import numpy as np

import sys

import tensorflow as tf

from tensorflow.examples.tutorials.mnist import input\_data

def conv2(X, k):

# as a demo code, here we ignore the shape check

x\_row, x\_col = X.shape

k\_row, k\_col = k.shape

ret\_row, ret\_col = x\_row - k\_row + 1, x\_col - k\_col + 1

ret = np.empty((ret\_row, ret\_col))

for y in range(ret\_row):

for x in range(ret\_col):

sub = X[y : y + k\_row, x : x + k\_col]

ret[y,x] = np.sum(sub \* k)

return ret

def rot180(in\_data):

ret = in\_data.copy()

yEnd = ret.shape[0] - 1

xEnd = ret.shape[1] - 1

for y in range(ret.shape[0] / 2):

for x in range(ret.shape[1]):

ret[yEnd - y][x] = ret[y][x]

for y in range(ret.shape[0]):

for x in range(ret.shape[1] / 2):

ret[y][xEnd - x] = ret[y][x]

return ret

def padding(in\_data, size):

cur\_r, cur\_w = in\_data.shape[0], in\_data.shape[1]

new\_r = cur\_r + size \* 2

new\_w = cur\_w + size \* 2

ret = np.zeros((new\_r, new\_w))

ret[size:cur\_r + size, size:cur\_w+size] = in\_data

return ret

def discreterize(in\_data, size):

num = in\_data.shape[0]

ret = np.zeros((num, size))

for i, idx in enumerate(in\_data):

ret[i, idx] = 1

return ret

class ConvLayer:

def \_\_init\_\_(self, in\_channel, out\_channel, kernel\_size, lr=0.01, momentum=0.9, name='Conv'):

self.w = np.random.randn(in\_channel, out\_channel, kernel\_size, kernel\_size)

self.b = np.zeros((out\_channel))

self.layer\_name = name

self.lr = lr

self.momentum = momentum

self.prev\_gradient\_w = np.zeros\_like(self.w)

self.prev\_gradient\_b = np.zeros\_like(self.b)

def forward(self, in\_data):

# assume the first index is channel index

print 'conv forward:' + str(in\_data.shape)

in\_batch, in\_channel, in\_row, in\_col = in\_data.shape

out\_channel, kernel\_size = self.w.shape[1], self.w.shape[2]

self.top\_val = np.zeros((in\_batch, out\_channel, in\_row - kernel\_size + 1, in\_col - kernel\_size + 1))

self.bottom\_val = in\_data

for b\_id in range(in\_batch):

for o in range(out\_channel):

for i in range(in\_channel):

self.top\_val[b\_id, o] += conv2(in\_data[b\_id, i], self.w[i, o])

self.top\_val[b\_id, o] += self.b[o]

return self.top\_val

def backward(self, residual):

in\_channel, out\_channel, kernel\_size = self.w.shape

in\_batch = residual.shape[0]

# gradient\_b

self.gradient\_b = residual.sum(axis=3).sum(axis=2).sum(axis=0) / self.batch\_size

# gradient\_w

self.gradient\_w = np.zeros\_like(self.w)

for b\_id in range(in\_batch):

for i in range(in\_channel):

for o in range(out\_channel):

self.gradient\_w[i, o] += conv2(self.bottom\_val[b\_id], residual[o])

self.gradient\_w /= self.batch\_size

# gradient\_x

gradient\_x = np.zeros\_like(self.bottom\_val)

for b\_id in range(in\_batch):

for i in range(in\_channel):

for o in range(out\_channel):

gradient\_x[b\_id, i] += conv2(padding(residual, kernel\_size - 1), rot180(self.w[i, o]))

gradient\_x /= self.batch\_size

# update

self.prev\_gradient\_w = self.prev\_gradient\_w \* self.momentum - self.gradient\_w

self.w += self.lr \* self.prev\_gradient\_w

self.prev\_gradient\_b = self.prev\_gradient\_b \* self.momentum - self.gradient\_b

self.b += self.lr \* self.prev\_gradient\_b

return gradient\_x

class FCLayer:

def \_\_init\_\_(self, in\_num, out\_num, lr = 0.01, momentum=0.9):

