This is the k-nearest neighbors workbook for ECE 239AS Assignment #2

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

Please print out the workbook entirely when completed.

We thank Serena Yeung & Justin Johnson for permission to use code written for the CS 231n class (cs231n.stanford.edu). These are the functions in the cs231n folders and code in the jupyer notebook to preprocess and show the images. The classifiers used are based off of code prepared for CS 231n as well.

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 [1]:

```
import numpy as np # for doing most of our calculations
import matplotlib.pyplot as plt# for plotting
from cs231n.data_utils import load_CIFAR10 # function to load the CIFAR-10 datas
et.

# 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-ipytho
n
%load_ext autoreload
%autoreload 2
```

In [2]:

```python
# Set the path to the CIFAR-10 data
import os
cur_dir = os.getcwd()
cifar10_dir = cur_dir+'/cs231n/datasets/cifar-10-batches-py'
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 [3]:

```
# 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 [4]:

```
# 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 [5]:

```
# Import the KNN class

from nndl import KNN
```

In [6]:

```
# 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 training step for knn simply involves storing the examples' feature representation and corresponding labels. At a high level, we are just carrying around our example data.

(2) Pros: extremely simple to implement (function is 2 lines). Cons: not efficient since we have to keep all of the examples in memory. Also, the actual prediction process may be slow since we have to look at distances of all the points. As we get more data, we will have to store that as well, so it does not scale.

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 [7]:

```
# 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: 38.56055402755737
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 [8]:

```
# 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 fo
r 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.linalg.norm(dists_L2 - dists_L2_vectorized, 'fro')))
```

```
Time to run code: 0.3460578918457031
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 [9]:

```
# 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




# ================================================================ #
# 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.
# ================================================================ #
y_pred = knn.predict_labels(dists_L2_vectorized, k=1)
total_samples = len(y_test)

error = np.sum(y_pred != y_test)/float(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 [10]:

```
# Create the dataset folds for cross-valdiation
import random

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]
# ================================================================ #
rows = X_train.shape[0]
cols = X_train.shape[1]

num_train_points = len(y_train)
rand_indices = random.sample(range(0, num_train_points), num_train_points)


X_train_folds = []
y_train_folds =  []
X_train_folds = np.array(np.array_split(X_train, num_folds))
y_train_folds = np.array(np.array_split(y_train, num_folds))


