```
import cPickle as pickle
import numpy as np
import os
#from scipy.misc import imread

def load_CIFAR_batch(filename):
  print "this is the file name\n"
  print filename
  print "yeah \n"

  """ load single batch of cifar """
  with open(filename, 'rb') as f:
    datadict = pickle.load(f)
    X = datadict['data']
    Y = datadict['labels']
    print len(X)
    print len(Y)

    X = X.reshape(10000, 3, 32, 32).transpose(0,2,3,1).astype("fl
oat")
    Y = np.array(Y)
    return X, Y

def load_CIFAR10(ROOT):
  print "this is the  ROOT\n"
  print ROOT
  print "yeah \n"
  """ load all of cifar """
  xs = []
  ys = []
  for b in range(1,6):
    f = os.path.join(ROOT, 'data_batch_%d' % (b, ))
    X, Y = load_CIFAR_batch(f)
    xs.append(X)
    ys.append(Y)
  Xtr = np.concatenate(xs)
  Ytr = np.concatenate(ys)
  del X, Y
  Xte, Yte = load_CIFAR_batch(os.path.join(ROOT, 'test_batch'))
  return Xtr, Ytr, Xte, Yte
```

In [2]:

```
import numpy as np
import matplotlib.pyplot as plt
import time

class TwoLayerNet(object):
  """
  A two-layer fully-connected neural network. The net
 has an input dimension of
  N, a hidden layer dimension of H, and performs clas
sification over C classes.
  We train the network with a softmax loss function a
nd L2 regularization on the
  weight matrices. The network uses a ReLU nonlineari
ty after the first fully
  connected layer.

  In other words, the network has the following archi
tecture:

  input - fully connected layer - ReLU - fully connec
ted layer - softmax

  The outputs of the second fully-connected layer are
```

```
the scores for each class.
  """

  def __init__(self, input_size, hidden_size, output_
size, std=1e-4,
    init_method="Normal"):
    """
    Initialize the model. Weights are initialized to
 small random values and
    biases are initialized to zero. Weights and biase
s are stored in the
    variable self.params, which is a dictionary with
 the following keys:

    W1: First layer weights; has shape (D, H)
    b1: First layer biases; has shape (H,)
    W2: Second layer weights; has shape (H, C)
    b2: Second layer biases; has shape (C,)

    Inputs:
    - input_size: The dimension D of the input data.
    - hidden_size: The number of neurons H in the hid
den layer.
    - output_size: The number of classes C.
    """
    self.params = {}
    self.params['W1'] = std * np.random.randn(input_s
ize, hidden_size)
    self.params['b1'] = np.zeros(hidden_size)
    self.params['W2'] = std * np.random.randn(hidden_
size, output_size)
    self.params['b2'] = np.zeros(output_size)

    #special initialization
    if init_method=="i":
      self.params['W1']=np.random.randn(input_size,hi
dden_size)/np.sqrt(input_size)
      self.params['W2']=np.random.randn(hidden_size,o
utput_size)/np.sqrt(hidden_size)
    elif init_method=="io":
      self.params['W1']=np.random.randn(input_size,hi
dden_size)*np.sqrt(2.0/(input_size+hidden_size))
      self.params['W2']=np.random.randn(hidden_size,o
utput_size)*np.sqrt(2.0/(hidden_size+output_size))
    elif init_method=="ReLU":
      self.params['W1']=np.random.randn(input_size,hi
dden_size)*np.sqrt(2.0/input_size)
      self.params['W2']=np.random.randn(hidden_size,o
utput_size)*np.sqrt(2.0/(hidden_size+output_size))

  def loss(self, X, y=None, reg=0.0, dropout=0, dropM
ask=None,activation='Relu'):
    """
    Compute the loss and gradients for a two layer fu
lly connected neural
    network.

    Inputs:
    - X: Input data of shape (N, D). Each X[i] is a t
raining sample.
    - y: Vector of training labels. y[i] is the label
 for X[i], and each y[i] is
      an integer in the range 0 <= y[i] < C. This par
ameter is optional; if it
      is not passed then we only return scores, and i
f it is passed then we
```

```
        instead return the loss and gradients.
      - reg: Regularization strength.

      Returns:
      If y is None, return a matrix scores of shape (N,
 C) where scores[i, c] is
      the score for class c on input X[i].

      If y is not None, instead return a tuple of:
      - loss: Loss (data loss and regularization loss)
 for this batch of training
        samples.
      - grads: Dictionary mapping parameter names to gr
adients of those parameters
        with respect to the loss function; has the same
 keys as self.params.
      """
      # Unpack variables from the params dictionary
      W1, b1 = self.params['W1'], self.params['b1']
      W2, b2 = self.params['W2'], self.params['b2']
      N, D = X.shape

      # Compute the forward pass
      scores = None
      ##################################################
############################
      # TODO: Perform the forward pass, computing the c
lass scores for the input. #
      # Store the result in the scores variable, which
 should be an array of      #
      # shape (N, C).
                              #
      ##################################################
############################


      if activation=='leaky':
        inp=X.dot(W1)+b1
        a2=np.maximum(inp,.01*inp)
      else:
        a2=np.maximum(X.dot(W1)+b1,0)

      if dropout != 0 and dropout<1:
        a2*=(np.random.randn(*a2.shape)<dropout)/dropou
t
      elif dropout>1:
            W2*=dropMask['W2']/(dropout-1)
            b2*=dropMask['b2']/(dropout-1)
            # for convinient this is drop connect , dro
p rate= dropout-1

      scores=a2.dot(W2)+b2 # z3
      ##################################################
############################
      #                              END OF YOUR CODE
                              #
      ##################################################
############################

      # If the targets are not given then jump out, w
e're done
      if y is None:
        return scores

      # Compute the loss
      loss = None
```

```
    ##################################################
############################
    # TODO: Finish the forward pass, and compute the
 loss. This should include  #
    # both the data loss and L2 regularization for W1
 and W2. Store the result  #
    # in the variable loss, which should be a scalar.
 Use the Softmax           #
    # classifier loss. So that your results match our
s, multiply the            #
    # regularization loss by 0.5
                           #
    ##################################################
############################
    #do a softmax first
    if dropout>1:
        print dropMask['W2']
    exp_scores=np.exp(scores)

    a3=exp_scores/(np.sum(exp_scores,1))[:,None] #h
(x)

    loss=-np.sum(np.log(a3[range(len(a3)),y]))/len(a3
)+\
      0.5*reg*(np.sum(np.power(W1,2))+np.sum(np.power
(W2,2)))
    ##################################################
############################
    #                              END OF YOUR CODE
                           #
    ##################################################
############################

    # Backward pass: compute gradients
    grads = {}
    ##################################################
############################
    # TODO: Compute the backward pass, computing the
 derivatives of the weights #
    # and biases. Store the results in the grads dict
ionary. For example,       #
    # grads['W1'] should store the gradient on W1, an
d be a matrix of same size #
    ##################################################
############################
    delta_3=a3
    delta_3[range(len(a3)),y]=a3[range(len(a3)),y]-1
    delta_3/=len(a3)
    grads['W2']=a2.T.dot(delta_3)+reg*W2
    grads['b2']=np.sum(delta_3,0)


    dF=np.ones(np.shape(a2))
    if activation=='leaky':
      dF[a2<0.0]=0.01
    else:
      dF[a2==0.0]=0 #activation res a2 has been ReLUe
d

    delta_2=delta_3.dot(W2.T)*dF
    grads['W1']=X.T.dot(delta_2)+reg*W1
    grads['b1']=np.sum(delta_2,0)
    ##################################################
############################
    #                              END OF YOUR CODE
                           #
```

```
    ################################################
#############################

    return loss, grads

  def train(self, X, y, X_val, y_val,
            learning_rate=1e-3, learning_rate_decay=
0.95,
            reg=1e-5, num_iters=100,
            batch_size=200, verbose=False,

            update="SGD",arg=.99,
            dropout=0,
            activation='ReLU'):
    """
    Train this neural network using stochastic gradie
nt descent.

    Inputs:
    - X: A numpy array of shape (N, D) giving trainin
g data.
    - y: A numpy array f shape (N,) giving training l
abels; y[i] = c means that
      X[i] has label c, where 0 <= c < C.
    - X_val: A numpy array of shape (N_val, D) giving
 validation data.
    - y_val: A numpy array of shape (N_val,) giving v
alidation labels.
    - learning_rate: Scalar giving learning rate for
 optimization.
    - learning_rate_decay: Scalar giving factor used
 to decay the learning rate
      after each epoch.
    - reg: Scalar giving regularization strength.
    - num_iters: Number of steps to take when optimiz
ing.
    - batch_size: Number of training examples to use
 per step.
    - verbose: boolean; if true print progress during
 optimization.
