ECE C247 HW 2 Solution

Ashish Kumar Singh (UID:105479019)

January 25, 2022

Problem 2. Softmax classifier gradient

Solution 2. Likelihood $\mathcal{L}$ and log-likelihood $\mathcal{LL}$ can be written as:

$$\mathcal{L} = \prod_{i=1}^{m} p(y^{(i)}|x^{(i)}, \theta)$$

$$\mathcal{L} = \prod_{i=1}^{m} softmax_{y^{(i)}}(x^{(i)})$$

$$\mathcal{L} = \prod_{i=1}^{m} \frac{e^{a_{y^{(i)}}(x^{(i)})}}{\sum_{k=1}^{c} e^{a_k(x^{(i)})}}$$

$$\mathcal{LL} = log\bigg(\prod_{i=1}^{m} \frac{e^{a_{y^{(i)}}(x^{(i)})}}{\sum_{k=1}^{c} e^{a_k(x^{(i)})}}\bigg)$$

$$\mathcal{LL} = \sum_{i=1}^{m} log\bigg(\frac{e^{a_{y^{(i)}}(x^{(i)})}}{\sum_{k=1}^{c} e^{a_k(x^{(i)})}}\bigg)$$

$$\mathcal{LL} = \sum_{i=1}^{m} \bigg(a_{y^{(i)}}(x^{(i)}) - log\sum_{k=1}^{c} e^{a_k(x^{(i)})}\bigg)$$

Using $\tilde{x}^{(i)} = \begin{bmatrix} x^{(i)} \\ 1 \end{bmatrix}$ and $\tilde{w}_i = \begin{bmatrix} w_i \\ b_i \end{bmatrix}$ and $a_j(x) = w_j^T x + b_j$

$$\mathcal{LL} = \sum_{j=1}^{m} \bigg(\tilde{w}_{y^{(j)}}^T \tilde{x}^{(j)} - log\sum_{k=1}^{c} e^{\tilde{w}_k^T \tilde{x}^{(j)}}\bigg)$$

$$\nabla_{\tilde{w}_i}\mathcal{LL} = \nabla_{\tilde{w}_i}\bigg(\sum_{j=1}^{m} \bigg(\tilde{w}_{y^{(j)}}^T \tilde{x}^{(j)} - log\sum_{k=1}^{c} e^{\tilde{w}_k^T \tilde{x}^{(j)}}\bigg)\bigg)$$

$$\nabla_{\tilde{w}_i}\mathcal{LL} = \sum_{j=1}^{m} \bigg(\mathcal{I}(y^{(j)} = i)\tilde{x}^{(j)} - \frac{e^{\tilde{w}_i^T \tilde{x}^{(j)}}\tilde{x}^{(j)}}{\sum_{k=1}^{c} e^{\tilde{w}_k^T \tilde{x}^{(j)}}}\bigg)$$

$$\nabla_{\tilde{w}_i}\mathcal{LL} = \sum_{j=1}^{m} \tilde{x}^{(j)} \left(\mathcal{I}(y^{(j)} = i) - \frac{e^{\tilde{w}_i^T \tilde{x}^{(j)}}}{\sum_{k=1}^{c} e^{\tilde{w}_k^T \tilde{x}^{(j)}}} \right)$$

where $\mathcal{I}(y^{(j)} = i)$ is 1 if $y^{(j)} = i$ else 0,

Using $\nabla_{\tilde{w}_i}\mathcal{L} = \begin{bmatrix} \nabla_{w_i}\mathcal{L} \\ \nabla_{b_i}\mathcal{L} \end{bmatrix}$

$$\nabla_{w_i}\mathcal{LL} = \sum_{j=1}^{m} x^{(j)} \left(\mathcal{I}(y^{(j)} = i) - \frac{e^{(w_i^T x^{(j)} + b_i)}}{\sum_{k=1}^{c} e^{(w_k^T x^{(j)} + b_k)}} \right)$$

$$\nabla_{b_i}\mathcal{LL} = \sum_{j=1}^{m} \left(\mathcal{I}(y^{(j)} = i) - \frac{e^{(w_i^T x^{(j)} + b_i)}}{\sum_{k=1}^{c} e^{(w_k^T x^{(j)} + b_k)}} \right)$$

Problem 1 and 3 solution notebook is attached below

This is the k-nearest neighbors workbook for ECE C147/C247 Assignment #2

Please follow the notebook linearly to implement k-nearest neighbors.

Please print out the workbook entirely when completed.

The goal of this workbook is to give you experience with the data, training and evaluating a simple classifier, k-fold cross validation, and as a Python refresher.

Import the appropriate libraries

In [73]:
```
import numpy as np # for doing most of our calculations
import matplotlib.pyplot as plt# for plotting
from utils.data_utils import load_CIFAR10 # function to load the CIFAR-10 dataset.

# Load matplotlib images inline
%matplotlib inline

# These are important for reloading any code you write in external .py files.
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

In [74]:
```
# Set the path to the CIFAR-10 data
cifar10_dir = './cifar-10-batches-py' # You need to update this line
X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

# As a sanity check, we print out the size of the training and test data.
print('Training data shape: ', X_train.shape)
print('Training labels shape: ', y_train.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

```
Training data shape:  (50000, 32, 32, 3)
Training labels shape:  (50000,)
Test data shape:  (10000, 32, 32, 3)
Test labels shape:  (10000,)
```

In [75]:
```
# Visualize some examples from the dataset.
# We show a few examples of training images from each class.
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
num_classes = len(classes)
samples_per_class = 7
for y, cls in enumerate(classes):
    idxs = np.flatnonzero(y_train == y)
    idxs = np.random.choice(idxs, samples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt_idx = i * num_classes + y + 1
        plt.subplot(samples_per_class, num_classes, plt_idx)
        plt.imshow(X_train[idx].astype('uint8'))
        plt.axis('off')
        if i == 0:
            plt.title(cls)
plt.show()
```

In [76]:
```
# Subsample the data for more efficient code execution in this exercise
num_training = 5000
mask = list(range(num_training))
X_train = X_train[mask]
y_train = y_train[mask]

num_test = 500
mask = list(range(num_test))
X_test = X_test[mask]
y_test = y_test[mask]

# Reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
print(X_train.shape, X_test.shape)
```

```
(5000, 3072) (500, 3072)
```

K-nearest neighbors

In the following cells, you will build a KNN classifier and choose hyperparameters via k-fold cross-validation.

In [77]:
```
# Import the KNN class
from nndl import KNN
```

In [78]:
```
# Declare an instance of the knn class.
knn = KNN()

# Train the classifier.
#   We have implemented the training of the KNN classifier.
#   Look at the train function in the KNN class to see what this does.
knn.train(X=X_train, y=y_train)
```

Questions

(1) Describe what is going on in the function knn.train().