# ================================================================ #
# 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 [11]:

```
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.

# ================================================================ #
def run_kfold(num_folds, kval, norm=None):
    cv_errors = []
    for holdout in range(0, num_folds):
        #generate indices for non-holdout
        indices = []
        for i in range(0, num_folds):
            if(i != holdout):
                indices.append(i)
        x_train_kf = np.concatenate(X_train_folds[indices])
        y_train_kf = np.concatenate(y_train_folds[indices])

        #train knn classifier
        knn = KNN()
        knn.train(x_train_kf, y_train_kf)

        # Train the classifier.
        dists = []
        if norm == None:
            dists = knn.compute_L2_distances_vectorized(X=X_train_folds[holdout]
) #use ho for test
        else:
            dists = knn.compute_distances(X=X_train_folds[holdout], norm=norm)

        y_pred = knn.predict_labels(dists, k=kval)

        total_samples = len(y_train_folds[holdout])
        num_incorrect = np.sum(y_pred != y_train_folds[holdout])

        error = num_incorrect/float(total_samples)

        cv_errors.append(error)

    #endfor
    #calculate CV averaged error
    print("CV errors: ", cv_errors)
    avg_error = np.mean(cv_errors)
    print("Avg error: ", avg_error)
    return avg_error

error_dict = {}
errors = []
for k in ks:
    error = run_kfold(num_folds, k, norm=None)
    error_dict[k] = error
    errors.append(error)

print(error_dict)

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

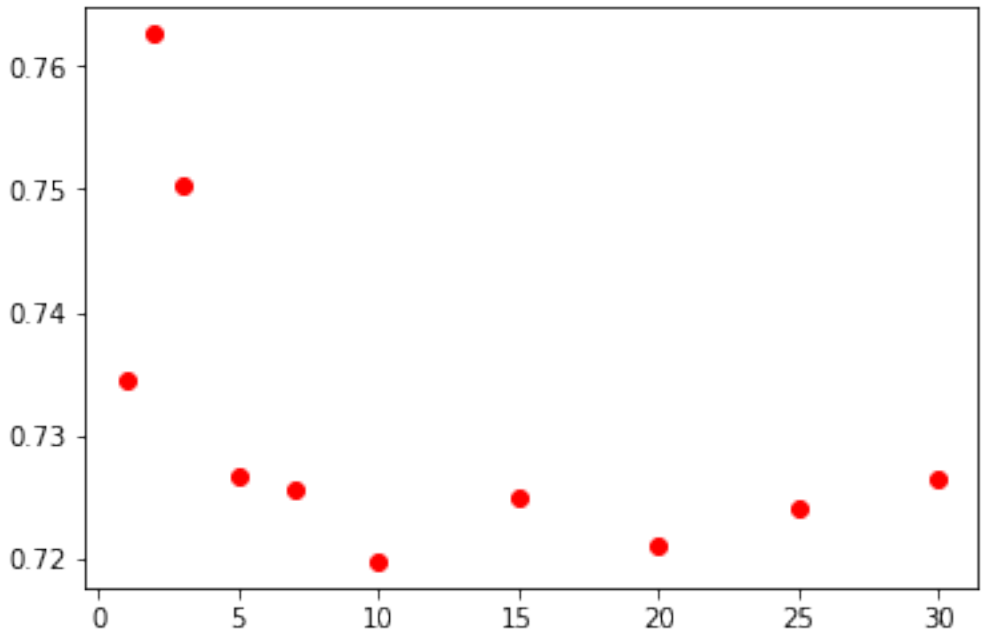
```
print('Computation time: %.2f'%(time.time()-time_start))
```

```
CV errors:  [0.7369999999999999, 0.7429999999999999, 0.73599999999
999999, 0.7219999999999998, 0.7339999999999999]
Avg error:  0.7344
CV errors:  [0.7650000000000001, 0.7810000000000003, 0.76600000000
000001, 0.753, 0.748]
Avg error:  0.7626
CV errors:  [0.7610000000000001, 0.751, 0.7600000000000001, 0.7339
9999999999999, 0.746]
Avg error:  0.7504
CV errors:  [0.752, 0.7339999999999999, 0.7199999999999997, 0.7079
9999999999996, 0.7199999999999997]
Avg error:  0.7268
CV errors:  [0.7389999999999999, 0.7209999999999997, 0.73199999999
999998, 0.7119999999999997, 0.7239999999999998]
Avg error:  0.7256
CV errors:  [0.7349999999999999, 0.7039999999999996, 0.72399999999
999998, 0.7159999999999997, 0.7199999999999997]
Avg error:  0.7198
CV errors:  [0.748, 0.7109999999999997, 0.7219999999999998, 0.7179
9999999999997, 0.7259999999999998]
Avg error:  0.725
CV errors:  [0.7299999999999998, 0.7209999999999997, 0.72099999999
999997, 0.7179999999999997, 0.7149999999999997]
Avg error:  0.721
CV errors:  [0.7299999999999998, 0.7139999999999997, 0.72299999999
999998, 0.7299999999999998, 0.7239999999999998]
Avg error:  0.7242
CV errors:  [0.7279999999999998, 0.7279999999999998, 0.71399999999
999997, 0.7349999999999999, 0.7279999999999998]
Avg error:  0.7266
{1: 0.7344000000000005, 2: 0.7626000000000017, 3: 0.75040000000000
007, 5: 0.7267999999999989, 7: 0.7256000000000002, 10: 0.7198, 15:
0.7249999999999998, 20: 0.7209999999999997, 25: 0.72419999999999999
5, 30: 0.7266000000000002}
Computation time: 42.74
```

In [14]:

```
#find the lowest error
print("Averaged errors: ", errors)
min_index = np.argmin(errors)
lowest_error = errors[min_index]
best_k = ks[min_index]

print(min_index, lowest_error, best_k)

#plotting (x,y)
plt.plot(ks, errors, 'ro')
plt.show()
```

```
Averaged errors:  [0.7344000000000005, 0.7626000000000017, 0.75040
000000000007, 0.7267999999999989, 0.7256000000000002, 0.7198, 0.72
49999999999998, 0.7209999999999997, 0.7241999999999995, 0.7266000
0000000002]
5 0.7198 10
```



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) According to the k-fold CV, the best of the tested k values was k=10. It makes sense that the best k-value would not be excessively high, since pixels should be closest to the ones in their immediate vicinity. Higher k-values would be counterintuitive because this would mean that pixels are dependent on larger surrounding areas. In the case of blurred or averaged images, this might make sense, but not in general.

(2) The cross-validation error for this value of k was 0.7198.

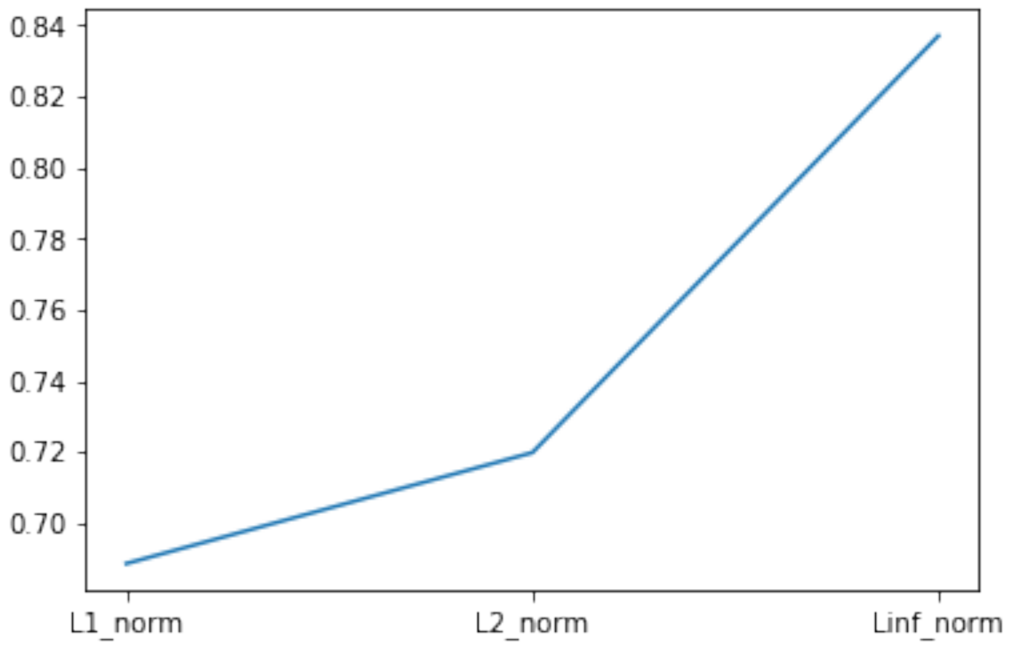
Optimizing the norm

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 [17]:

```
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_errors = []
for norm in norms:
    norm_errors.append(run_kfold(num_folds, kval=best_k, norm=norm))

#find the best norm
min_index = np.argmin(norm_errors)
min_error = norm_errors[min_index]
best_norm = norms[min_index]
print(min_index, min_error, best_norm)

#plotting
data = {'L1_norm': norm_errors[0], 'L2_norm': norm_errors[1], 'Linf_norm': norm_
errors[2]}
names = list(data.keys())
values = list(data.values())
fig, ax = plt.subplots()
ax.plot(names, values)

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

```
CV errors:  [0.7109999999999997, 0.6879999999999994, 0.68000000000
000005, 0.6770000000000005, 0.6870000000000006]
Avg error:  0.6886
CV errors:  [0.7349999999999999, 0.7039999999999996, 0.72399999999
999998, 0.7159999999999997, 0.7199999999999997]
Avg error:  0.7198
CV errors:  [0.8299999999999996, 0.8359999999999997, 0.84599999999
999997, 0.8369999999999997, 0.8359999999999997]
Avg error:  0.837
0 0.6886 <function <lambda> at 0x109661a60>
Computation time: 830.72
```



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) The L1-norm gives the best cross-validation error.