    """
    num_train = X.shape[0]
    iterations_per_epoch = max(num_train / batch_size
, 1)

    # Use SGD to optimize the parameters in self.mode
l
    loss_history = []
    train_acc_history = []
    val_acc_history = []
    #### for tracking top model
    top_params=dict()
    cache_params=dict()
    top_acc=0
    cache=dict()
    dropMask=dict()
    start_time=time.time()
    ####

    for it in xrange(num_iters):
      X_batch = None
      y_batch = None

      ###############################################
###########################
      # TODO: Create a random minibatch of training d
```

```
ata and labels, storing  #
      # them in X_batch and y_batch respectively.
                         #
      ###############################################
##########################
      if num_train >= batch_size:
        rand_idx=np.random.choice(num_train,batch_siz
e)
      else:
        rand_idx=np.random.choice(num_train,batch_siz
e,replace=True)
      X_batch=X[rand_idx]
      y_batch=y[rand_idx]

      if dropout>1:
        for param in ['W2','b2']:
          dropMask[param]=np.random.randn(*self.param
s[param].shape)<(dropout-1)
      ###############################################
##########################
      #                              END OF YOUR CODE
                         #
      ###############################################
##########################

      # Compute loss and gradients using the current
 minibatch
      loss, grads = self.loss(X_batch, y=y_batch, reg
=reg, dropout=dropout,dropMask=dropMask,activation=ac
tivation)
      loss_history.append(loss)

      ###############################################
##########################
      # TODO: Use the gradients in the grads dictiona
ry to update the         #
      # parameters of the network (stored in the dict
ionary self.params)      #
      # using stochastic gradient descent. You'll nee
d to use the gradients   #
      # stored in the grads dictionary defined above.
                         #
      ###############################################
##########################
      if np.isnan(grads['W1']).any() or np.isnan(grad
s['W2']).any() or \
        np.isnan(grads['b1']).any() or np.isnan(grads
['b2']).any():
        continue
     #cache_params=self.params.copy()
      dx=None
      for param in self.params:
        if update=="SGD":
          dx=learning_rate*grads[param]
          #self.params[param]-=learning_rate*grads[pa
ram]

        elif update=="momentum":
          if not param in cache:
            cache[param]=np.zeros(grads[param].shape)
          cache[param]=arg*cache[param]-learning_rate
*grads[param]
          dx=-cache[param]
          #self.params[param]+=cache[param]

        elif update=="Nesterov momentum":
```

```
          if not param in cache:
            cache[param]=np.zeros(grads[param].shape)
          v_prev = cache[param] # back this up
          cache[param] = arg * cache[param] - learnin
g_rate * grads[param] # velocity update stays the sam
e
          dx=arg * v_prev - (1 + arg) * cache[param]
          #self.params[param] += -arg * v_prev + (1 +
 arg) * cache[param] # position update changes form

        elif update=="rmsprop":
          if not param in cache:
            cache[param]=np.zeros(grads[param].shape)
          cache[param]=arg*cache[param]+(1-arg)*np.po
wer(grads[param],2)
          dx=learning_rate*grads[param]/np.sqrt(cache
[param]+1e-8)
          #self.params[param]-=learning_rate*grads[pa
ram]/np.sqrt(cache[param]+1e-8)


        elif update=="Adam":
          print "update error"

        elif update=="Adagrad":
          print "update error"

        else:
          # if have time try more update methods
          print "choose update method!"
        if dropout>1:
          if param == 'W2' or param == 'b2':
            dx*=dropMask[param]
        self.params[param]-=dx
      #Bug: learning rate should not decay at first e
poch
      it+=1
      ###############################################
##########################
      #                              END OF YOUR CODE 
                         #
      ###############################################
##########################

      if verbose and it % 100 == 0:
        print 'iteration %d / %d: loss %f' % (it, num
_iters, loss)

      # Every epoch, check train and val accuracy and
 decay learning rate.
      if it % iterations_per_epoch == 0:
        # Check accuracy
        train_acc = (self.predict(X_batch) == y_batch
).mean()
        val_acc = (self.predict(X_val) == y_val).mean
()
        train_acc_history.append(train_acc)
        val_acc_history.append(val_acc)

        # Decay learning rate
        learning_rate *= learning_rate_decay

        ### update top model
        if val_acc > top_acc:
                top_acc = val_acc
                top_params=self.params.copy()
```

```
        if verbose:
          print ('train_acc %f, val_acc %f, time %d'
% (train_acc, val_acc,(time.time()-start_time)/60.0))

    self.params=top_params.copy()
    ### update params to top params finally

    return {
      'loss_history': loss_history,
      'train_acc_history': train_acc_history,
      'val_acc_history': val_acc_history,
    }

  def predict(self, X):
    """
    Use the trained weights of this two-layer network
 to predict labels for
    data points. For each data point we predict score
s for each of the C
    classes, and assign each data point to the class
 with the highest score.

    Inputs:
    - X: A numpy array of shape (N, D) giving N D-dim
ensional data points to
      classify.

    Returns:
    - y_pred: A numpy array of shape (N,) giving pred
icted labels for each of
      the elements of X. For all i, y_pred[i] = c mea
ns that X[i] is predicted
      to have class c, where 0 <= c < C.
    """
    y_pred = None

    ##################################################
##########################
    # TODO: Implement this function; it should be VER
Y simple!                 #
    ##################################################
##########################
    y_pred=np.argmax(np.maximum(0,(X.dot(self.params[
"W1"])+self.params['b1']))\
             .dot(self.params['W2'])+self.params['b2'
],1) # z3
    ##################################################
##########################
    #                              END OF YOUR CODE
                          #
    ##################################################
##########################

    return y_pred

  def accuracy(self,X,y):
    """
    Use the trained model to predict labels for X, an
d compute the accuracy.

    Inputs:
    - X: A numpy array of shape (N, D) giving N D-dim
ensional data points to
      classify.
    - y: A numpy array of shape (N,) giving the corre
```

```
ct labels.

    Returns:
    - acc: Accuracy

    """
    acc = (self.predict(X) == y).mean()

    return acc

  def gradient_check(self,X,y):
    realGrads=dict()
    _,grads=self.loss(X,y)
    keys=['W1','b1',
          'W2','b2']
    for key in keys:
      W1=self.params[key]
      W1_grad=[]
      delta=1e-4
      if len(np.shape(W1))==2:
        for i in range(np.shape(W1)[0]):
          grad=[]
          for j in range(np.shape(W1)[1]):
            W1[i,j]+=delta
            self.params[key]=W1
            l_plus,_=self.loss(X,y)
            W1[i,j]-=2*delta
            self.params[key]=W1
            l_minus,_=self.loss(X,y)
            grad.append((l_plus-l_minus)/2.0/delta)
            W1[i,j]+=delta
          W1_grad.append(grad)
      else:
        for i in range(len(W1)):
          W1[i]+=delta
          self.params[key]=W1
          l_plus,_=self.loss(X,y)
          W1[i]-=2*delta
          self.params[key]=W1
          l_minus,_=self.loss(X,y)
          W1_grad.append((l_plus-l_minus)/2.0/delta)
          W1[i]+=delta

      print(W1_grad)
      print(grads[key])
      print key,"error",np.mean(np.sum(np.power((W1_g
rad-grads[key]),2),len(np.shape(W1))-1)\
                        /np.sum(np.power((W1_grad+gra
ds[key]),2),len(np.shape(W1))-1))
```

In [ ]:

```
# coding: utf-8

# In[59]:
import sys
from data_utils import load_CIFAR10
from neural_net import *
import matplotlib.pyplot as plt
import time

print "sys.argv : "
print len(sys.argv)

if len(sys.argv)!=1:
    print "something goes wrong,try% python redo.py d
ataset"
```

```
    quit()

dataset_dir = sys.argv[0]
start_time=time.time()

def get_CIFAR10_data(num_training=49000, num_validati
on=1000, num_test=1000):
    cifar10_dir = 'cifar-10-batches-py'
    print cifar10_dir
    X_train, y_train, X_test, y_test = load_CIFAR10(c
ifar10_dir)
    mask = range(num_training, num_training + num_val
idation)
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = range(num_training)
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = range(num_test)
    X_test = X_test[mask]
    y_test = y_test[mask]
    mean_image = np.mean(X_train, axis=0)
    X_train -= mean_image
    X_val -= mean_image
    X_test -= mean_image
    X_train=X_train.swapaxes(1,3)
    X_val=X_val.swapaxes(1,3)
    X_test=X_test.swapaxes(1,3)
    return X_train, y_train, X_val, y_val, X_test, y_
test


# Invoke the above function to get our data.
X_train, y_train, X_val, y_val, X_test, y_test = get_
CIFAR10_data()
print "finish loading"
print 'Train data : ', X_train.shape
print 'Validation data : ', X_val.shape
print 'Test data: ', X_test.shape
print "Time", (time.time()-start_time)/60.0
rfSize = 6
numCentroids=1600
whitening=True
numPatches = 400000
CIFAR_DIM=[32,32,3]

#create unsurpervised data
patches=[]
for i in range(numPatches):
    if(np.mod(i,10000) == 0):
        print "sampling for Kmeans",i,"/",numPatches
    start_r=np.random.randint(CIFAR_DIM[0]-rfSize)
    start_c=np.random.randint(CIFAR_DIM[1]-rfSize)
    patch=np.array([])
    img=X_train[np.mod(i,X_train.shape[0])]
    for layer in img:
        patch=np.append(patch,layer[start_r:start_r+r
fSize].T[start_c:start_c+rfSize].T.ravel())
    patches.append(patch)
patches=np.array(patches)
#normalize patches
patches=(patches-patches.mean(1)[:,None])/np.sqrt(pat
ches.var(1)+10)[:,None]
print "time",(time.time()-start_time)/60.0

# In[66]:
```

```
#for csil
del X_train, y_train, X_val, y_val, X_test, y_test
#whitening
print "whitening"
[D,V]=np.linalg.eig(np.cov(patches,rowvar=0))

P = V.dot(np.diag(np.sqrt(1/(D + 0.1)))).dot(V.T)
patches = patches.dot(P)

print "time",(time.time()-start_time)/60.0
del D,V
# In[ ]:

centroids=np.random.randn(numCentroids,patches.shape[
1])*.1
num_iters=50
batch_size=1000#CSIL do not have enough memory, dam
for ite in range(num_iters):
    print "kmeans iters",ite+1,"/",num_iters
#     c2=.5*np.power(centroids,2).sum(1)
#     idx=np.argmax(patches.dot(centroids.T)-c2,axis=
1) # x2 the same omit
    hf_c2_sum=.5*np.power(centroids,2).sum(1)
    counts=np.zeros(numCentroids)
    summation=np.zeros_like(centroids)
    for i in range(0,len(patches),batch_size):
        last_i=min(i+batch_size,len(patches))
        idx=np.argmax(patches[i:last_i].dot(centroids
.T)                          -hf_c2_sum.T,                          a
xis=1)
        S=np.zeros([last_i-i,numCentroids])
        S[range(last_i-i),
          np.argmax(patches[i:last_i].dot(centroids.T
)-hf_c2_sum.T
                        ,axis=1)]=1
        summation+=S.T.dot(patches[i:last_i])
        counts+=S.sum(0)
    centroids=summation/counts[:,None]
    centroids[counts==0]=0 # some centroids didn't ge
t members, divide by zero
    #the thing is, they will stay zero forever

print "time",(time.time()-start_time)/60.0


# In[82]:

def sliding(img,window=[6,6]):
    out=np.array([])
    for i in range(3):
        s=img.shape
        row=s[1]
        col=s[2]
        col_extent = col - window[1] + 1
        row_extent = row - window[0] + 1
        start_idx = np.arange(window[0])[:,None]*col
+ np.arange(window[1])
        offset_idx = np.arange(row_extent)[:,None]*co
l + np.arange(col_extent)
        if len(out)==0:
            out=np.take (img[i],start_idx.ravel()[:,N
one] + offset_idx.ravel())
        else:
            out=np.append(out,np.take (img[i],start_i
dx.ravel()[:,None] + offset_idx.ravel()),axis=0)
    return out
```

```
# In[111]:

def extract_features(X_train):
    trainXC=[]
    idx=0
    for img in X_train:
        idx+=1
        if not np.mod(idx,1000):
            print "extract features",idx,'/',len(X_tr
ain)
            print "time",(time.time()-start_time)/60.