self.\_in\_num = in\_num

self.\_out\_num = out\_num

self.w = np.random.randn(in\_num, out\_num)

self.b = np.zeros((out\_num, 1))

self.lr = lr

self.momentum = momentum

self.prev\_grad\_w = np.zeros\_like(self.w)

self.prev\_grad\_b = np.zeros\_like(self.b)

def forward(self, in\_data):

print 'fc forward=' + str(in\_data.shape)

self.topVal = np.dot(self.w.T, in\_data) + self.b

self.bottomVal = in\_data

return self.topVal

def backward(self, loss):

batch\_size = loss.shape[0]

# residual\_z = loss \* self.topVal \* (1 - self.topVal)

grad\_w = np.dot(self.bottomVal, loss.T) / batch\_size

grad\_b = np.sum(loss) / batch\_size

residual\_x = np.dot(self.w, loss)

self.prev\_grad\_w = self.prev\_grad\_w \* momentum - grad\_w

self.prev\_grad\_b = self.prev\_grad\_b \* momentum - grad\_b

self.w -= self.lr \* self.prev\_grad\_w

self.b -= self.lr \* self.prev\_grad\_b

return residual\_x

class ReLULayer:

def \_\_init\_\_(self, name='ReLU'):

pass

def forward(self, in\_data):

self.top\_val = in\_data

ret = in\_data.copy()

ret[ret < 0] = 0

return ret

def backward(self, residual):

gradient\_x = residual.copy()

gradient\_x[self.top\_val < 0] = 0

return gradient\_x

class MaxPoolingLayer:

def \_\_init\_\_(self, kernel\_size, name='MaxPool'):

self.kernel\_size = kernel\_size

def forward(self, in\_data):

in\_batch, in\_channel, in\_row, in\_col = in\_data.shape

k = self.kernel\_size

out\_row = in\_row / k + (1 if in\_row % k != 0 else 0)

out\_col = in\_col / k + (1 if in\_col % k != 0 else 0)

self.flag = np.zeros\_like(in\_data)

ret = np.empty((in\_batch, in\_channel, out\_row, out\_col))

for b\_id in range(in\_batch):

for c in range(in\_channel):

for oy in range(out\_row):

for ox in range(out\_col):

height = k if (oy + 1) \* k <= in\_row else in\_row - oy \* k

width = k if (ox + 1) \* k <= in\_col else in\_col - ox \* k

idx = np.argmax(in\_data[b\_id, c, oy \* k: oy \* k + height, ox \* k: ox \* k + width])

offset\_r = idx / width

offset\_c = idx % width

self.flag[b\_id, c, oy \* k + offset\_r, ox \* k + offset\_c] = 1

ret[b\_id, c, oy, ox] = in\_data[b\_id, c, oy \* k + offset\_r, ox \* k + offset\_c]

return ret

def backward(self, residual):

in\_batch, in\_channel, in\_row, in\_col = self.flag

k = self.kernel\_size

out\_row, out\_col = residual.shape[2], residual.shape[3]

gradient\_x = np.zeros\_like(self.flag)

for b\_id in range(in\_batch):

for c in range(in\_channel):

for oy in range(out\_row):

for ox in range(out\_col):

height = k if (oy + 1) \* k <= in\_row else in\_row - oy \* k

width = k if (ox + 1) \* k <= in\_col else in\_col - ox \* k

gradient\_x[b\_id, c, oy \* k + offset\_r, ox \* k + offset\_c] = residual[b\_id, c, oy, ox]

gradient\_x[self.flag == 0] = 0

return gradient\_x

class FlattenLayer:

def \_\_init\_\_(self, name='Flatten'):

pass

def forward(self, in\_data):

self.in\_batch, self.in\_channel, self.r, self.c = in\_data.shape

return in\_data.reshape(self.in\_batch, self.in\_channel \* self.r \* self.c)

def backward(self, residual):

return residual.reshape(self.in\_batch, self.in\_channel, self.r, self.c)

class SoftmaxLayer:

def \_\_init\_\_(self, name='Softmax'):

pass

def forward(self, in\_data):

