **import** print\_function
51. **import** sys
52. **import** os
53. **import** time
54. **import** numpy as np
55. np.random.seed(1234) # for reproducibility?
57. # specifying the gpu to use
58. # import theano.sandbox.cuda
59. # theano.sandbox.cuda.use('gpu1')
60. **import** theano
61. **import** theano.tensor as T
62. **import** lasagne
63. **import** cPickle as pickle
64. **import** gzip
66. **import** binary\_net
68. **from** pylearn2.datasets.zca\_dataset **import** ZCA\_Dataset
69. **from** pylearn2.datasets.cifar10 **import** CIFAR10
70. **from** pylearn2.utils **import** serial
72. **from** collections **import** OrderedDict
74. **if** \_\_name\_\_ == "\_\_main\_\_":
76. # BN parameters (设定超参数)
77. batch\_size = 50
78. **print**("batch\_size = "+str(batch\_size))
79. # alpha is the exponential moving average factor
80. alpha = .1
81. **print**("alpha = "+str(alpha))
82. epsilon = 1e-4
83. **print**("epsilon = "+str(epsilon))
85. # BinaryOut
86. activation = binary\_net.binary\_tanh\_unit
87. **print**("activation = binary\_net.binary\_tanh\_unit")
88. # activation = binary\_net.binary\_sigmoid\_unit
89. # print("activation = binary\_net.binary\_sigmoid\_unit")
91. # BinaryConnect    (设定BNN-神经网络标识)
92. binary = True
93. **print**("binary = "+str(binary))
94. stochastic = False
95. **print**("stochastic = "+str(stochastic))
96. # (-H,+H) are the two binary values
97. # H = "Glorot"
98. H = 1.
99. **print**("H = "+str(H))
100. # W\_LR\_scale = 1.
101. W\_LR\_scale = "Glorot" # "Glorot" means we are using the coefficients from Glorot's paper
102. **print**("W\_LR\_scale = "+str(W\_LR\_scale))
104. # Training parameters
105. num\_epochs = 500
106. **print**("num\_epochs = "+str(num\_epochs))
108. # Decaying LR   (设定学习速率)
109. LR\_start = 0.001
110. **print**("LR\_start = "+str(LR\_start))
111. LR\_fin = 0.0000003
112. **print**("LR\_fin = "+str(LR\_fin))
113. LR\_decay = (LR\_fin/LR\_start)\*\*(1./num\_epochs)
114. **print**("LR\_decay = "+str(LR\_decay))
115. # BTW, LR decay might good for the BN moving average...
117. train\_set\_size = 45000
118. **print**("train\_set\_size = "+str(train\_set\_size))
119. shuffle\_parts = 1
120. **print**("shuffle\_parts = "+str(shuffle\_parts))
122. **print**('Loading CIFAR-10 dataset...')    (加载数据)
124. train\_set = CIFAR10(which\_set="train",start=0,stop = train\_set\_size)
125. valid\_set = CIFAR10(which\_set="train",start=train\_set\_size,stop = 50000)
126. test\_set = CIFAR10(which\_set="test")
128. # bc01 format
129. # Inputs in the range [-1,+1]
130. # print("Inputs in the range [-1,+1]")
131. train\_set.X = np.reshape(np.subtract(np.multiply(2./255.,train\_set.X),1.),(-1,3,32,32))
132. valid\_set.X = np.reshape(np.subtract(np.multiply(2./255.,valid\_set.X),1.),(-1,3,32,32))
133. test\_set.X = np.reshape(np.subtract(np.multiply(2./255.,test\_set.X),1.),(-1,3,32,32))
135. # flatten targets
136. train\_set.y = np.hstack(train\_set.y)
137. valid\_set.y = np.hstack(valid\_set.y)
138. test\_set.y = np.hstack(test\_set.y)
140. # Onehot the targets
141. train\_set.y = np.float32(np.eye(10)[train\_set.y])
142. valid\_set.y = np.float32(np.eye(10)[valid\_set.y])
143. test\_set.y = np.float32(np.eye(10)[test\_set.y])
145. # for hinge loss
146. train\_set.y = 2\* train\_set.y - 1.
147. valid\_set.y = 2\* valid\_set.y - 1.
148. test\_set.y = 2\* test\_set.y - 1.
150. **print**('Building the CNN...')     (构建深度网络)
152. # Prepare Theano variables for inputs and targets
153. input = T.tensor4('inputs')
154. target = T.matrix('targets')
155. LR = T.scalar('LR', dtype=theano.config.floatX)
157. cnn = lasagne.layers.InputLayer()
159. # 128C3-128C3-P2
160. cnn = binary\_net.Conv2DLayer()
161. cnn = lasagne.layers.BatchNormLayer()
162. cnn = lasagne.layers.NonlinearityLayer()
164. cnn = binary\_net.Conv2DLayer()
165. cnn = lasagne.layers.MaxPool2DLayer(cnn, pool\_size=(2, 2))
166. cnn = lasagne.layers.BatchNormLayer()