(2) What are the pros and cons of this training step?

Answers

(1) The function knn.train() is just assigning training data to X_train and training label to y_train in the KNN class.

(2) Pro: No precomputation done, so constant time complexity. Con: Test will take time, for each test data point it will take O(n) time complexity if n is # of training samples.

KNN prediction

In the following sections, you will implement the functions to calculate the distances of test points to training points, and from this information, predict the class of the KNN.

In [79]:
```
# Implement the function compute_distances() in the KNN class.
# Do not worry about the input 'norm' for now; use the default definition of the norm
#   in the code, which is the 2-norm.
# You should only have to fill out the clearly marked sections.

import time
time_start =time.time()

dists_L2 = knn.compute_distances(X=X_test)

print('Time to run code: {}'.format(time.time()-time_start))
print('Frobenius norm of L2 distances: {}'.format(np.linalg.norm(dists_L2, 'fro')))
```

```
Time to run code: 34.35724949836731
Frobenius norm of L2 distances: 7906696.077040902
```

Really slow code

Note: This probably took a while. This is because we use two for loops. We could increase the speed via vectorization, removing the for loops.

If you implemented this correctly, evaluating np.linalg.norm(dists_L2, 'fro') should return: ~7906696

KNN vectorization

The above code took far too long to run. If we wanted to optimize hyperparameters, it would be time-expensive. Thus, we will speed up the code by vectorizing it, removing the for loops.

In [81]:
```
# Implement the function compute_L2_distances_vectorized() in the KNN class.
# In this function, you ought to achieve the same L2 distance but WITHOUT any for loops.
# Note, this is SPECIFIC for the L2 norm.

time_start =time.time()
dists_L2_vectorized = knn.compute_L2_distances_vectorized(X=X_test)
print('Time to run code: {}'.format(time.time()-time_start))
print('Difference in L2 distances between your KNN implementations (should be 0): {}'.format(np
```

```
Time to run code: 0.20433902740478516
Difference in L2 distances between your KNN implementations (should be 0): 0.0
```

Speedup

Depending on your computer speed, you should see a 10-100x speed up from vectorization. On our computer, the vectorized form took 0.36 seconds while the naive implementation took 38.3 seconds.

Implementing the prediction

Now that we have functions to calculate the distances from a test point to given training points, we now implement the function that will predict the test point labels.

In [51]:
```
# Implement the function predict_labels in the KNN class.
# Calculate the training error (num_incorrect / total_samples)
#   from running knn.predict_labels with k=1

error = 1
```

```
# ================================================================ #
# YOUR CODE HERE:
#   Calculate the error rate by calling predict_labels on the test
#   data with k = 1.  Store the error rate in the variable error.
# ================================================================ #
num_incorrect = (knn.predict_labels(dists_L2_vectorized)!=y_test).sum()
total_samples = X_test.shape[0]
error = (num_incorrect / total_samples)
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

print(error)
```

```
0.726
```

If you implemented this correctly, the error should be: 0.726.

This means that the k-nearest neighbors classifier is right 27.4% of the time, which is not great, considering that chance levels are 10%.

Optimizing KNN hyperparameters

In this section, we'll take the KNN classifier that you have constructed and perform cross-validation to choose a best value of k, as well as a best choice of norm.

Create training and validation folds

First, we will create the training and validation folds for use in k-fold cross validation.

In [52]:

```
# Create the dataset folds for cross-valdiation.
num_folds = 5

X_train_folds = []
y_train_folds =  []

# ================================================================ #
# YOUR CODE HERE:
#   Split the training data into num_folds (i.e., 5) folds.
#   X_train_folds is a list, where X_train_folds[i] contains the
#      data points in fold i.
#   y_train_folds is also a list, where y_train_folds[i] contains
#      the corresponding labels for the data in X_train_folds[i]
# ================================================================ #

X_train_folds = [ X_train[i*1000:(i+1)*1000] for i in range(5)]
y_train_folds = [ y_train[i*1000:(i+1)*1000] for i in range(5)]
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

Optimizing the number of nearest neighbors hyperparameter.

In this section, we select different numbers of nearest neighbors and assess which one has the lowest k-fold cross validation error.

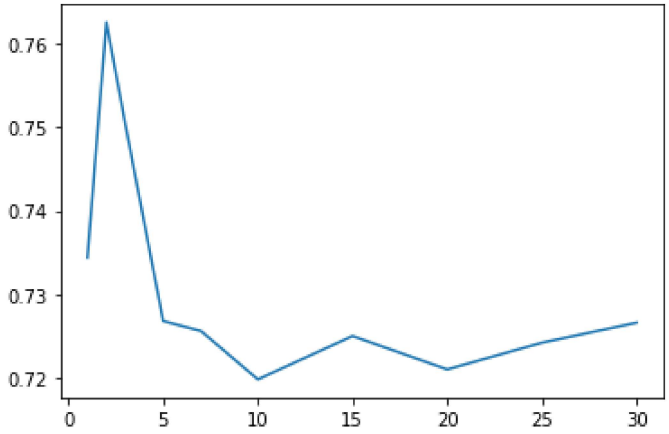
In [68]:

```
time_start =time.time()

ks = [1, 2, 3, 5, 7, 10, 15, 20, 25, 30]

# ================================================================ #
# YOUR CODE HERE:
#   Calculate the cross-validation error for each k in ks, testing
```

```
#   the trained model on each of the 5 folds.  Average these errors
#   together and make a plot of k vs. cross-validation error. Since
#   we are assuming L2 distance here, please use the vectorized code!
#   Otherwise, you might be waiting a long time.
# ================================================================ #
from collections import defaultdict
error = defaultdict(lambda: 0)
for i in range(5):
    #ith fold to be tested

    X_train_ith_fold = X_train_folds[(i+1)%5]
    y_train_ith_fold = y_train_folds[(i+1)%5]
    for j in range(5):
        if j==i or j==((i+1)%5):
            continue
        else:
            X_train_ith_fold = np.append(X_train_ith_fold,X_train_folds[j], axis=0)
            y_train_ith_fold = np.append(y_train_ith_fold,y_train_folds[j], axis=0)