0
        patches=sliding(img,[rfSize,rfSize]).T
        #normalize
        patches=(patches-patches.mean(1)[:,None])/(np
.sqrt(patches.var(1)+10)[:,None])
        #map to feature space
        patches=patches.dot(P)
        #calculate distance using x2-2xc+c2
        x2=np.power(patches,2).sum(1)
        c2=np.power(centroids,2).sum(1)
        xc=patches.dot(centroids.T)

        dist=np.sqrt(-2*xc+x2[:,None]+c2)
        u=dist.mean(1)
        patches=np.maximum(-dist+u[:,None],0)
        rs=CIFAR_DIM[0]-rfSize+1
        cs=CIFAR_DIM[1]-rfSize+1
        patches=np.reshape(patches,[rs,cs,-1])
        q=[]
        q.append(patches[0:rs/2,0:cs/2].sum(0).sum(0
))
        q.append(patches[0:rs/2,cs/2:cs-1].sum(0).sum
(0))
        q.append(patches[rs/2:rs-1,0:cs/2].sum(0).sum
(0))
        q.append(patches[rs/2:rs-1,cs/2:cs-1].sum(0).
sum(0))
        q=np.array(q).ravel()
        trainXC.append(q)
    trainXC=np.array(trainXC)
    trainXC=(trainXC-trainXC.mean(1)[:,None])/(np.sqr
t(trainXC.var(1)+.01)[:,None])
    return trainXC


X_train, y_train, X_val, y_val, X_test, y_test = get_
CIFAR10_data()
# In[112]:
trainXC=extract_features(X_train)
print "time",(time.time()-start_time)/60.0
valXC=extract_features(X_val)

testXC=extract_features(X_test)


# # save features

# In[131]:

#import cPickle as pickle
#with open("features.pickle","w") as f:
#    pickle.dump([trainXC,valXC,testXC,y_train,y_val,
y_test],f)
```

```
# In[125]:

from neural_net import *

input_size = trainXC.shape[1]
hidden_size = 200
num_classes = 10

net = TwoLayerNet(input_size, hidden_size, num_classe
s,1e-4)
stats = net.train(trainXC, y_train, valXC, y_val,
                            num_iters=70000, batch_si
ze=128,
                            learning_rate=5e-4, learn
ing_rate_decay=0.99,
                            reg=0, verbose=True,updat
e="momentum",arg=0.95,dropout=0.3)


# In[126]:

val_acc = (net.predict(trainXC) == y_train).mean()
print 'Train accuracy: ', val_acc
val_acc = (net.predict(valXC) == y_val).mean()
print 'Validation accuracy: ', val_acc

val_acc = (net.predict(testXC) == y_test).mean()
print 'Test accuracy: ', val_acc

print "time",(time.time()-start_time)/60.0

# In[121]:

##Plot the loss function and train / validation accur
acies
#plt.plot(stats['loss_history'])
#plt.title('Loss history')
#plt.xlabel('Iteration')
#plt.ylabel('Loss')
#plt.show()
##plt.savefig("dropout loss_history.eps")
#
#plt.plot(stats['train_acc_history'], label='train')
#plt.plot(stats['val_acc_history'], label='val')
#plt.title('Classification accuracy history')
#plt.xlabel('Epoch')
#plt.show()
#plt.ylabel('Clasification accuracy')
##plt.savefig('dropout accuracy.eps')


# In[ ]:
```

```
sys.argv :
3
something goes wrong,try% python redo.py dataset
cifar-10-batches-py
this is the  ROOT

cifar-10-batches-py
yeah

this is the file name
```

```
cifar-10-batches-py/data_batch_1
yeah

10000
10000
this is the file name

cifar-10-batches-py/data_batch_2
yeah

10000
10000
this is the file name

cifar-10-batches-py/data_batch_3
yeah

10000
10000
this is the file name

cifar-10-batches-py/data_batch_4
yeah

10000
10000
this is the file name

cifar-10-batches-py/data_batch_5
yeah

10000
10000
this is the file name

cifar-10-batches-py/test_batch
yeah

10000
10000
finish loading
Train data :  (49000, 3, 32, 32)
Validation data :  (1000, 3, 32, 32)
Test data:  (1000, 3, 32, 32)
Time 0.0332777818044
sampling for Kmeans 0 / 400000
sampling for Kmeans 10000 / 400000
sampling for Kmeans 20000 / 400000
sampling for Kmeans 30000 / 400000
sampling for Kmeans 40000 / 400000
sampling for Kmeans 50000 / 400000
sampling for Kmeans 60000 / 400000
sampling for Kmeans 70000 / 400000
sampling for Kmeans 80000 / 400000
sampling for Kmeans 90000 / 400000
sampling for Kmeans 100000 / 400000
sampling for Kmeans 110000 / 400000
sampling for Kmeans 120000 / 400000
sampling for Kmeans 130000 / 400000
sampling for Kmeans 140000 / 400000
sampling for Kmeans 150000 / 400000
sampling for Kmeans 160000 / 400000
sampling for Kmeans 170000 / 400000
sampling for Kmeans 180000 / 400000
sampling for Kmeans 190000 / 400000
sampling for Kmeans 200000 / 400000
```

```
sampling for Kmeans 210000 / 400000
sampling for Kmeans 220000 / 400000
sampling for Kmeans 230000 / 400000
sampling for Kmeans 240000 / 400000
sampling for Kmeans 250000 / 400000
sampling for Kmeans 260000 / 400000
sampling for Kmeans 270000 / 400000
sampling for Kmeans 280000 / 400000
sampling for Kmeans 290000 / 400000
sampling for Kmeans 300000 / 400000
sampling for Kmeans 310000 / 400000
sampling for Kmeans 320000 / 400000
sampling for Kmeans 330000 / 400000
sampling for Kmeans 340000 / 400000
sampling for Kmeans 350000 / 400000
sampling for Kmeans 360000 / 400000
sampling for Kmeans 370000 / 400000
sampling for Kmeans 380000 / 400000
sampling for Kmeans 390000 / 400000
time 0.265322780609
whitening
time 0.280948881308
kmeans iters 1 / 50
kmeans iters 2 / 50
kmeans iters 3 / 50
kmeans iters 4 / 50
kmeans iters 5 / 50
kmeans iters 6 / 50
kmeans iters 7 / 50
kmeans iters 8 / 50
kmeans iters 9 / 50
kmeans iters 10 / 50
kmeans iters 11 / 50
kmeans iters 12 / 50
kmeans iters 13 / 50
kmeans iters 14 / 50
kmeans iters 15 / 50
kmeans iters 16 / 50
kmeans iters 17 / 50
kmeans iters 18 / 50
kmeans iters 19 / 50
kmeans iters 20 / 50
kmeans iters 21 / 50
kmeans iters 22 / 50
kmeans iters 23 / 50
kmeans iters 24 / 50
kmeans iters 25 / 50
kmeans iters 26 / 50
kmeans iters 27 / 50
kmeans iters 28 / 50
kmeans iters 29 / 50
kmeans iters 30 / 50
kmeans iters 31 / 50
kmeans iters 32 / 50
kmeans iters 33 / 50
kmeans iters 34 / 50
kmeans iters 35 / 50
kmeans iters 36 / 50
kmeans iters 37 / 50
kmeans iters 38 / 50
kmeans iters 39 / 50
kmeans iters 40 / 50
kmeans iters 41 / 50
kmeans iters 42 / 50
kmeans iters 43 / 50
kmeans iters 44 / 50
```

```
kmeans iters 45 / 50
kmeans iters 46 / 50
kmeans iters 47 / 50
kmeans iters 48 / 50
kmeans iters 49 / 50
kmeans iters 50 / 50
time 13.8563201666
cifar-10-batches-py
this is the  ROOT

cifar-10-batches-py
yeah

this is the file name

cifar-10-batches-py/data_batch_1
yeah

10000
10000
this is the file name

cifar-10-batches-py/data_batch_2
yeah

10000
10000
this is the file name

cifar-10-batches-py/data_batch_3
yeah

10000
10000
this is the file name

cifar-10-batches-py/data_batch_4
yeah

10000
10000
this is the file name

cifar-10-batches-py/data_batch_5
yeah

10000
10000
this is the file name

cifar-10-batches-py/test_batch
yeah

10000
10000
extract features 1000 / 49000
time 14.4298948646
extract features 2000 / 49000
time 14.9755757491
extract features 3000 / 49000
time 15.5218364636
extract features 4000 / 49000
time 16.0616054138
extract features 5000 / 49000
time 16.5973998666
extract features 6000 / 49000
```

```
time 17.1400368492
extract features 7000 / 49000
time 17.6820870479
extract features 8000 / 49000
time 18.2272395809
extract features 9000 / 49000
time 18.7695302327
extract features 10000 / 49000
time 19.315405798
extract features 11000 / 49000
time 19.8621707479
extract features 12000 / 49000
time 20.4106993
extract features 13000 / 49000
time 20.9563242316
extract features 14000 / 49000
time 21.5043971141
extract features 15000 / 49000
time 22.0550905665
extract features 16000 / 49000
time 22.6078875979
extract features 17000 / 49000
time 23.1564775467
extract features 18000 / 49000
time 23.7031750798
extract features 19000 / 49000
time 24.2530056993
extract features 20000 / 49000
time 24.8020825466