167. cnn = lasagne.layers.NonlinearityLayer()
169. # 256C3-256C3-P2
170. cnn = binary\_net.Conv2DLayer()
171. cnn = lasagne.layers.BatchNormLayer()
172. cnn = lasagne.layers.NonlinearityLayer()
174. cnn = binary\_net.Conv2DLayer()
175. cnn = lasagne.layers.MaxPool2DLayer(cnn, pool\_size=(2, 2))
176. cnn = lasagne.layers.BatchNormLayer()
177. cnn = lasagne.layers.NonlinearityLayer()
179. # 512C3-512C3-P2
180. cnn = binary\_net.Conv2DLayer()
181. cnn = lasagne.layers.BatchNormLayer()
182. cnn = lasagne.layers.NonlinearityLayer()
184. cnn = binary\_net.Conv2DLayer()
185. cnn = lasagne.layers.MaxPool2DLayer(cnn, pool\_size=(2, 2))
186. cnn = lasagne.layers.BatchNormLayer()
187. cnn = lasagne.layers.NonlinearityLayer()
189. # print(cnn.output\_shape)
190. # 1024FP-1024FP-10FP
191. cnn = binary\_net.DenseLayer()
192. cnn = lasagne.layers.BatchNormLayer()
193. cnn = lasagne.layers.NonlinearityLayer()
195. cnn = binary\_net.DenseLayer()
196. cnn = lasagne.layers.BatchNormLayer()
197. cnn = lasagne.layers.NonlinearityLayer()
199. cnn = binary\_net.DenseLayer()
200. cnn = lasagne.layers.BatchNormLayer()
201. #(计算网络输出)
202. train\_output = lasagne.layers.get\_output(cnn, deterministic=False)
204. # squared hinge loss   (定义误差计算函数)
205. loss = T.mean(T.sqr(T.maximum(0.,1.-target\*train\_output)))
207. **if** binary:
209. # W updates   (权重更新)
210. W = lasagne.layers.get\_all\_params(cnn, binary=True)
211. W\_grads = binary\_net.compute\_grads(loss,cnn)
212. updates = lasagne.updates.adam(loss\_or\_grads=W\_grads, params=W, learning\_rate=LR)
213. updates = binary\_net.clipping\_scaling(updates,cnn)
215. # other parameters updates    (其它参数更新)
216. params = lasagne.layers.get\_all\_params(cnn, trainable=True, binary=False)
217. updates = OrderedDict(updates.items() + lasagne.updates.adam(loss\_or\_grads=loss, params=params, learning\_rate=LR).items())
219. **else**:
220. # (参数更新)
221. params = lasagne.layers.get\_all\_params(cnn, trainable=True)
222. updates = lasagne.updates.adam(loss\_or\_grads=loss, params=params, learning\_rate=LR)
224. # test prediction and loss expressions (测试数据和误差表示)
225. test\_output = lasagne.layers.get\_output(cnn, deterministic=True)
226. test\_loss = T.mean(T.sqr(T.maximum(0.,1.-target\*test\_output)))
227. test\_err = T.mean(T.neq(T.argmax(test\_output, axis=1), T.argmax(target, axis=1)),dtype=theano.config.floatX)
229. (建立训练数据和测试数据的theano函数)
230. # Compile a function performing a training step on a mini-batch (by giving the updates dictionary)
231. # and returning the corresponding training loss:
232. train\_fn = theano.function([input, target, LR], loss, updates=updates)
234. # Compile a second function computing the validation loss and accuracy:
235. val\_fn = theano.function([input, target], [test\_loss, test\_err])
237. **print**('Training...')
238. binary\_net.train(
239. train\_fn,val\_fn,              # 训练和测试数据的theano函数
240. cnn,                          # 神经网络
241. batch\_size,                   # 小批量数据大小
242. LR\_start,LR\_decay,            # 学习速率
243. num\_epochs,                   # 迭代期次数
244. train\_set.X,train\_set.y,      # 训练数据
245. valid\_set.X,valid\_set.y,      # 验证数据
246. test\_set.X,test\_set.y,        # 测试数据
247. shuffle\_parts=shuffle\_parts)

**2.3 Analysis about Results**

But this code need to run two days. So, we can’t write the answer into our report. Fortunately, we also utilize the *mnist* version using binarized neutral network. And it only cost 8 ours to get the parameters. (*mnist* is also a train set which is like *Cifar10.* But it is simpler.)

This is the 954-epoch answer on *mnist* datasets. The k80 GPU memory is too busy。



**Bibliography**

[1] Binarized Neural Networks : Training Neural Networks with Weights and Activations Constrained to +1 or−1

[2]Bahdanau,Dzmitry,Cho,Kyunghyun,andBengio,Yoshua. Neural machine translation by jointly learning to align and translate. In ICLR’2015, arXiv:1409.0473, 2015.