    X_test_ith_fold = X_train_folds[i]
    y_test_ith_fold = y_train_folds[i]

    knn = KNN()
    knn.train(X=X_train_ith_fold, y=y_train_ith_fold)
    dists_L2_vectorized = knn.compute_L2_distances_vectorized(X=X_test_ith_fold)

    for k in ks:
        err = np.mean(knn.predict_labels(dists_L2_vectorized,k)!=y_test_ith_fold)
        total_samples = X_test_ith_fold.shape[0]
        print("fold: ",i," k: ",k," error: ",err )
        error[k]+= err

for k in ks:
    error[k] = error[k]/5
    print("Average error for k= ",k," is ",error[k])

x,y = zip(*(sorted(error.items())))
plt.plot(x,y)
plt.show()
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

print('Computation time: %.2f'%(time.time()-time_start))
```

```
fold:  0  k:  1  error:  0.737
fold:  0  k:  2  error:  0.765
fold:  0  k:  3  error:  0.761
fold:  0  k:  5  error:  0.752
fold:  0  k:  7  error:  0.739
fold:  0  k:  10  error:  0.735
fold:  0  k:  15  error:  0.748
fold:  0  k:  20  error:  0.73
fold:  0  k:  25  error:  0.73
fold:  0  k:  30  error:  0.728
fold:  1  k:  1  error:  0.743
fold:  1  k:  2  error:  0.781
fold:  1  k:  3  error:  0.751
fold:  1  k:  5  error:  0.734
fold:  1  k:  7  error:  0.721
fold:  1  k:  10  error:  0.704
fold:  1  k:  15  error:  0.711
fold:  1  k:  20  error:  0.721
fold:  1  k:  25  error:  0.714
fold:  1  k:  30  error:  0.728
fold:  2  k:  1  error:  0.736
fold:  2  k:  2  error:  0.766
fold:  2  k:  3  error:  0.76
fold:  2  k:  5  error:  0.72
fold:  2  k:  7  error:  0.732
fold:  2  k:  10  error:  0.724
```

```
fold:  2  k:  15  error:  0.722
fold:  2  k:  20  error:  0.721
fold:  2  k:  25  error:  0.723
fold:  2  k:  30  error:  0.714
fold:  3  k:  1  error:  0.722
fold:  3  k:  2  error:  0.753
fold:  3  k:  3  error:  0.734
fold:  3  k:  5  error:  0.708
fold:  3  k:  7  error:  0.712
fold:  3  k:  10  error:  0.716
fold:  3  k:  15  error:  0.718
fold:  3  k:  20  error:  0.718
fold:  3  k:  25  error:  0.73
fold:  3  k:  30  error:  0.735
fold:  4  k:  1  error:  0.734
fold:  4  k:  2  error:  0.748
fold:  4  k:  3  error:  0.746
fold:  4  k:  5  error:  0.72
fold:  4  k:  7  error:  0.724
fold:  4  k:  10  error:  0.72
fold:  4  k:  15  error:  0.726
fold:  4  k:  20  error:  0.715
fold:  4  k:  25  error:  0.724
fold:  4  k:  30  error:  0.728
Average error for k=  1  is  0.7344
Average error for k=  2  is  0.7626000000000002
Average error for k=  3  is  0.7504000000000001
Average error for k=  5  is  0.7267999999999999
Average error for k=  7  is  0.7256
Average error for k=  10  is  0.7198
Average error for k=  15  is  0.725
Average error for k=  20  is  0.721
Average error for k=  25  is  0.7242
Average error for k=  30  is  0.7266
```




```
Computation time: 16.93
```

Questions:

(1) What value of k is best amongst the tested k's?

(2) What is the cross-validation error for this value of k?

Answers:

(1) k=10 gives the best cross-validation error.

(2) cross validation error for k=10 is 0.7198

Optimizing the norm

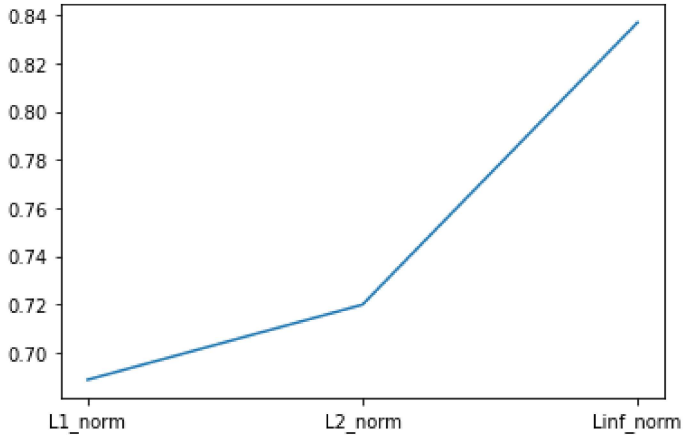
Next, we test three different norms (the 1, 2, and infinity norms) and see which distance metric results in the best cross-validation performance.

In [70]:

```
time_start =time.time()

L1_norm = lambda x: np.linalg.norm(x, ord=1)
L2_norm = lambda x: np.linalg.norm(x, ord=2)
Linf_norm = lambda x: np.linalg.norm(x, ord= np.inf)
norms = [L1_norm, L2_norm, Linf_norm]

# ================================================================ #
# YOUR CODE HERE:
#   Calculate the cross-validation error for each norm in norms, testing
#   the trained model on each of the 5 folds.  Average these errors
#   together and make a plot of the norm used vs the cross-validation error
#   Use the best cross-validation k from the previous part.
#
#   Feel free to use the compute_distances function.  We're testing just
#   three norms, but be advised that this could still take some time.
#   You're welcome to write a vectorized form of the L1- and Linf- norms
#   to speed this up, but it is not necessary.
# ================================================================ #
norm_names = ["L1_norm", "L2_norm", "Linf_norm"]
k=10
error = defaultdict(lambda: 0)
for i in range(5):
    #ith fold to be tested

    X_train_ith_fold = X_train_folds[(i+1)%5]
    y_train_ith_fold = y_train_folds[(i+1)%5]
    for j in range(5):
        if j==i or j==((i+1)%5):
            continue
        else:
            X_train_ith_fold = np.append(X_train_ith_fold,X_train_folds[j], axis=0)
            y_train_ith_fold = np.append(y_train_ith_fold,y_train_folds[j], axis=0)

    X_test_ith_fold = X_train_folds[i]
    y_test_ith_fold = y_train_folds[i]

    knn = KNN()
    knn.train(X=X_train_ith_fold, y=y_train_ith_fold)
    for j,norm in enumerate(norms):
        dists = knn.compute_distances(X=X_test_ith_fold,norm=norm)
        err = np.mean(knn.predict_labels(dists,k)!=y_test_ith_fold)
        print("fold: ",i," norm: ",norm_names[j]," error: ",err )
        error[norm_names[j]]+= err

for j,norm in enumerate(norms):
    error[norm_names[j]] = error[norm_names[j]]/5
    print("Average error for norm= ",norm_names[j]," is ",error[norm_names[j]])

x,y = zip(*(error.items()))
plt.plot(x,y)
plt.show()