extract features 21000 / 49000
time 25.3450099985
extract features 22000 / 49000
time 25.8889302135
extract features 23000 / 49000
time 26.4369010965
extract features 24000 / 49000
time 26.984493947
extract features 25000 / 49000
time 27.5283052802
extract features 26000 / 49000
time 28.0696772973
extract features 27000 / 49000
time 28.6093527834
extract features 28000 / 49000
time 29.1499058326
extract features 29000 / 49000
time 29.6922206481
extract features 30000 / 49000
time 30.2369059801
extract features 31000 / 49000
time 30.7791065971
extract features 32000 / 49000
time 31.329838864
extract features 33000 / 49000
time 31.876629583
extract features 34000 / 49000
time 32.4268755158
extract features 35000 / 49000
time 32.9795043151
extract features 36000 / 49000
time 33.5289269646
extract features 37000 / 49000
time 34.0789804816
extract features 38000 / 49000
time 34.6270870646
extract features 39000 / 49000
```

```
time 35.1667925318
extract features 40000 / 49000
time 35.7095374465
extract features 41000 / 49000
time 36.2494390806
extract features 42000 / 49000
time 36.7886640986
extract features 43000 / 49000
time 37.3280550639
extract features 44000 / 49000
time 37.8664046129
extract features 45000 / 49000
time 38.4066018144
extract features 46000 / 49000
time 38.9457132657
extract features 47000 / 49000
time 39.4834793131
extract features 48000 / 49000
time 40.0278574824
extract features 49000 / 49000
time 40.5701967478
time 40.6543869495
extract features 1000 / 1000
time 41.1924306989
extract features 1000 / 1000
time 41.7343853315
iteration 100 / 70000: loss 2.302550
iteration 200 / 70000: loss 2.282967
iteration 300 / 70000: loss 2.051948
train_acc 0.226562, val_acc 0.239000, time 0
iteration 400 / 70000: loss 1.900824
iteration 500 / 70000: loss 1.859865
iteration 600 / 70000: loss 1.763703
iteration 700 / 70000: loss 1.740913
train_acc 0.390625, val_acc 0.387000, time 0
iteration 800 / 70000: loss 1.674741
iteration 900 / 70000: loss 1.603166
iteration 1000 / 70000: loss 1.719080
iteration 1100 / 70000: loss 1.648749
train_acc 0.429688, val_acc 0.458000, time 0
iteration 1200 / 70000: loss 1.484712
iteration 1300 / 70000: loss 1.704116
iteration 1400 / 70000: loss 1.556561
iteration 1500 / 70000: loss 1.576333
train_acc 0.515625, val_acc 0.518000, time 1
iteration 1600 / 70000: loss 1.368950
iteration 1700 / 70000: loss 1.463172
iteration 1800 / 70000: loss 1.448691
iteration 1900 / 70000: loss 1.340691
train_acc 0.539062, val_acc 0.551000, time 1
iteration 2000 / 70000: loss 1.315769
iteration 2100 / 70000: loss 1.307350
iteration 2200 / 70000: loss 1.231680
train_acc 0.539062, val_acc 0.558000, time 1
iteration 2300 / 70000: loss 1.312475
iteration 2400 / 70000: loss 1.310738
iteration 2500 / 70000: loss 1.240137
iteration 2600 / 70000: loss 1.119938
train_acc 0.593750, val_acc 0.571000, time 2
iteration 2700 / 70000: loss 1.191043
iteration 2800 / 70000: loss 1.046885
iteration 2900 / 70000: loss 1.346495
iteration 3000 / 70000: loss 1.224660
train_acc 0.601562, val_acc 0.567000, time 2
iteration 3100 / 70000: loss 1.380767
iteration 3200 / 70000: loss 1.266210
```

```
iteration 3300 / 70000: loss 1.083146
iteration 3400 / 70000: loss 1.187526
train_acc 0.632812, val_acc 0.588000, time 2
iteration 3500 / 70000: loss 1.184365
iteration 3600 / 70000: loss 1.222658
iteration 3700 / 70000: loss 1.332623
iteration 3800 / 70000: loss 1.042937
train_acc 0.593750, val_acc 0.599000, time 3
iteration 3900 / 70000: loss 1.326720
iteration 4000 / 70000: loss 1.092221
iteration 4100 / 70000: loss 1.225842
iteration 4200 / 70000: loss 1.113048
train_acc 0.570312, val_acc 0.609000, time 3
iteration 4300 / 70000: loss 1.229283
iteration 4400 / 70000: loss 1.204325
iteration 4500 / 70000: loss 1.177719
train_acc 0.593750, val_acc 0.613000, time 3
iteration 4600 / 70000: loss 1.251272
iteration 4700 / 70000: loss 0.966569
iteration 4800 / 70000: loss 1.055023
iteration 4900 / 70000: loss 1.173206
train_acc 0.718750, val_acc 0.615000, time 4
iteration 5000 / 70000: loss 1.256344
iteration 5100 / 70000: loss 1.122686
iteration 5200 / 70000: loss 1.074769
iteration 5300 / 70000: loss 1.116009
train_acc 0.640625, val_acc 0.629000, time 4
iteration 5400 / 70000: loss 0.977122
iteration 5500 / 70000: loss 0.978666
iteration 5600 / 70000: loss 1.093144
iteration 5700 / 70000: loss 1.074519
train_acc 0.648438, val_acc 0.613000, time 4
iteration 5800 / 70000: loss 0.989326
iteration 5900 / 70000: loss 1.044299
iteration 6000 / 70000: loss 1.102523
iteration 6100 / 70000: loss 1.243159
train_acc 0.718750, val_acc 0.627000, time 5
iteration 6200 / 70000: loss 1.186926
iteration 6300 / 70000: loss 1.181103
iteration 6400 / 70000: loss 1.153347
train_acc 0.640625, val_acc 0.617000, time 5
iteration 6500 / 70000: loss 0.999560
iteration 6600 / 70000: loss 1.165058
iteration 6700 / 70000: loss 1.036365
iteration 6800 / 70000: loss 1.067316
train_acc 0.601562, val_acc 0.640000, time 5
iteration 6900 / 70000: loss 1.048155
iteration 7000 / 70000: loss 1.190646
iteration 7100 / 70000: loss 1.161766
iteration 7200 / 70000: loss 1.006866
train_acc 0.695312, val_acc 0.633000, time 6
iteration 7300 / 70000: loss 0.972489
iteration 7400 / 70000: loss 1.049888
iteration 7500 / 70000: loss 1.081532
iteration 7600 / 70000: loss 1.108342
train_acc 0.757812, val_acc 0.654000, time 6
iteration 7700 / 70000: loss 1.066823
iteration 7800 / 70000: loss 0.968113
iteration 7900 / 70000: loss 0.835822
iteration 8000 / 70000: loss 1.031501
train_acc 0.703125, val_acc 0.639000, time 6
iteration 8100 / 70000: loss 1.138332
iteration 8200 / 70000: loss 0.996870
iteration 8300 / 70000: loss 1.156585
iteration 8400 / 70000: loss 1.064495
train_acc 0.656250, val_acc 0.644000, time 7
```

```
iteration 8500 / 70000: loss 0.998167
iteration 8600 / 70000: loss 1.005703
iteration 8700 / 70000: loss 1.037420
train_acc 0.664062, val_acc 0.644000, time 7
iteration 8800 / 70000: loss 1.090727
iteration 8900 / 70000: loss 0.983271
iteration 9000 / 70000: loss 1.058497
iteration 9100 / 70000: loss 1.122564
train_acc 0.578125, val_acc 0.659000, time 7
iteration 9200 / 70000: loss 1.077176
iteration 9300 / 70000: loss 0.956432
iteration 9400 / 70000: loss 0.927782
iteration 9500 / 70000: loss 1.140321
train_acc 0.648438, val_acc 0.641000, time 8
iteration 9600 / 70000: loss 0.984117
iteration 9700 / 70000: loss 1.019935
iteration 9800 / 70000: loss 0.910029
iteration 9900 / 70000: loss 1.118935
train_acc 0.695312, val_acc 0.660000, time 8
iteration 10000 / 70000: loss 0.924132
iteration 10100 / 70000: loss 1.096376
iteration 10200 / 70000: loss 0.896235
iteration 10300 / 70000: loss 0.978672
train_acc 0.671875, val_acc 0.672000, time 8
iteration 10400 / 70000: loss 0.877650
iteration 10500 / 70000: loss 0.906416
iteration 10600 / 70000: loss 1.047400
train_acc 0.656250, val_acc 0.662000, time 9
iteration 10700 / 70000: loss 0.950416
iteration 10800 / 70000: loss 0.788344
iteration 10900 / 70000: loss 1.082032
iteration 11000 / 70000: loss 1.059189
train_acc 0.664062, val_acc 0.663000, time 9
iteration 11100 / 70000: loss 0.959533
iteration 11200 / 70000: loss 0.906568
iteration 11300 / 70000: loss 0.801165
iteration 11400 / 70000: loss 1.089156
train_acc 0.742188, val_acc 0.664000, time 9
iteration 11500 / 70000: loss 0.948434
iteration 11600 / 70000: loss 1.019118
iteration 11700 / 70000: loss 0.883635
iteration 11800 / 70000: loss 0.941161
train_acc 0.703125, val_acc 0.666000, time 10