(2) For k=10 and L1 norm, the cross-validaiton error 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 [18]:

```
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.
# ================================================================ #
# Declare an instance of the knn class.
knn = KNN()

knn.train(X=X_train, y=y_train)
dists = knn.compute_distances(X=X_test, norm=best_norm)

y_pred = knn.predict_labels(dists, k=best_k)

total_samples = len(y_pred)
num_incorrect = 0
i = 0
for val in y_pred:
    if(val != y_test[i]):
        num_incorrect+=1
    i+=1

error = num_incorrect/float(total_samples)

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

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

```
Error rate achieved: 0.722
```

Question:

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

In [19]:

```
error_diff = abs(error-norm_errors[1])
print("Improvement in error rate: ", error_diff)
```

```
Improvement in error rate:  0.0022
```

Answer:

The error rate improved by 0.0022 after using a hyperparameter sweep to select k=10 and L1 norm.

In [ ]:

```
import numpy as np
import pdb
import math

"""
This code was based off of code from cs231n at Stanford University, and modified
for ece239as at UCLA.
"""

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].

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

        # ================================================================ #
        # 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.
    """
    m = num_test = X.shape[0]
    n = 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.
    # ================================================================ #
    #X_test has shape (m, d)
    #X_train has shape (n, d)

    # (x-y)^2 = x^2 + y^2 -2xy
    #axis=1 refers to columns
    x2 = np.sum(X**2, axis=1).reshape(m, 1)
    y2 = np.sum(self.X_train**2, axis=1).reshape(1,n)
    xy = X.dot(self.X_train.T) # (m,n)


    dists = np.sqrt(x2 + y2 -2*xy)

    # ================================================================ #
    # 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.
      # ================================================================ #

      #get indices corresponding to increasing sorting order
      sorted_indices = np.argsort(dists[i])
      for kval in range(0, k):
        closest_y.append(self.y_train[sorted_indices[kval]])

      #figure out most common value
      most_freq_val = np.bincount(closest_y).argmax()

      y_pred[i] = most_freq_val
      # ================================================================ #
      # END YOUR CODE HERE
      # ================================================================ #

    return y_pred
```

This is the svm workbook for ECE 239AS Assignment #2

Please follow the notebook linearly to implement a linear support vector machine.

Please print out the workbook entirely when completed.

We thank Serena Yeung & Justin Johnson for permission to use code written for the CS 231n class (cs231n.stanford.edu). These are the functions in the cs231n folders and includes code to preprocess and show the images. The classifiers used are based off of code prepared for CS 231n as well.

The goal of this workbook is to give you experience with training an SVM classifier via gradient descent.

Importing libraries and data setup

In [58]:

```
import numpy as np # for doing most of our calculations
import matplotlib.pyplot as plt# for plotting
from cs231n.data_utils import load_CIFAR10 # function to load the CIFAR-10 datas
et.
import pdb

# 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-ipytho
n
%load_ext autoreload
%autoreload 2
```

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

In [59]:

```
# Set the path to the CIFAR-10 data
import os
cur_dir = os.getcwd()
cifar10_dir = cur_dir+'/cs231n/datasets/cifar-10-batches-py'
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 [60]:

```
# 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 [61]:

```python
# Split the data into train, val, and test sets. In addition we will
# create a small development set as a subset of the training data;
# we can use this for development so our code runs faster.
num_training = 49000
num_validation = 1000
num_test = 1000
num_dev = 500