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
print('Computation time: %.2f'%(time.time()-time_start))
```

```
fold:  0  norm:  L1_norm  error:  0.711
fold:  0  norm:  L2_norm  error:  0.735
fold:  0  norm:  Linf_norm  error:  0.83
fold:  1  norm:  L1_norm  error:  0.688
fold:  1  norm:  L2_norm  error:  0.704
fold:  1  norm:  Linf_norm  error:  0.833
fold:  2  norm:  L1_norm  error:  0.68
fold:  2  norm:  L2_norm  error:  0.724
```

```
fold:  2  norm:  Linf_norm  error:  0.844
fold:  3  norm:  L1_norm  error:  0.677
fold:  3  norm:  L2_norm  error:  0.716
fold:  3  norm:  Linf_norm  error:  0.842
fold:  4  norm:  L1_norm  error:  0.687
fold:  4  norm:  L2_norm  error:  0.72
fold:  4  norm:  Linf_norm  error:  0.836
Average error for norm=  L1_norm  is  0.6886000000000001
Average error for norm=  L2_norm  is  0.7198
Average error for norm=  Linf_norm  is  0.837
```




```
Computation time: 745.08
```

Questions:

(1) What norm has the best cross-validation error?

(2) What is the cross-validation error for your given norm and k?

Answers:

(1) L1 norm gives the best cross-validation error.

(2) The cross validation for L1 norm with k=10 is 0.6886

Evaluating the model on the testing dataset.

Now, given the optimal k and norm you found in earlier parts, evaluate the testing error of the k-nearest neighbors model.

In [72]:

```
error = 1

# ================================================================ #
# YOUR CODE HERE:
#   Evaluate the testing error of the k-nearest neighbors classifier
#   for your optimal hyperparameters found by 5-fold cross-validation.
# ================================================================ #
k=10
L1_norm = lambda x: np.linalg.norm(x, ord=1)
dists_L1 = knn.compute_distances(X=X_test, norm=L1_norm)
err = np.mean(knn.predict_labels(dists_L1,k)!=y_test)
error = err

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

print('Error rate achieved: {}'.format(error))
```

```
Error rate achieved: 0.716
```

Question:

How much did your error improve by cross-validation over naively choosing $k = 1$ and using the L2-norm?

Answer:

The error rate improved from 0.726 to 0.716

```python
import numpy as np
import pdb
import scipy.stats


class KNN(object):

  def __init__(self):
    pass

  def train(self, X, y):
    """
    Inputs:
    - X is a numpy array of size (num_examples, D)
    - y is a numpy array of size (num_examples, )
    """
    self.X_train = X
    self.y_train = y

  def compute_distances(self, X, norm=None):
    """
    Compute the distance between each test point in X and each training point
    in self.X_train.

    Inputs:
    - X: A numpy array of shape (num_test, D) containing test data.
    - norm: the function with which the norm is taken.

    Returns:
    - dists: A numpy array of shape (num_test, num_train) where dists[i, j]
      is the Euclidean distance between the ith test point and the jth training
      point.
    """
    if norm is None:
      norm = lambda x: np.sqrt(np.sum(x**2))
      #norm = 2

    num_test = X.shape[0]
    num_train = self.X_train.shape[0]
    dists = np.zeros((num_test, num_train))
    for i in np.arange(num_test):

      for j in np.arange(num_train):
        # ================================================================ #
        # YOUR CODE HERE:
        #   Compute the distance between the ith test point and the jth
        #   training point using norm(), and store the result in dists[i, j].
        # ================================================================ #

        dists[i,j] = norm(self.X_train[j] - X[i])


        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #

    return dists

  def compute_L2_distances_vectorized(self, X):
    """
    Compute the distance between each test point in X and each training point
    in self.X_train WITHOUT using any for loops.

    Inputs:
    - X: A numpy array of shape (num_test, D) containing test data.

    Returns:
    - dists: A numpy array of shape (num_test, num_train) where dists[i, j]
      is the Euclidean distance between the ith test point and the jth training
      point.
    """
    num_test = X.shape[0]
```

```
    num_train = self.X_train.shape[0]
    dists = np.zeros((num_test, num_train))

    # ========================================================== #
    # YOUR CODE HERE:
    #   Compute the L2 distance between the ith test point and the jth
    #   training point and store the result in dists[i, j].  You may
    #   NOT use a for loop (or list comprehension).  You may only use
    #   numpy operations.
    #
    #   HINT: use broadcasting.  If you have a shape (N,1) array and
    #   a shape (M,) array, adding them together produces a shape (N, M)
    #   array.
    # ========================================================== #

    dists = np.sqrt(np.square(X).sum(1)[:,np.newaxis] +
    np.square(self.X_train).sum(1) - 2*np.dot(X,self.X_train.T))

    # ========================================================== #
    # END YOUR CODE HERE
    # ========================================================== #

    return dists


  def predict_labels(self, dists, k=1):
    """
    Given a matrix of distances between test points and training points,
    predict a label for each test point.

    Inputs:
    - dists: A numpy array of shape (num_test, num_train) where dists[i, j]
      gives the distance betwen the ith test point and the jth training point.

    Returns:
    - y: A numpy array of shape (num_test,) containing predicted labels for the
      test data, where y[i] is the predicted label for the test point X[i].
    """
    num_test = dists.shape[0]
    y_pred = np.zeros(num_test)
    for i in np.arange(num_test):
      # A list of length k storing the labels of the k nearest neighbors to
      # the ith test point.
      closest_y = []
      # ========================================================== #
      # YOUR CODE HERE:
      #   Use the distances to calculate and then store the labels of
      #   the k-nearest neighbors to the ith test point.  The function
      #   numpy.argsort may be useful.
      #
      #   After doing this, find the most common label of the k-nearest
      #   neighbors.  Store the predicted label of the ith training example
      #   as y_pred[i].  Break ties by choosing the smaller label.
      # ========================================================== #

      closest_y = [self.y_train[j] for j in np.argsort(dists[i,:])[:k]]
      y_pred[i] = scipy.stats.mode(closest_y)[0][0]
      # ========================================================== #
      # END YOUR CODE HERE
      # ========================================================== #

    return y_pred
```

This is the softmax workbook for ECE C147/C247 Assignment #2

Please follow the notebook linearly to implement a softmax classifier.

Please print out the workbook entirely when completed.

The goal of this workbook is to give you experience with training a softmax classifier.

In [2]:
```
import random
import numpy as np
from utils.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