iteration 11900 / 70000: loss 1.025876
iteration 12000 / 70000: loss 1.077164
iteration 12100 / 70000: loss 0.958841
iteration 12200 / 70000: loss 1.073617
train_acc 0.679688, val_acc 0.656000, time 10
iteration 12300 / 70000: loss 0.796467
iteration 12400 / 70000: loss 0.704608
iteration 12500 / 70000: loss 1.261353
iteration 12600 / 70000: loss 0.982834
train_acc 0.640625, val_acc 0.655000, time 10
iteration 12700 / 70000: loss 0.885424
iteration 12800 / 70000: loss 0.872497
iteration 12900 / 70000: loss 1.003111
train_acc 0.617188, val_acc 0.678000, time 11
iteration 13000 / 70000: loss 0.848834
iteration 13100 / 70000: loss 0.961732
iteration 13200 / 70000: loss 0.977405
iteration 13300 / 70000: loss 1.063359
train_acc 0.687500, val_acc 0.660000, time 11
iteration 13400 / 70000: loss 0.881917
iteration 13500 / 70000: loss 1.082812
iteration 13600 / 70000: loss 0.910345
iteration 13700 / 70000: loss 0.955019
```

```
train_acc 0.742188, val_acc 0.657000, time 11
iteration 13800 / 70000: loss 0.869089
iteration 13900 / 70000: loss 0.957771
iteration 14000 / 70000: loss 0.997027
iteration 14100 / 70000: loss 0.951177
train_acc 0.726562, val_acc 0.685000, time 12
iteration 14200 / 70000: loss 0.977152
iteration 14300 / 70000: loss 0.879129
iteration 14400 / 70000: loss 0.816919
iteration 14500 / 70000: loss 0.927724
train_acc 0.718750, val_acc 0.680000, time 12
iteration 14600 / 70000: loss 0.880810
iteration 14700 / 70000: loss 1.024604
iteration 14800 / 70000: loss 0.943535
train_acc 0.648438, val_acc 0.670000, time 12
iteration 14900 / 70000: loss 1.034858
iteration 15000 / 70000: loss 0.948886
iteration 15100 / 70000: loss 0.914209
iteration 15200 / 70000: loss 0.785248
train_acc 0.812500, val_acc 0.691000, time 13
iteration 15300 / 70000: loss 0.779184
iteration 15400 / 70000: loss 0.853494
iteration 15500 / 70000: loss 0.846600
iteration 15600 / 70000: loss 0.979597
train_acc 0.757812, val_acc 0.682000, time 13
iteration 15700 / 70000: loss 0.967526
iteration 15800 / 70000: loss 0.909604
iteration 15900 / 70000: loss 1.046342
iteration 16000 / 70000: loss 0.936757
train_acc 0.718750, val_acc 0.698000, time 13
iteration 16100 / 70000: loss 1.039720
iteration 16200 / 70000: loss 0.904597
iteration 16300 / 70000: loss 0.947767
iteration 16400 / 70000: loss 0.978088
train_acc 0.679688, val_acc 0.688000, time 14
iteration 16500 / 70000: loss 0.844755
iteration 16600 / 70000: loss 1.042896
iteration 16700 / 70000: loss 1.025411
iteration 16800 / 70000: loss 1.023685
train_acc 0.734375, val_acc 0.682000, time 14
iteration 16900 / 70000: loss 1.018551
iteration 17000 / 70000: loss 1.091056
iteration 17100 / 70000: loss 0.942478
train_acc 0.765625, val_acc 0.682000, time 14
iteration 17200 / 70000: loss 0.978816
iteration 17300 / 70000: loss 1.069868
iteration 17400 / 70000: loss 0.869582
iteration 17500 / 70000: loss 0.840120
train_acc 0.742188, val_acc 0.690000, time 15
iteration 17600 / 70000: loss 0.926371
iteration 17700 / 70000: loss 0.820555
iteration 17800 / 70000: loss 0.888810
iteration 17900 / 70000: loss 0.844805
train_acc 0.671875, val_acc 0.693000, time 15
iteration 18000 / 70000: loss 0.986059
iteration 18100 / 70000: loss 1.014061
iteration 18200 / 70000: loss 0.983496
iteration 18300 / 70000: loss 0.967919
train_acc 0.710938, val_acc 0.679000, time 15
iteration 18400 / 70000: loss 1.003831
iteration 18500 / 70000: loss 0.715973
iteration 18600 / 70000: loss 0.928987
iteration 18700 / 70000: loss 0.942465
train_acc 0.750000, val_acc 0.689000, time 16
iteration 18800 / 70000: loss 0.905443
iteration 18900 / 70000: loss 0.935337
```

```
iteration 19000 / 70000: loss 0.888900
iteration 19100 / 70000: loss 0.898285
train_acc 0.703125, val_acc 0.699000, time 16
iteration 19200 / 70000: loss 0.868214
iteration 19300 / 70000: loss 0.850434
iteration 19400 / 70000: loss 0.801074
train_acc 0.617188, val_acc 0.698000, time 16
iteration 19500 / 70000: loss 1.075964
iteration 19600 / 70000: loss 0.886457
iteration 19700 / 70000: loss 0.879318
iteration 19800 / 70000: loss 0.876255
train_acc 0.734375, val_acc 0.685000, time 17
iteration 19900 / 70000: loss 0.804190
iteration 20000 / 70000: loss 0.869415
iteration 20100 / 70000: loss 1.042462
iteration 20200 / 70000: loss 0.800603
train_acc 0.750000, val_acc 0.684000, time 17
iteration 20300 / 70000: loss 0.991945
iteration 20400 / 70000: loss 0.954093
iteration 20500 / 70000: loss 0.955457
iteration 20600 / 70000: loss 0.873104
train_acc 0.734375, val_acc 0.699000, time 17
iteration 20700 / 70000: loss 0.897081
iteration 20800 / 70000: loss 0.870058
iteration 20900 / 70000: loss 1.026918
iteration 21000 / 70000: loss 0.866257
train_acc 0.742188, val_acc 0.691000, time 18
iteration 21100 / 70000: loss 0.990298
iteration 21200 / 70000: loss 0.904502
iteration 21300 / 70000: loss 0.885065
train_acc 0.757812, val_acc 0.714000, time 18
iteration 21400 / 70000: loss 0.781013
iteration 21500 / 70000: loss 0.778841
iteration 21600 / 70000: loss 1.046646
iteration 21700 / 70000: loss 0.925963
train_acc 0.734375, val_acc 0.701000, time 18
iteration 21800 / 70000: loss 0.784208
iteration 21900 / 70000: loss 0.807382
iteration 22000 / 70000: loss 0.862831
iteration 22100 / 70000: loss 0.764200
train_acc 0.703125, val_acc 0.704000, time 19
iteration 22200 / 70000: loss 0.827520
iteration 22300 / 70000: loss 1.014624
iteration 22400 / 70000: loss 0.722039
iteration 22500 / 70000: loss 0.780349
train_acc 0.695312, val_acc 0.701000, time 19
iteration 22600 / 70000: loss 0.860506
iteration 22700 / 70000: loss 0.968060
iteration 22800 / 70000: loss 0.785601
iteration 22900 / 70000: loss 0.832004
train_acc 0.695312, val_acc 0.694000, time 19
iteration 23000 / 70000: loss 0.795821
iteration 23100 / 70000: loss 0.758982
iteration 23200 / 70000: loss 0.794293
iteration 23300 / 70000: loss 0.785394
train_acc 0.726562, val_acc 0.699000, time 20
iteration 23400 / 70000: loss 0.867627
iteration 23500 / 70000: loss 0.749242
iteration 23600 / 70000: loss 0.949701
train_acc 0.750000, val_acc 0.694000, time 20
iteration 23700 / 70000: loss 0.852784
iteration 23800 / 70000: loss 0.800460
iteration 23900 / 70000: loss 0.909474
iteration 24000 / 70000: loss 0.982112
train_acc 0.734375, val_acc 0.704000, time 20
iteration 24100 / 70000: loss 0.774726
```

```
iteration 24200 / 70000: loss 0.784734
iteration 24300 / 70000: loss 0.885008
iteration 24400 / 70000: loss 1.070238
train_acc 0.726562, val_acc 0.717000, time 21
iteration 24500 / 70000: loss 0.699356
iteration 24600 / 70000: loss 0.765832
iteration 24700 / 70000: loss 0.893971
iteration 24800 / 70000: loss 0.844013
train_acc 0.695312, val_acc 0.708000, time 21
iteration 24900 / 70000: loss 0.834013
iteration 25000 / 70000: loss 0.873120
iteration 25100 / 70000: loss 0.827360
iteration 25200 / 70000: loss 0.977934
train_acc 0.789062, val_acc 0.699000, time 21
iteration 25300 / 70000: loss 0.900819
iteration 25400 / 70000: loss 0.756501
iteration 25500 / 70000: loss 0.820809
train_acc 0.718750, val_acc 0.704000, time 22
iteration 25600 / 70000: loss 0.937117
iteration 25700 / 70000: loss 0.849396
iteration 25800 / 70000: loss 1.003035
iteration 25900 / 70000: loss 0.859857
train_acc 0.734375, val_acc 0.703000, time 22
iteration 26000 / 70000: loss 0.801117
iteration 26100 / 70000: loss 0.926082
iteration 26200 / 70000: loss 0.835418
iteration 26300 / 70000: loss 0.850389
train_acc 0.804688, val_acc 0.718000, time 22
iteration 26400 / 70000: loss 1.019281