# Our validation set will be num_validation points from the original
# training set.
mask = range(num_training, num_training + num_validation)
X_val = X_train[mask]
y_val = y_train[mask]

# Our training set will be the first num_train points from the original
# training set.
mask = range(num_training)
X_train = X_train[mask]
y_train = y_train[mask]

# We will also make a development set, which is a small subset of
# the training set.
mask = np.random.choice(num_training, num_dev, replace=False)
X_dev = X_train[mask]
y_dev = y_train[mask]

# We use the first num_test points of the original test set as our
# test set.
mask = range(num_test)
X_test = X_test[mask]
y_test = y_test[mask]

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, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
Dev data shape:  (500, 32, 32, 3)
Dev labels shape:  (500,)
```

In [62]:

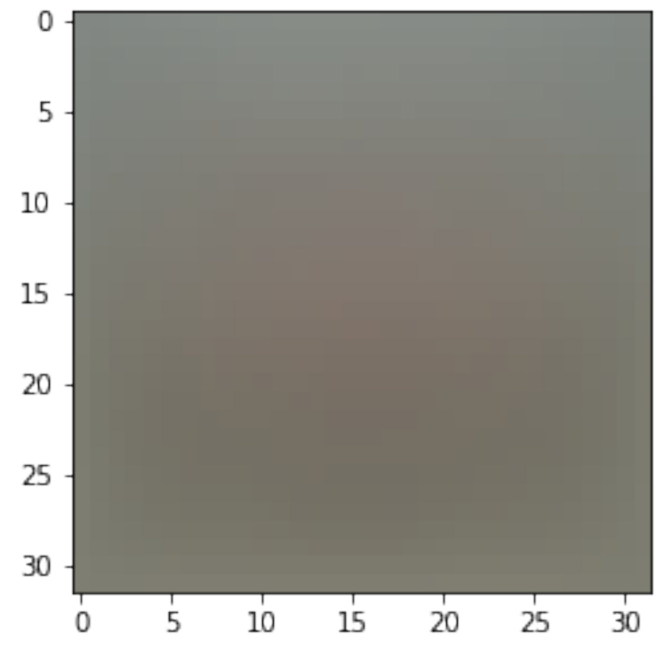
```
# Preprocessing: reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_val = np.reshape(X_val, (X_val.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

# As a sanity check, print out the shapes of the data
print('Training data shape: ', X_train.shape)
print('Validation data shape: ', X_val.shape)
print('Test data shape: ', X_test.shape)
print('dev data shape: ', X_dev.shape)
```

```
Training data shape:  (49000, 3072)
Validation data shape:  (1000, 3072)
Test data shape:  (1000, 3072)
dev data shape:  (500, 3072)
```

In [63]:

```
# Preprocessing: subtract the mean image
# first: compute the image mean based on the training data
mean_image = np.mean(X_train, axis=0)
print(mean_image[:10]) # print a few of the elements
plt.figure(figsize=(4,4))
plt.imshow(mean_image.reshape((32,32,3)).astype('uint8')) # visualize the mean i
mage
plt.show()
```

```
[ 130.64189796  135.98173469  132.47391837  130.05569388  135.348040
82
  131.75402041  130.96055102  136.14328571  132.47636735  131.484673
47]
```



In [64]:

```
# second: subtract the mean image from train and test data
X_train -= mean_image
X_val -= mean_image
X_test -= mean_image
X_dev -= mean_image
```

In [65]:

```
# third: append the bias dimension of ones (i.e. bias trick) so that our SVM
# only has to worry about optimizing a single weight matrix W.
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))])

print(X_train.shape, X_val.shape, X_test.shape, X_dev.shape)
```

```
(49000, 3073) (1000, 3073) (1000, 3073) (500, 3073)
```

Question:

(1) For the SVM, we perform mean-subtraction on the data. However, for the KNN notebook, we did not. Why?

Answer:

(1) Mean subtraction is a method of feature scaling that would not have an effect for K-NN. Given some "mean vector", subtracting this from every data point within our training set would correspondingly shift everything by the same magnitude and direction. The nearest neighbors would stay the same, and the outputs of KNN before and after mean subtraction would be the same. Therefore, it would be an unnecessary extra computation step.

Training an SVM

The following cells will take you through building an SVM. 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 [66]:

```
from nndl.svm import SVM
```

In [67]:

```
# Declare an instance of the SVM class.
# Weights are initialized to a random value.
# Note, to keep people's initial solutions consistent, we are going to use a ran
dom seed.

np.random.seed(1)

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

svm = SVM(dims=[num_classes, num_features])
```

```
3073 10
```

SVM loss

In [68]:

```
## Implement the loss function for in the SVM class(nndl/svm.py), svm.loss()

loss = svm.loss(X_train, y_train)
print('The training set loss is {}.'.format(loss))

# If you implemented the loss correctly, it should be 15569.98
```

```
The training set loss is 15569.97791541019.
```

SVM gradient

In [69]:

```
## Calculate the gradient of the SVM class.
# For convenience, we'll write one function that computes the loss
#   and gradient together. Please modify svm.loss_and_grad(X, y).
# You may copy and paste your loss code from svm.loss() here, and then
#   use the appropriate intermediate values to calculate the gradient.

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

# Compare your gradient to a numerical gradient check.
# You should see relative gradient errors on the order of 1e-07 or less if you i
mplemented the gradient correctly.
svm.grad_check_sparse(X_dev, y_dev, grad)
```

```
numerical: -14.486718 analytic: -14.486717, relative error: 3.426800
e-08
numerical: -4.304328 analytic: -4.304329, relative error: 3.317946e-
08
numerical: -1.540845 analytic: -1.540846, relative error: 2.005298e-
07
numerical: 6.694719 analytic: 6.694720, relative error: 4.180430e-08
numerical: -0.745168 analytic: -0.745169, relative error: 7.002595e-
07
numerical: 11.416845 analytic: 11.416845, relative error: 4.692967e-
09
numerical: 7.342247 analytic: 7.342248, relative error: 4.025915e-08
numerical: -6.535749 analytic: -6.535748, relative error: 6.080351e-
08
numerical: -1.621508 analytic: -1.621507, relative error: 3.155933e-
07
numerical: -13.477379 analytic: -13.477380, relative error: 4.728766
e-08
```

A vectorized version of SVM

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

In [70]:

```
import time
```

In [71]:

```
## Implement svm.fast_loss_and_grad which calculates the loss and gradient
#    WITHOUT using any for loops.

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

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

# The losses should match but your vectorized implementation should be much fast
er.
print('difference in loss / grad: {} / {}'.format(loss - loss_vectorized, np.lin
alg.norm(grad - grad_vectorized)))