%matplotlib inline
%load_ext autoreload
%autoreload 2
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

In [4]:
```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000, num_dev=500):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    it for the linear classifier. These are the same steps as we used for the
    SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = './cifar-10-batches-py' # You need to update this line
    X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

    # subsample the data
    mask = list(range(num_training, num_training + num_validation))
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = list(range(num_training))
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = list(range(num_test))
    X_test = X_test[mask]
    y_test = y_test[mask]
    mask = np.random.choice(num_training, num_dev, replace=False)
    X_dev = X_train[mask]
    y_dev = y_train[mask]

    # Preprocessing: reshape the image data into rows
    X_train = np.reshape(X_train, (X_train.shape[0], -1))
    X_val = np.reshape(X_val, (X_val.shape[0], -1))
    X_test = np.reshape(X_test, (X_test.shape[0], -1))
    X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

    # Normalize the data: subtract the mean image
    mean_image = np.mean(X_train, axis = 0)
    X_train -= mean_image
    X_val -= mean_image
    X_test -= mean_image
    X_dev -= mean_image

    # add bias dimension and transform into columns
    X_train = np.hstack([X_train, np.ones((X_train.shape[0], 1))])
    X_val = np.hstack([X_val, np.ones((X_val.shape[0], 1))])
    X_test = np.hstack([X_test, np.ones((X_test.shape[0], 1))])
    X_dev = np.hstack([X_dev, np.ones((X_dev.shape[0], 1))])

    return X_train, y_train, X_val, y_val, X_test, y_test, X_dev, y_dev
```

```
# Invoke the above function to get our data.
X_train, y_train, X_val, y_val, X_test, y_test, X_dev, y_dev = get_CIFAR10_data()
print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
print('dev data shape: ', X_dev.shape)
print('dev labels shape: ', y_dev.shape)
```

```
Train data shape:  (49000, 3073)
Train labels shape:  (49000,)
Validation data shape:  (1000, 3073)
Validation labels shape:  (1000,)
Test data shape:  (1000, 3073)
Test labels shape:  (1000,)
dev data shape:  (500, 3073)
dev labels shape:  (500,)
```

Training a softmax classifier.

The following cells will take you through building a softmax classifier. You will implement its loss function, then subsequently train it with gradient descent. Finally, you will choose the learning rate of gradient descent to optimize its classification performance.

In [5]:
```
from nndl import Softmax
```

In [6]:
```
# Declare an instance of the Softmax class.
# Weights are initialized to a random value.
# Note, to keep people's first solutions consistent, we are going to use a random seed.

np.random.seed(1)

num_classes = len(np.unique(y_train))
num_features = X_train.shape[1]

softmax = Softmax(dims=[num_classes, num_features])
```

Softmax loss

In [16]:
```
## Implement the loss function of the softmax using a for loop over
#  the number of examples

loss = softmax.loss(X_train, y_train)
```

In [17]:
```
print(loss)
```

```
2.3277607028048966
```

Question:

You'll notice the loss returned by the softmax is about 2.3 (if implemented correctly). Why does this make sense?

Answer:

Beacuse the weights are randomly initialized and are not yet trained, so their output will predict randomly one of the 10 classes, so the probability for guessing correct class will be 0.1, Softmax loss for this should be -

$\ln(0.1) = \ln(10) \approx 2.3$

Softmax gradient

In [47]:
```
## Calculate the gradient of the softmax loss in the Softmax class.
# For convenience, we'll write one function that computes the loss
#   and gradient together, softmax.loss_and_grad(X, y)
# You may copy and paste your loss code from softmax.loss() here, and then
#   use the appropriate intermediate values to calculate the gradient.

loss, grad = softmax.loss_and_grad(X_dev,y_dev)

# Compare your gradient to a gradient check we wrote.
# You should see relative gradient errors on the order of 1e-07 or less if you implemented the
softmax.grad_check_sparse(X_dev, y_dev, grad)
```

```
numerical: -3.647501 analytic: -3.647501, relative error: 5.572790e-09
numerical: 0.451700 analytic: 0.451700, relative error: 2.844425e-08
numerical: -3.732429 analytic: -3.732429, relative error: 1.804868e-08
numerical: 0.647963 analytic: 0.647963, relative error: 6.568576e-08
numerical: -1.820191 analytic: -1.820191, relative error: 2.507468e-08
numerical: 1.109531 analytic: 1.109531, relative error: 1.441653e-09
numerical: -5.199350 analytic: -5.199350, relative error: 5.170286e-09
numerical: 0.594704 analytic: 0.594704, relative error: 4.507831e-09
numerical: 1.727080 analytic: 1.727080, relative error: 1.661232e-08
numerical: 1.027455 analytic: 1.027455, relative error: 7.851578e-08
```

A vectorized version of Softmax

To speed things up, we will vectorize the loss and gradient calculations. This will be helpful for stochastic gradient descent.

In [22]:
```
import time
```

In [83]:
```
## Implement softmax.fast_loss_and_grad which calculates the loss and gradient
#    WITHOUT using any for loops.

# Standard loss and gradient
tic = time.time()
loss, grad = softmax.loss_and_grad(X_dev, y_dev)
toc = time.time()
print('Normal loss / grad_norm: {} / {} computed in {}s'.format(loss, np.linalg.norm(grad, 'fro

tic = time.time()
loss_vectorized, grad_vectorized = softmax.fast_loss_and_grad(X_dev, y_dev)
toc = time.time()
print('Vectorized loss / grad: {} / {} computed in {}s'.format(loss_vectorized, np.linalg.norm(

# The losses should match but your vectorized implementation should be much faster.
print('difference in loss / grad: {} /{} '.format(loss - loss_vectorized, np.linalg.norm(grad -

# You should notice a speedup with the same output.
```

```
Normal loss / grad_norm: 2.321937839839383 / 355.5657032496941 computed in 0.04790449142456055s
Vectorized loss / grad: 2.321937839839384 / 355.5657032496941 computed in 0.002991199493408203s
difference in loss / grad: -8.881784197001252e-16 /3.2759122096389006e-13
```

Stochastic gradient descent

We now implement stochastic gradient descent. This uses the same principles of gradient descent we discussed in class, however, it calculates the gradient by only using examples from a subset of the training set (so each gradient calculation is faster).