iteration 26500 / 70000: loss 0.810387
iteration 26600 / 70000: loss 0.769914
iteration 26700 / 70000: loss 0.823127
train_acc 0.765625, val_acc 0.713000, time 23
iteration 26800 / 70000: loss 0.987151
iteration 26900 / 70000: loss 1.021286
iteration 27000 / 70000: loss 0.817133
iteration 27100 / 70000: loss 0.925809
train_acc 0.726562, val_acc 0.699000, time 23
iteration 27200 / 70000: loss 0.740572
iteration 27300 / 70000: loss 0.745062
iteration 27400 / 70000: loss 0.833314
iteration 27500 / 70000: loss 0.825810
train_acc 0.781250, val_acc 0.710000, time 23
iteration 27600 / 70000: loss 0.774725
iteration 27700 / 70000: loss 0.828279
iteration 27800 / 70000: loss 0.972994
train_acc 0.734375, val_acc 0.712000, time 24
iteration 27900 / 70000: loss 0.804038
iteration 28000 / 70000: loss 1.074641
iteration 28100 / 70000: loss 0.812473
iteration 28200 / 70000: loss 0.937987
train_acc 0.750000, val_acc 0.693000, time 24
iteration 28300 / 70000: loss 0.992963
iteration 28400 / 70000: loss 0.865925
iteration 28500 / 70000: loss 0.877349
iteration 28600 / 70000: loss 0.638761
train_acc 0.796875, val_acc 0.694000, time 24
iteration 28700 / 70000: loss 0.799846
iteration 28800 / 70000: loss 0.883155
iteration 28900 / 70000: loss 0.968313
iteration 29000 / 70000: loss 0.947478
train_acc 0.796875, val_acc 0.718000, time 25
iteration 29100 / 70000: loss 0.894138
iteration 29200 / 70000: loss 0.746584
iteration 29300 / 70000: loss 0.910179
iteration 29400 / 70000: loss 0.785871
```

```
train_acc 0.734375, val_acc 0.715000, time 25
iteration 29500 / 70000: loss 0.961841
iteration 29600 / 70000: loss 0.948080
iteration 29700 / 70000: loss 0.779055
train_acc 0.828125, val_acc 0.718000, time 25
iteration 29800 / 70000: loss 1.000345
iteration 29900 / 70000: loss 1.022735
iteration 30000 / 70000: loss 0.712108
iteration 30100 / 70000: loss 0.770206
train_acc 0.664062, val_acc 0.719000, time 26
iteration 30200 / 70000: loss 0.835011
iteration 30300 / 70000: loss 0.856460
iteration 30400 / 70000: loss 0.654347
iteration 30500 / 70000: loss 0.835493
train_acc 0.828125, val_acc 0.717000, time 26
iteration 30600 / 70000: loss 0.767045
iteration 30700 / 70000: loss 0.835004
iteration 30800 / 70000: loss 0.952454
iteration 30900 / 70000: loss 0.864302
train_acc 0.710938, val_acc 0.719000, time 27
iteration 31000 / 70000: loss 0.867636
iteration 31100 / 70000: loss 0.694161
iteration 31200 / 70000: loss 0.964958
iteration 31300 / 70000: loss 0.821993
train_acc 0.796875, val_acc 0.728000, time 27
iteration 31400 / 70000: loss 0.846875
iteration 31500 / 70000: loss 0.844936
iteration 31600 / 70000: loss 0.805952
iteration 31700 / 70000: loss 0.869807
train_acc 0.757812, val_acc 0.714000, time 27
iteration 31800 / 70000: loss 0.738874
iteration 31900 / 70000: loss 0.908889
iteration 32000 / 70000: loss 0.745145
train_acc 0.789062, val_acc 0.714000, time 28
iteration 32100 / 70000: loss 0.871979
iteration 32200 / 70000: loss 0.837833
iteration 32300 / 70000: loss 0.688405
iteration 32400 / 70000: loss 0.745517
train_acc 0.773438, val_acc 0.709000, time 28
iteration 32500 / 70000: loss 0.719144
iteration 32600 / 70000: loss 0.850455
iteration 32700 / 70000: loss 0.718693
iteration 32800 / 70000: loss 0.917279
train_acc 0.789062, val_acc 0.714000, time 28
iteration 32900 / 70000: loss 0.956490
iteration 33000 / 70000: loss 0.875181
iteration 33100 / 70000: loss 0.722028
iteration 33200 / 70000: loss 0.805254
train_acc 0.703125, val_acc 0.713000, time 29
iteration 33300 / 70000: loss 0.840509
iteration 33400 / 70000: loss 0.839951
iteration 33500 / 70000: loss 0.774387
iteration 33600 / 70000: loss 0.904148
train_acc 0.781250, val_acc 0.717000, time 29
iteration 33700 / 70000: loss 0.839938
iteration 33800 / 70000: loss 0.915176
iteration 33900 / 70000: loss 0.780377
train_acc 0.750000, val_acc 0.719000, time 29
iteration 34000 / 70000: loss 0.931732
iteration 34100 / 70000: loss 0.666326
iteration 34200 / 70000: loss 0.844061
iteration 34300 / 70000: loss 0.747812
train_acc 0.687500, val_acc 0.725000, time 30
iteration 34400 / 70000: loss 0.863707
iteration 34500 / 70000: loss 0.828443
iteration 34600 / 70000: loss 0.753217
```

```
iteration 34700 / 70000: loss 0.862173
train_acc 0.859375, val_acc 0.709000, time 30
iteration 34800 / 70000: loss 0.805796
iteration 34900 / 70000: loss 0.888405
iteration 35000 / 70000: loss 0.825887
iteration 35100 / 70000: loss 0.850519
train_acc 0.789062, val_acc 0.717000, time 30
iteration 35200 / 70000: loss 0.745485
iteration 35300 / 70000: loss 0.783295
iteration 35400 / 70000: loss 0.903746
iteration 35500 / 70000: loss 0.824758
train_acc 0.750000, val_acc 0.711000, time 31
iteration 35600 / 70000: loss 0.765938
iteration 35700 / 70000: loss 0.843912
iteration 35800 / 70000: loss 0.753152
iteration 35900 / 70000: loss 0.864857
train_acc 0.820312, val_acc 0.722000, time 31
iteration 36000 / 70000: loss 0.725687
iteration 36100 / 70000: loss 0.778369
iteration 36200 / 70000: loss 0.621626
train_acc 0.734375, val_acc 0.706000, time 31
iteration 36300 / 70000: loss 0.877340
iteration 36400 / 70000: loss 0.726968
iteration 36500 / 70000: loss 0.855074
iteration 36600 / 70000: loss 0.726135
train_acc 0.742188, val_acc 0.720000, time 32
iteration 36700 / 70000: loss 0.929854
iteration 36800 / 70000: loss 0.754141
iteration 36900 / 70000: loss 0.757374
iteration 37000 / 70000: loss 0.767275
train_acc 0.718750, val_acc 0.712000, time 32
iteration 37100 / 70000: loss 0.706232
iteration 37200 / 70000: loss 0.638257
iteration 37300 / 70000: loss 0.681856
iteration 37400 / 70000: loss 0.864550
train_acc 0.789062, val_acc 0.729000, time 32
iteration 37500 / 70000: loss 0.981321
iteration 37600 / 70000: loss 0.954613
iteration 37700 / 70000: loss 0.935690
iteration 37800 / 70000: loss 0.784394
train_acc 0.750000, val_acc 0.724000, time 33
iteration 37900 / 70000: loss 0.755393
iteration 38000 / 70000: loss 0.797677
iteration 38100 / 70000: loss 0.790261
iteration 38200 / 70000: loss 0.792092
train_acc 0.765625, val_acc 0.722000, time 33
iteration 38300 / 70000: loss 0.936260
iteration 38400 / 70000: loss 0.780488
iteration 38500 / 70000: loss 0.818089
train_acc 0.781250, val_acc 0.722000, time 33
iteration 38600 / 70000: loss 0.785412
iteration 38700 / 70000: loss 0.675502
iteration 38800 / 70000: loss 0.696695
iteration 38900 / 70000: loss 0.910499
train_acc 0.703125, val_acc 0.719000, time 34
iteration 39000 / 70000: loss 0.991065
iteration 39100 / 70000: loss 0.858246
iteration 39200 / 70000: loss 0.776260
iteration 39300 / 70000: loss 0.809989
train_acc 0.632812, val_acc 0.726000, time 34
iteration 39400 / 70000: loss 0.971220
iteration 39500 / 70000: loss 0.706397
iteration 39600 / 70000: loss 0.753891
iteration 39700 / 70000: loss 0.826238
train_acc 0.750000, val_acc 0.718000, time 34
iteration 39800 / 70000: loss 0.792408
```

```
iteration 39900 / 70000: loss 0.870086
iteration 40000 / 70000: loss 0.852051
iteration 40100 / 70000: loss 0.744788
train_acc 0.750000, val_acc 0.728000, time 35
iteration 40200 / 70000: loss 0.686981
iteration 40300 / 70000: loss 0.849556
iteration 40400 / 70000: loss 0.978500
train_acc 0.734375, val_acc 0.730000, time 35
iteration 40500 / 70000: loss 0.649984
iteration 40600 / 70000: loss 0.701581
iteration 40700 / 70000: loss 0.803117
iteration 40800 / 70000: loss 0.764066
train_acc 0.765625, val_acc 0.730000, time 35
iteration 40900 / 70000: loss 0.800538
iteration 41000 / 70000: loss 0.717815