exp\_out = np.exp(in\_data)

self.top\_val = exp\_out / np.sum(exp\_out, axis=1)

return self.top\_val

def backward(self, residual):

return self.top\_val - residual

class Net:

def \_\_init\_\_(self):

self.layers = []

def addLayer(self, layer):

self.layers.append(layer)

def train(self, trainData, trainLabel, validData, validLabel, batch\_size, iteration):

train\_num = trainData.shape[0]

for iter in range(iteration):

print 'iter=' + str(iter)

for batch\_iter in range(0, train\_num, batch\_size):

print('batch\_iter=', batch\_iter)

if batch\_iter + batch\_size < train\_num:

self.train\_inner(trainData[batch\_iter: batch\_iter + batch\_size],

trainLabel[batch\_iter: batch\_iter + batch\_size])

else:

self.train\_inner(trainData[batch\_iter: train\_num],

trainLabel[batch\_iter: train\_num])

print "eval=" + str(self.eval(validData, validLabel))

def train\_inner(self, data, label):

lay\_num = len(self.layers)

in\_data = data

for i in range(lay\_num):

out\_data = self.layers[i].forward(in\_data)

in\_data = out\_data

residual\_in = label

for i in range(0, lay\_num, -1):

residual\_out = self.layers[i].backward(residual\_in)

residual\_in = residual\_out

def eval(self, data, label):

lay\_num = len(self.layers)

in\_data = data

for i in range(lay\_num):

out\_data = self.layers[i].forward(in\_data)

in\_data = out\_data

out\_idx = np.argmax(in\_data, axis=1)

label\_idx = np.argmax(label, axis=1)

return np.sum(out\_idx == label\_idx) / float(out\_idx.shape[0])

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

import struct

from array import array

def load\_data(data\_path, label\_path):

with open(label\_path, 'rb') as file:

magic, size = struct.unpack(">II", file.read(8))

if magic != 2049:

raise ValueError('Magic number mismatch, expected 2049,'

'got %d' % magic)

labels = array("B", file.read())

with open(data\_path, 'rb') as file:

magic, size, rows, cols = struct.unpack(">IIII", file.read(16))

if magic != 2051:

raise ValueError('Magic number mismatch, expected 2051,'

'got %d' % magic)

image\_data = array("B", file.read())

images = []

for i in xrange(size):

images.append([0]\*rows\*cols)

for i in xrange(size):

images[i][:] = image\_data[i\*rows\*cols : (i+1)\*rows\*cols]

return np.array(images), np.array(labels)

# use my\_image dataset

# train\_feature\_raw, train\_label\_raw = load\_data('train.feat', 'train.label')

# valid\_feature\_raw, valid\_label\_raw = load\_data('valid.feat', 'valid.label')

mnist = input\_data.read\_data\_sets('my\_image\_folder/', one\_hot=False)

train\_feature\_raw, train\_label\_raw, valid\_feature\_raw, valid\_label\_raw = mnist.train.images, mnist.train.labels, mnist.test.images, mnist.test.labels

train\_feature = train\_feature\_raw.reshape(55000, 1, 28, 28)

valid\_feature = valid\_feature\_raw.reshape(10000, 1, 28, 28)

train\_label = discreterize(train\_label\_raw, 10)

valid\_label = discreterize(valid\_label\_raw, 10)

print 'load ok'

net = Net()

net.addLayer(ConvLayer(1, 20, 4, 0.01, 0.9))

net.addLayer(ReLULayer())

net.addLayer(MaxPoolingLayer(2))

net.addLayer(ConvLayer(20, 40, 5, 0.01, 0.9))

net.addLayer(ReLULayer())

net.addLayer(MaxPoolingLayer(3))

net.addLayer(FlattenLayer())

net.addLayer(FCLayer(40 \* 3 \* 3, 150, 0.01, 0.9))

net.addLayer(ReLULayer())

net.addLayer(FCLayer(150, 10, 0.01, 0.9))

net.addLayer(SoftmaxLayer())

print 'net build ok'

net.train(train\_feature, train\_label, valid\_feature, valid\_label, 16, 10)