Question:

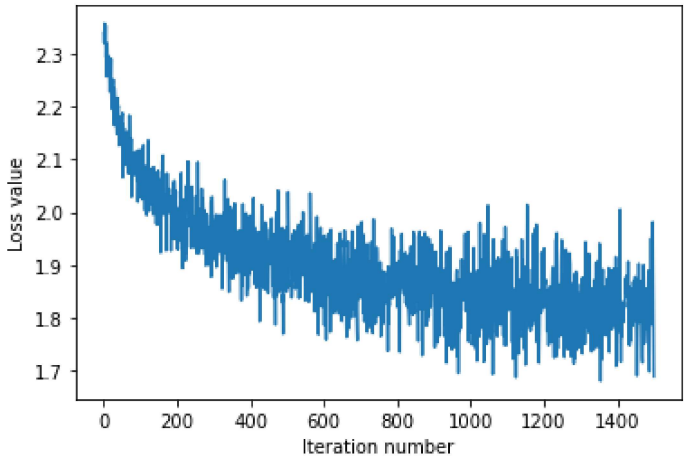
How should the softmax gradient descent training step differ from the svm training step, if at all?

Answer:

Both uses gradient descent.

In [87]:
```
# Implement softmax.train() by filling in the code to extract a batch of data
# and perform the gradient step.
import time


tic = time.time()
loss_hist = softmax.train(X_train, y_train, learning_rate=1e-7,
                          num_iters=1500, verbose=True)
toc = time.time()
print('That took {}s'.format(toc - tic))

plt.plot(loss_hist)
plt.xlabel('Iteration number')
plt.ylabel('Loss value')
plt.show()
```

```
iteration 0 / 1500: loss 2.340083207460104
iteration 100 / 1500: loss 2.0796101858228555
iteration 200 / 1500: loss 1.9509828239589024
iteration 300 / 1500: loss 1.9094554741811902
iteration 400 / 1500: loss 1.9249155589928528
iteration 500 / 1500: loss 1.8564142779293118
iteration 600 / 1500: loss 1.955236276552195
iteration 700 / 1500: loss 1.8739863809177661
iteration 800 / 1500: loss 1.8282954997114702
iteration 900 / 1500: loss 1.8613853134130016
iteration 1000 / 1500: loss 1.801377358060681
iteration 1100 / 1500: loss 1.838523870326722
iteration 1200 / 1500: loss 1.93128662446051
iteration 1300 / 1500: loss 1.8633359921607544
iteration 1400 / 1500: loss 1.7841174341843271
That took 5.049999237060547s
```



Evaluate the performance of the trained softmax classifier on the validation data.

In [88]:
```
## Implement softmax.predict() and use it to compute the training and testing error.

y_train_pred = softmax.predict(X_train)
print('training accuracy: {}'.format(np.mean(np.equal(y_train,y_train_pred), )))
y_val_pred = softmax.predict(X_val)
print('validation accuracy: {}'.format(np.mean(np.equal(y_val, y_val_pred)), ))
```

```
training accuracy: 0.38042857142857145
validation accuracy: 0.386
```

Optimize the softmax classifier

You may copy and paste your optimization code from the SVM here.

In [75]:
```
np.finfo(float).eps
```

Out[75]:
```
2.220446049250313e-16
```

In [82]:
```
# ================================================================ #
# YOUR CODE HERE:
#   Train the Softmax classifier with different learning rates and
#     evaluate on the validation data.
#   Report:
#     - The best learning rate of the ones you tested.
#     - The best validation accuracy corresponding to the best validation error.
#
#   Select the SVM that achieved the best validation error and report
#     its error rate on the test set.
# ================================================================ #
from collections import defaultdict
lrs = [1e-3,1e-4,1e-5,1e-6,1e-7,1e-8,1e-9,1e-10]
lrs_metrics = defaultdict(list) #training error, val error, test_error
for lr in lrs:
    print("Learning rate: ",lr)
    softmax = Softmax(dims=[num_classes, num_features])
    loss_hist = softmax.train(X_train, y_train, learning_rate=lr,
                      num_iters=1500, verbose=True)
    y_train_pred = softmax.predict(X_train)
    train_acc = np.mean(np.equal(y_train,y_train_pred))
    y_val_pred = softmax.predict(X_val)
    val_acc = np.mean(np.equal(y_val,y_val_pred))
    y_test_pred = softmax.predict(X_test)
    test_acc = np.mean(np.equal(y_test,y_test_pred))

    lrs_metrics[lr].append(train_acc)
    lrs_metrics[lr].append(val_acc)
    lrs_metrics[lr].append(test_acc)

best_val = 0
for k,v in lrs_metrics.items():
    print("lr: ",k,"train acc: ",v[0],"val acc: ",v[1])
    if best_val<v[1]:
        best_val=v[1]
        best_lr = k
        best_test = v[2]

print("Best lr: ",best_lr)
print("Best val acc: ", best_val, "Best val err: ", 1-best_val)
print("Best test acc: ",best_test, "Best test err: ",1-best_test)

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

```
Learning rate:  0.001
iteration 0 / 1500: loss 2.2835571027153025
iteration 100 / 1500: loss nan
iteration 200 / 1500: loss nan
iteration 300 / 1500: loss nan
iteration 400 / 1500: loss nan
iteration 500 / 1500: loss nan
iteration 600 / 1500: loss nan
iteration 700 / 1500: loss nan
iteration 800 / 1500: loss nan
```