iteration 41100 / 70000: loss 0.767355
iteration 41200 / 70000: loss 0.761377
train_acc 0.773438, val_acc 0.714000, time 36
iteration 41300 / 70000: loss 0.813761
iteration 41400 / 70000: loss 0.685020
iteration 41500 / 70000: loss 0.777245
iteration 41600 / 70000: loss 0.697764
train_acc 0.828125, val_acc 0.726000, time 36
iteration 41700 / 70000: loss 0.694018
iteration 41800 / 70000: loss 0.824895
iteration 41900 / 70000: loss 0.703806
iteration 42000 / 70000: loss 0.780892
train_acc 0.796875, val_acc 0.721000, time 36
iteration 42100 / 70000: loss 0.740003
iteration 42200 / 70000: loss 0.682604
iteration 42300 / 70000: loss 0.815966
iteration 42400 / 70000: loss 0.853186
train_acc 0.781250, val_acc 0.729000, time 37
iteration 42500 / 70000: loss 0.776944
iteration 42600 / 70000: loss 0.978213
iteration 42700 / 70000: loss 0.857898
train_acc 0.757812, val_acc 0.722000, time 37
iteration 42800 / 70000: loss 0.709963
iteration 42900 / 70000: loss 0.724592
iteration 43000 / 70000: loss 0.714472
iteration 43100 / 70000: loss 0.788598
train_acc 0.742188, val_acc 0.731000, time 37
iteration 43200 / 70000: loss 0.793900
iteration 43300 / 70000: loss 0.732274
iteration 43400 / 70000: loss 0.730673
iteration 43500 / 70000: loss 0.755303
train_acc 0.734375, val_acc 0.710000, time 38
iteration 43600 / 70000: loss 0.835932
iteration 43700 / 70000: loss 0.832261
iteration 43800 / 70000: loss 0.635715
iteration 43900 / 70000: loss 0.871151
train_acc 0.765625, val_acc 0.724000, time 38
iteration 44000 / 70000: loss 0.685330
iteration 44100 / 70000: loss 0.746036
iteration 44200 / 70000: loss 0.761495
iteration 44300 / 70000: loss 0.778861
train_acc 0.796875, val_acc 0.734000, time 38
iteration 44400 / 70000: loss 0.770768
iteration 44500 / 70000: loss 0.815129
iteration 44600 / 70000: loss 0.706186
train_acc 0.781250, val_acc 0.721000, time 39
iteration 44700 / 70000: loss 0.777017
iteration 44800 / 70000: loss 0.703979
iteration 44900 / 70000: loss 0.614390
iteration 45000 / 70000: loss 0.875043
train_acc 0.765625, val_acc 0.727000, time 39
```

```
iteration 45100 / 70000: loss 0.816297
iteration 45200 / 70000: loss 0.683459
iteration 45300 / 70000: loss 0.811980
iteration 45400 / 70000: loss 0.717073
train_acc 0.789062, val_acc 0.730000, time 39
iteration 45500 / 70000: loss 0.646271
iteration 45600 / 70000: loss 0.823140
iteration 45700 / 70000: loss 0.677951
iteration 45800 / 70000: loss 0.559351
train_acc 0.742188, val_acc 0.731000, time 40
iteration 45900 / 70000: loss 0.733061
iteration 46000 / 70000: loss 0.765484
iteration 46100 / 70000: loss 0.742641
iteration 46200 / 70000: loss 0.741235
train_acc 0.804688, val_acc 0.733000, time 40
iteration 46300 / 70000: loss 0.756821
iteration 46400 / 70000: loss 0.744831
iteration 46500 / 70000: loss 0.689911
iteration 46600 / 70000: loss 0.665936
train_acc 0.781250, val_acc 0.738000, time 40
iteration 46700 / 70000: loss 0.875159
iteration 46800 / 70000: loss 0.765961
iteration 46900 / 70000: loss 0.767675
train_acc 0.835938, val_acc 0.734000, time 41
iteration 47000 / 70000: loss 0.660496
iteration 47100 / 70000: loss 0.755461
iteration 47200 / 70000: loss 0.752207
iteration 47300 / 70000: loss 0.779191
train_acc 0.742188, val_acc 0.725000, time 41
iteration 47400 / 70000: loss 0.668950
iteration 47500 / 70000: loss 0.740402
iteration 47600 / 70000: loss 0.722498
iteration 47700 / 70000: loss 0.798509
train_acc 0.773438, val_acc 0.729000, time 41
iteration 47800 / 70000: loss 0.965318
iteration 47900 / 70000: loss 0.801341
iteration 48000 / 70000: loss 0.625792
iteration 48100 / 70000: loss 0.861843
train_acc 0.742188, val_acc 0.730000, time 42
iteration 48200 / 70000: loss 0.576607
iteration 48300 / 70000: loss 0.870848
iteration 48400 / 70000: loss 0.686437
iteration 48500 / 70000: loss 0.786754
train_acc 0.765625, val_acc 0.734000, time 42
iteration 48600 / 70000: loss 0.803925
iteration 48700 / 70000: loss 0.821171
iteration 48800 / 70000: loss 0.662968
train_acc 0.765625, val_acc 0.734000, time 42
iteration 48900 / 70000: loss 0.685506
iteration 49000 / 70000: loss 0.675705
iteration 49100 / 70000: loss 0.727187
iteration 49200 / 70000: loss 0.739175
train_acc 0.820312, val_acc 0.737000, time 43
iteration 49300 / 70000: loss 0.876014
iteration 49400 / 70000: loss 0.721119
iteration 49500 / 70000: loss 0.687557
iteration 49600 / 70000: loss 0.851873
train_acc 0.765625, val_acc 0.728000, time 43
iteration 49700 / 70000: loss 0.798397
iteration 49800 / 70000: loss 0.685073
iteration 49900 / 70000: loss 0.677490
iteration 50000 / 70000: loss 0.724173
train_acc 0.734375, val_acc 0.727000, time 43
iteration 50100 / 70000: loss 0.611368
iteration 50200 / 70000: loss 0.868504
iteration 50300 / 70000: loss 0.808371
```

```
iteration 50400 / 70000: loss 0.624997
train_acc 0.804688, val_acc 0.735000, time 44
iteration 50500 / 70000: loss 0.586096
iteration 50600 / 70000: loss 0.836513
iteration 50700 / 70000: loss 0.777719
iteration 50800 / 70000: loss 0.686615
train_acc 0.781250, val_acc 0.740000, time 44
iteration 50900 / 70000: loss 0.775559
iteration 51000 / 70000: loss 0.655367
iteration 51100 / 70000: loss 0.851583
train_acc 0.812500, val_acc 0.732000, time 44
iteration 51200 / 70000: loss 0.606910
iteration 51300 / 70000: loss 0.693667
iteration 51400 / 70000: loss 0.729012
iteration 51500 / 70000: loss 0.729488
train_acc 0.859375, val_acc 0.733000, time 45
iteration 51600 / 70000: loss 0.635499
iteration 51700 / 70000: loss 0.847504
iteration 51800 / 70000: loss 0.801913
iteration 51900 / 70000: loss 0.664114
train_acc 0.773438, val_acc 0.739000, time 45
iteration 52000 / 70000: loss 0.662015
iteration 52100 / 70000: loss 0.773299
iteration 52200 / 70000: loss 0.893592
iteration 52300 / 70000: loss 0.887781
train_acc 0.750000, val_acc 0.725000, time 45
iteration 52400 / 70000: loss 0.654039
iteration 52500 / 70000: loss 0.649595
iteration 52600 / 70000: loss 0.898376
iteration 52700 / 70000: loss 0.588097
train_acc 0.804688, val_acc 0.739000, time 46
iteration 52800 / 70000: loss 0.643437
iteration 52900 / 70000: loss 0.751301
iteration 53000 / 70000: loss 0.682358
train_acc 0.789062, val_acc 0.729000, time 46
iteration 53100 / 70000: loss 0.598263
iteration 53200 / 70000: loss 0.666399
iteration 53300 / 70000: loss 0.701247
iteration 53400 / 70000: loss 0.723430
train_acc 0.804688, val_acc 0.730000, time 46
iteration 53500 / 70000: loss 0.779279
iteration 53600 / 70000: loss 0.756737
iteration 53700 / 70000: loss 0.745897
iteration 53800 / 70000: loss 0.668887
train_acc 0.781250, val_acc 0.742000, time 47
iteration 53900 / 70000: loss 0.739724
iteration 54000 / 70000: loss 0.771017
iteration 54100 / 70000: loss 0.697399
iteration 54200 / 70000: loss 0.621473
train_acc 0.859375, val_acc 0.730000, time 47
iteration 54300 / 70000: loss 0.708740
iteration 54400 / 70000: loss 0.777118
iteration 54500 / 70000: loss 0.625302
iteration 54600 / 70000: loss 0.670845
train_acc 0.812500, val_acc 0.735000, time 47
iteration 54700 / 70000: loss 0.701203
iteration 54800 / 70000: loss 0.633896
iteration 54900 / 70000: loss 0.736933
iteration 55000 / 70000: loss 0.773894
train_acc 0.750000, val_acc 0.743000, time 48
iteration 55100 / 70000: loss 0.805963
iteration 55200 / 70000: loss 0.680839
iteration 55300 / 70000: loss 0.738048
train_acc 0.726562, val_acc 0.738000, time 48
iteration 55400 / 70000: loss 0.616559
iteration 55500 / 70000: loss 0.577614
```

```
iteration 55600 / 70000: loss 0.718908