```
iteration 900 / 1500: loss nan
iteration 1000 / 1500: loss nan
iteration 1100 / 1500: loss nan
iteration 1200 / 1500: loss nan
iteration 1300 / 1500: loss nan
iteration 1400 / 1500: loss nan
Learning rate:  0.0001
iteration 0 / 1500: loss 2.3741269036684773
iteration 100 / 1500: loss 21.566058045533524
iteration 200 / 1500: loss 21.58936956028798
iteration 300 / 1500: loss 23.002643868236717
iteration 400 / 1500: loss 28.245920728779094
iteration 500 / 1500: loss 25.92942062535044
iteration 600 / 1500: loss 19.075525790192017
iteration 700 / 1500: loss 23.70727229305301
iteration 800 / 1500: loss 22.159543997997456
iteration 900 / 1500: loss 19.685382694143737
iteration 1000 / 1500: loss 18.53918056240376
iteration 1100 / 1500: loss 37.89698044580087
iteration 1200 / 1500: loss 26.190186281951014
iteration 1300 / 1500: loss 17.47048053428887
iteration 1400 / 1500: loss 34.95236455776998
Learning rate:  1e-05
iteration 0 / 1500: loss 2.3583198724679124
iteration 100 / 1500: loss 3.1441386776978093
iteration 200 / 1500: loss 3.480804618374719
iteration 300 / 1500: loss 2.542572878400548
iteration 400 / 1500: loss 3.147300913233951
iteration 500 / 1500: loss 3.426142923236357
iteration 600 / 1500: loss 2.5021331818079116
iteration 700 / 1500: loss 2.474520066343636
iteration 800 / 1500: loss 2.382328730749394
iteration 900 / 1500: loss 3.4308575589936847
iteration 1000 / 1500: loss 2.081864808305783
iteration 1100 / 1500: loss 2.7300271801069598
iteration 1200 / 1500: loss 2.541404542039993
iteration 1300 / 1500: loss 2.7075820098026564
iteration 1400 / 1500: loss 2.551841440383185
Learning rate:  1e-06
iteration 0 / 1500: loss 2.3416597885450123
iteration 100 / 1500: loss 1.8391973732882259
iteration 200 / 1500: loss 1.8210858476184548
iteration 300 / 1500: loss 1.7657174495064851
iteration 400 / 1500: loss 1.7182399427694128
iteration 500 / 1500: loss 1.728727141191668
iteration 600 / 1500: loss 1.7695810989491774
iteration 700 / 1500: loss 1.68452544070862
iteration 800 / 1500: loss 1.633023856888361
iteration 900 / 1500: loss 1.6483173750169164
iteration 1000 / 1500: loss 1.7900163450298086
iteration 1100 / 1500: loss 1.8000270831639398
iteration 1200 / 1500: loss 1.7500070459884969
iteration 1300 / 1500: loss 1.7140488624121668
iteration 1400 / 1500: loss 1.84134027719777
Learning rate:  1e-07
iteration 0 / 1500: loss 2.3513053580344856
iteration 100 / 1500: loss 2.068895539834847
iteration 200 / 1500: loss 1.9669995283606965
iteration 300 / 1500: loss 1.9899268364151783
iteration 400 / 1500: loss 1.923360526196894
iteration 500 / 1500: loss 1.8847095451002107
iteration 600 / 1500: loss 1.8619623955002245
iteration 700 / 1500: loss 1.8869851963912563
iteration 800 / 1500: loss 1.8467188794182943
iteration 900 / 1500: loss 1.8593598344392688
iteration 1000 / 1500: loss 1.765959381474745
iteration 1100 / 1500: loss 1.840633087672667
iteration 1200 / 1500: loss 1.8663659887430333
iteration 1300 / 1500: loss 1.8642120445976074
iteration 1400 / 1500: loss 1.9952288177197806
Learning rate:  1e-08
```

```
iteration 0 / 1500: loss 2.3440559196467468
iteration 100 / 1500: loss 2.2419265217806554
iteration 200 / 1500: loss 2.251037140420148
iteration 300 / 1500: loss 2.2298865788873776
iteration 400 / 1500: loss 2.144391076798619
iteration 500 / 1500: loss 2.156646086772012
iteration 600 / 1500: loss 2.144818075555676
iteration 700 / 1500: loss 2.1036041838125934
iteration 800 / 1500: loss 2.0768398300825908
iteration 900 / 1500: loss 2.0887748140337137
iteration 1000 / 1500: loss 2.0983620723624936
iteration 1100 / 1500: loss 2.0631575942638785
iteration 1200 / 1500: loss 2.0613572596050656
iteration 1300 / 1500: loss 2.101202796663624
iteration 1400 / 1500: loss 1.9568810330947901
Learning rate:  1e-09
iteration 0 / 1500: loss 2.307638514938267
iteration 100 / 1500: loss 2.3373064302705306
iteration 200 / 1500: loss 2.3230306476333418
iteration 300 / 1500: loss 2.309552587840203
iteration 400 / 1500: loss 2.306594321755637
iteration 500 / 1500: loss 2.3242400819076585
iteration 600 / 1500: loss 2.3074075612470675
iteration 700 / 1500: loss 2.2834966359908306
iteration 800 / 1500: loss 2.2780901677902334
iteration 900 / 1500: loss 2.299201740697041
iteration 1000 / 1500: loss 2.2503681584965385
iteration 1100 / 1500: loss 2.2628266644397157
iteration 1200 / 1500: loss 2.2755902076033556
iteration 1300 / 1500: loss 2.275820827342614
iteration 1400 / 1500: loss 2.2430451942099943
Learning rate:  1e-10
iteration 0 / 1500: loss 2.3516602062074345
iteration 100 / 1500: loss 2.3584117826733135
iteration 200 / 1500: loss 2.3317077500657803
iteration 300 / 1500: loss 2.3243597562904013
iteration 400 / 1500: loss 2.3981623126706966
iteration 500 / 1500: loss 2.349918053086092
iteration 600 / 1500: loss 2.3234021809969914
iteration 700 / 1500: loss 2.289166427605307
iteration 800 / 1500: loss 2.319720787510532
iteration 900 / 1500: loss 2.3327406191152007
iteration 1000 / 1500: loss 2.3269640249527117
iteration 1100 / 1500: loss 2.3533928343598776
iteration 1200 / 1500: loss 2.3136784358071196
iteration 1300 / 1500: loss 2.3435259683116154
iteration 1400 / 1500: loss 2.314236459559979
lr:  0.001 train acc:  0.10026530612244898 val acc:  0.087
lr:  0.0001 train acc:  0.2957959183673469 val acc:  0.299
lr:  1e-05 train acc:  0.3962857142857143 val acc:  0.375
lr:  1e-06 train acc:  0.4203877551020408 val acc:  0.404
lr:  1e-07 train acc:  0.379 val acc:  0.393
lr:  1e-08 train acc:  0.2924897959183673 val acc:  0.307
lr:  1e-09 train acc:  0.1613265306122449 val acc:  0.152
lr:  1e-10 train acc:  0.09683673469387755 val acc:  0.115
Best lr:  1e-06
Best val acc:  0.404 Best val err:  0.596
Best test acc:  0.391 Best test err:  0.609
```

```python
import numpy as np


class Softmax(object):

  def __init__(self, dims=[10, 3073]):
    self.init_weights(dims=dims)

  def init_weights(self, dims):
    """
    Initializes the weight matrix of the Softmax classifier.
    Note that it has shape (C, D) where C is the number of
    classes and D is the feature size.
    """
    self.W = np.random.normal(size=dims) * 0.0001

  def loss(self, X, y):
    """
    Calculates the softmax loss.

    Inputs have dimension D, there are C classes, and we operate on minibatches
    of N examples.

    Inputs:
    - X: A numpy array of shape (N, D) containing a minibatch of data.
    - y: A numpy array of shape (N,) containing training labels; y[i] = c means
      that X[i] has label c, where 0 <= c < C.

    Returns a tuple of:
    - loss as single float
    """

    # Initialize the loss to zero.
    loss = 0.0

    # ============================================================= #
    # YOUR CODE HERE:
    #   Calculate the normalized softmax loss.  Store it as the variable loss.
    #   (That is, calculate the sum of the losses of all the training
    #   set margins, and then normalize the loss by the number of
    #   training examples.)
    # ============================================================= #
    for i in range(X.shape[0]):
        ay = self.W.dot(X[i,:])
        max_ay = max(ay)
        loss += -(ay[y[i]]-max_ay) + np.log(np.sum(np.exp(ay-max_ay)))
    loss = loss/X.shape[0]
    # ============================================================= #
    # END YOUR CODE HERE
    # ============================================================= #

    return loss

  def loss_and_grad(self, X, y):
    """
    Same as self.loss(X, y), except that it also returns the gradient.

    Output: grad -- a matrix of the same dimensions as W containing
      the gradient of the loss with respect to W.
    """

    # Initialize the loss and gradient to zero.
    loss = 0.0
    grad = np.zeros_like(self.W)

    # ============================================================= #
    # YOUR CODE HERE:
    #   Calculate the softmax loss and the gradient. Store the gradient
    #   as the variable grad.
    # ============================================================= #
    for i in range(X.shape[0]):
        ay = self.W.dot(X[i,:])
```

```
            max_ay = max(ay)
            loss += -(ay[y[i]]-max_ay) + np.log(np.sum(np.exp(ay-max_ay)))
            grad += (np.exp(ay-max_ay)/np.sum(np.exp(ay-max_ay)))[:,np.newaxis]* X[i,:]
            grad[y[i],:] -=  X[i,:]
        grad = grad/X.shape[0]
        loss = loss/X.shape[0]

        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #

        return loss, grad

    def grad_check_sparse(self, X, y, your_grad, num_checks=10, h=1e-5):
        """
        sample a few random elements and only return numerical
        in these dimensions.
        """

        for i in np.arange(num_checks):
            ix = tuple([np.random.randint(m) for m in self.W.shape])

            oldval = self.W[ix]
            self.W[ix] = oldval + h # increment by h
            fxph = self.loss(X, y)
            self.W[ix] = oldval - h # decrement by h
            fxmh = self.loss(X,y) # evaluate f(x - h)
            self.W[ix] = oldval # reset

            grad_numerical = (fxph - fxmh) / (2 * h)
            grad_analytic = your_grad[ix]
            rel_error = abs(grad_numerical - grad_analytic) / (abs(grad_numerical) +
            abs(grad_analytic))
            print('numerical: %f analytic: %f, relative error: %e' % (grad_numerical,
            grad_analytic, rel_error))

    def fast_loss_and_grad(self, X, y):
        """
        A vectorized implementation of loss_and_grad. It shares the same
        inputs and ouptuts as loss_and_grad.
        """
        loss = 0.0
        grad = np.zeros(self.W.shape) # initialize the gradient as zero

        # ================================================================ #
        # YOUR CODE HERE:
        #   Calculate the softmax loss and gradient WITHOUT any for loops.
        # ================================================================ #
        scores = self.W.dot(X.T)
        ma_scores = scores.max(axis=0, keepdims=True)
        scores -= ma_scores
        exp_scores = np.exp(scores) #(c,n)
        scores_sum = np.sum(exp_scores,axis=0) #(1,n)
        loglike =
        np.log(np.squeeze(np.take_along_axis(exp_scores,y[np.newaxis,:],0),0)/scores_sum)
        #(n,)
        loss = -np.average(loglike)

        grad = (exp_scores/scores_sum).dot(X)
        #grad[y,:] -=  X
        for i,a in enumerate(y):
            grad[a,:] -=  X[i,:]
        grad = grad/X.shape[0]
        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #

        return loss, grad

    def train(self, X, y, learning_rate=1e-3, num_iters=100,
              batch_size=200, verbose=False):
        """
```

```
    Train this linear classifier using stochastic gradient descent.

    Inputs:
    - X: A numpy array of shape (N, D) containing training data; there are N
      training samples each of dimension D.
    - y: A numpy array of shape (N,) containing training labels; y[i] = c
      means that X[i] has label 0 <= c < C for C classes.
    - learning_rate: (float) learning rate for optimization.
    - num_iters: (integer) number of steps to take when optimizing
    - batch_size: (integer) number of training examples to use at each step.
    - verbose: (boolean) If true, print progress during optimization.

    Outputs:
    A list containing the value of the loss function at each training iteration.
    """
    num_train, dim = X.shape
    num_classes = np.max(y) + 1 # assume y takes values 0...K-1 where K is number of
    classes

    self.init_weights(dims=[np.max(y) + 1, X.shape[1]]) # initializes the weights of
    self.W

    # Run stochastic gradient descent to optimize W
    loss_history = []

    for it in np.arange(num_iters):
      X_batch = None
      y_batch = None

      # ========================================================== #
      # YOUR CODE HERE:
      #   Sample batch_size elements from the training data for use in
      #     gradient descent.  After sampling,
      #     - X_batch should have shape: (dim, batch_size)
      #     - y_batch should have shape: (batch_size,)
      #   The indices should be randomly generated to reduce correlations
      #   in the dataset.  Use np.random.choice.  It's okay to sample with
      #   replacement.
      # ========================================================== #
      mask = np.random.choice(X.shape[0], batch_size, replace=True)
      X_batch = X[mask]
      y_batch = y[mask]
      # ========================================================== #
      # END YOUR CODE HERE
      # ========================================================== #

      # evaluate loss and gradient
      loss, grad = self.fast_loss_and_grad(X_batch, y_batch)
      loss_history.append(loss)

      # ========================================================== #
      # YOUR CODE HERE:
      #   Update the parameters, self.W, with a gradient step
      # ========================================================== #
      self.W -= learning_rate*grad

      # ========================================================== #
      # END YOUR CODE HERE
      # ========================================================== #

      if verbose and it % 100 == 0:
        print('iteration {} / {}: loss {}'.format(it, num_iters, loss))

    return loss_history

  def predict(self, X):
    """
    Inputs:
    - X: N x D array of training data. Each row is a D-dimensional point.

    Returns:
    - y_pred: Predicted labels for the data in X. y_pred is a 1-dimensional
```

```
        array of length N, and each element is an integer giving the predicted
        class.
      """
      y_pred = np.zeros(X.shape[0])
      # ========================================================== #
      # YOUR CODE HERE:
      #   Predict the labels given the training data.
      # ========================================================== #
      y_pred = np.argmax(self.W.dot(X.T),axis=0)
      # ========================================================== #
      # END YOUR CODE HERE
      # ========================================================== #

      return y_pred


```