iteration 55700 / 70000: loss 0.616026
train_acc 0.796875, val_acc 0.737000, time 48
iteration 55800 / 70000: loss 0.735265
iteration 55900 / 70000: loss 0.633102
iteration 56000 / 70000: loss 0.797398
iteration 56100 / 70000: loss 0.735551
train_acc 0.796875, val_acc 0.730000, time 49
iteration 56200 / 70000: loss 0.683261
iteration 56300 / 70000: loss 0.830656
iteration 56400 / 70000: loss 0.844566
iteration 56500 / 70000: loss 0.789698
train_acc 0.781250, val_acc 0.742000, time 49
iteration 56600 / 70000: loss 0.687998
iteration 56700 / 70000: loss 0.770270
iteration 56800 / 70000: loss 0.838665
iteration 56900 / 70000: loss 0.800033
train_acc 0.820312, val_acc 0.733000, time 49
iteration 57000 / 70000: loss 0.581969
iteration 57100 / 70000: loss 0.918380
iteration 57200 / 70000: loss 0.804145
iteration 57300 / 70000: loss 0.726662
train_acc 0.750000, val_acc 0.740000, time 50
iteration 57400 / 70000: loss 0.827190
iteration 57500 / 70000: loss 0.809071
iteration 57600 / 70000: loss 0.745621
train_acc 0.820312, val_acc 0.730000, time 50
iteration 57700 / 70000: loss 0.641640
iteration 57800 / 70000: loss 0.757706
iteration 57900 / 70000: loss 0.893384
iteration 58000 / 70000: loss 0.730423
train_acc 0.765625, val_acc 0.728000, time 50
iteration 58100 / 70000: loss 0.668006
iteration 58200 / 70000: loss 0.967094
iteration 58300 / 70000: loss 0.612156
iteration 58400 / 70000: loss 0.687895
train_acc 0.757812, val_acc 0.729000, time 51
iteration 58500 / 70000: loss 0.641489
iteration 58600 / 70000: loss 0.784849
iteration 58700 / 70000: loss 0.730490
iteration 58800 / 70000: loss 0.617149
train_acc 0.773438, val_acc 0.744000, time 51
iteration 58900 / 70000: loss 0.750912
iteration 59000 / 70000: loss 0.775759
iteration 59100 / 70000: loss 0.673309
iteration 59200 / 70000: loss 0.818017
train_acc 0.843750, val_acc 0.730000, time 51
iteration 59300 / 70000: loss 0.813751
iteration 59400 / 70000: loss 0.697141
iteration 59500 / 70000: loss 0.607003
train_acc 0.789062, val_acc 0.729000, time 52
iteration 59600 / 70000: loss 0.643509
iteration 59700 / 70000: loss 0.760300
iteration 59800 / 70000: loss 0.773346
iteration 59900 / 70000: loss 0.691380
train_acc 0.843750, val_acc 0.727000, time 52
iteration 60000 / 70000: loss 0.750408
iteration 60100 / 70000: loss 0.871450
iteration 60200 / 70000: loss 0.704460
iteration 60300 / 70000: loss 0.604667
train_acc 0.781250, val_acc 0.737000, time 52
iteration 60400 / 70000: loss 0.769461
iteration 60500 / 70000: loss 0.840901
iteration 60600 / 70000: loss 0.714193
iteration 60700 / 70000: loss 0.664626
train_acc 0.734375, val_acc 0.741000, time 53
```

```
iteration 60800 / 70000: loss 0.550712
iteration 60900 / 70000: loss 0.581104
iteration 61000 / 70000: loss 0.682428
iteration 61100 / 70000: loss 0.617747
train_acc 0.796875, val_acc 0.741000, time 53
iteration 61200 / 70000: loss 0.622871
iteration 61300 / 70000: loss 0.702773
iteration 61400 / 70000: loss 0.762026
iteration 61500 / 70000: loss 0.725576
train_acc 0.734375, val_acc 0.736000, time 53
iteration 61600 / 70000: loss 0.666827
iteration 61700 / 70000: loss 0.814486
iteration 61800 / 70000: loss 0.697234
train_acc 0.750000, val_acc 0.736000, time 54
iteration 61900 / 70000: loss 0.679558
iteration 62000 / 70000: loss 0.829697
iteration 62100 / 70000: loss 0.723610
iteration 62200 / 70000: loss 0.745086
train_acc 0.726562, val_acc 0.736000, time 54
iteration 62300 / 70000: loss 0.647633
iteration 62400 / 70000: loss 0.708676
iteration 62500 / 70000: loss 0.581942
iteration 62600 / 70000: loss 0.788497
train_acc 0.812500, val_acc 0.736000, time 54
iteration 62700 / 70000: loss 0.729790
iteration 62800 / 70000: loss 0.812387
iteration 62900 / 70000: loss 0.629651
iteration 63000 / 70000: loss 0.894127
train_acc 0.789062, val_acc 0.739000, time 55
iteration 63100 / 70000: loss 0.675445
iteration 63200 / 70000: loss 0.710630
iteration 63300 / 70000: loss 0.682272
iteration 63400 / 70000: loss 0.754669
train_acc 0.812500, val_acc 0.739000, time 55
iteration 63500 / 70000: loss 0.960681
iteration 63600 / 70000: loss 0.687025
iteration 63700 / 70000: loss 0.702816
train_acc 0.820312, val_acc 0.733000, time 56
iteration 63800 / 70000: loss 0.627647
iteration 63900 / 70000: loss 0.815521
iteration 64000 / 70000: loss 0.665738
iteration 64100 / 70000: loss 0.802782
train_acc 0.726562, val_acc 0.750000, time 56
iteration 64200 / 70000: loss 0.712189
iteration 64300 / 70000: loss 0.608308
iteration 64400 / 70000: loss 0.609079
iteration 64500 / 70000: loss 0.682450
train_acc 0.820312, val_acc 0.735000, time 56
iteration 64600 / 70000: loss 0.583255
iteration 64700 / 70000: loss 0.681972
iteration 64800 / 70000: loss 0.823839
iteration 64900 / 70000: loss 0.786124
train_acc 0.804688, val_acc 0.735000, time 57
iteration 65000 / 70000: loss 0.689299
iteration 65100 / 70000: loss 0.692767
iteration 65200 / 70000: loss 0.691779
iteration 65300 / 70000: loss 0.864899
train_acc 0.726562, val_acc 0.737000, time 57
iteration 65400 / 70000: loss 0.810011
iteration 65500 / 70000: loss 0.725723
iteration 65600 / 70000: loss 0.671036
iteration 65700 / 70000: loss 0.628926
train_acc 0.828125, val_acc 0.734000, time 57
iteration 65800 / 70000: loss 0.900002
iteration 65900 / 70000: loss 0.740613
iteration 66000 / 70000: loss 0.562329
```

```
train_acc 0.757812, val_acc 0.738000, time 58
iteration 66100 / 70000: loss 0.557871
iteration 66200 / 70000: loss 0.652382
iteration 66300 / 70000: loss 0.572430
iteration 66400 / 70000: loss 0.575010
train_acc 0.773438, val_acc 0.748000, time 58
iteration 66500 / 70000: loss 0.556576
iteration 66600 / 70000: loss 0.651001
iteration 66700 / 70000: loss 0.760619
iteration 66800 / 70000: loss 0.776056
train_acc 0.828125, val_acc 0.739000, time 58
iteration 66900 / 70000: loss 0.684883
iteration 67000 / 70000: loss 0.932047
iteration 67100 / 70000: loss 0.695935
iteration 67200 / 70000: loss 0.725615
train_acc 0.773438, val_acc 0.740000, time 59
iteration 67300 / 70000: loss 0.678493
iteration 67400 / 70000: loss 0.708723
iteration 67500 / 70000: loss 0.549422
iteration 67600 / 70000: loss 0.708957
train_acc 0.796875, val_acc 0.735000, time 59
iteration 67700 / 70000: loss 0.709389
iteration 67800 / 70000: loss 0.860724
iteration 67900 / 70000: loss 0.712943
train_acc 0.828125, val_acc 0.741000, time 59
iteration 68000 / 70000: loss 0.639184
iteration 68100 / 70000: loss 0.746590
iteration 68200 / 70000: loss 0.628429
iteration 68300 / 70000: loss 0.639412
train_acc 0.828125, val_acc 0.743000, time 60
iteration 68400 / 70000: loss 0.651731
iteration 68500 / 70000: loss 0.590436
iteration 68600 / 70000: loss 0.676012
iteration 68700 / 70000: loss 0.720614
train_acc 0.835938, val_acc 0.744000, time 60
iteration 68800 / 70000: loss 0.708387
iteration 68900 / 70000: loss 0.619908
iteration 69000 / 70000: loss 0.622283
iteration 69100 / 70000: loss 0.751334
train_acc 0.789062, val_acc 0.737000, time 60
iteration 69200 / 70000: loss 0.899096
iteration 69300 / 70000: loss 0.693297
iteration 69400 / 70000: loss 0.685889
iteration 69500 / 70000: loss 0.583009
train_acc 0.804688, val_acc 0.743000, time 61
iteration 69600 / 70000: loss 0.794187
iteration 69700 / 70000: loss 0.649544
iteration 69800 / 70000: loss 0.589749
iteration 69900 / 70000: loss 0.838994
train_acc 0.757812, val_acc 0.738000, time 61
iteration 70000 / 70000: loss 0.638144
Train accuracy:  0.79912244898
Validation accuracy:  0.743
Test accuracy:  0.75
time 103.380934763
```