# You should notice a speedup with the same output, i.e., differences on the ord
er of 1e-12
```

```
Normal loss / grad_norm: 15807.88880779353 / 2229.005182658606 compu
ted in 0.02430582046508789s
Vectorized loss / grad: 15807.88880779355 / 2229.005182658606 comput
ed in 0.008964061737060547s
difference in loss / grad: -2.000888343900442e-11 / 7.17282901881172
2e-12
```

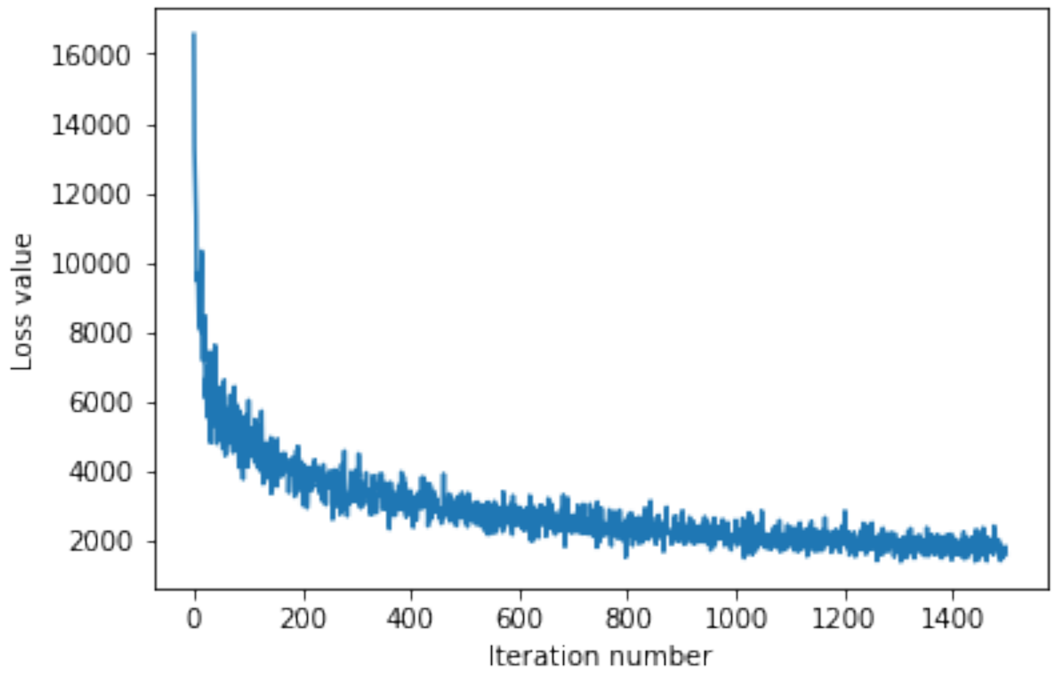
Stochastic gradient descent

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).

In [72]:

```
# Implement svm.train() by filling in the code to extract a batch of data
# and perform the gradient step.

tic = time.time()
loss_hist = svm.train(X_train, y_train, learning_rate=5e-4,
                      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 16557.38000190916
iteration 100 / 1500: loss 4701.089451272714
iteration 200 / 1500: loss 4017.333137942788
iteration 300 / 1500: loss 3681.9226471953616
iteration 400 / 1500: loss 2732.6164373988995
iteration 500 / 1500: loss 2786.6378424645054
iteration 600 / 1500: loss 2837.035784278267
iteration 700 / 1500: loss 2206.234868739932
iteration 800 / 1500: loss 2269.0388241169803
iteration 900 / 1500: loss 2543.23781538592
iteration 1000 / 1500: loss 2566.692135726825
iteration 1100 / 1500: loss 2182.068905905163
iteration 1200 / 1500: loss 1861.1182244250458
iteration 1300 / 1500: loss 1982.901385852825
iteration 1400 / 1500: loss 1927.520415858212
That took 6.337159156799316s
```



Evaluate the performance of the trained SVM on the validation data.

In [73]:

```
## Implement svm.predict() and use it to compute the training and testing error.

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

```
[6 1 9 ..., 2 1 9]
training accuracy: 0.28530612244897957
validation accuracy: 0.3
```

Optimize the SVM

Note, to make things faster and simpler, we won't do k-fold cross-validation, but will only optimize the hyperparameters on the validation dataset (X_val, y_val).

In [74]:

```
# ================================================================= #
# YOUR CODE HERE:
#   Train the SVM 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.
#   Note: You do not need to modify SVM class for this section
# ================================================================= #
def sweep_lr(num_iters):
    learning_rates = [1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1e0, 1e1, 1e2, 1e3,
                     1e4, 1e5, 1e6]

    results = {}
    for lr in learning_rates:
        np.random.seed(1)

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

        svm = SVM(dims=[num_classes, num_features])

        loss_hist = svm.train(X_train, y_train, learning_rate=lr,
                             num_iters=num_iters, verbose=False)

        y_val_pred = svm.predict(X_val)
        val_accuracy = np.mean(np.equal(y_val, y_val_pred))
        results[lr] = val_accuracy

    return results


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

Learning rates were chosen based on order of magnitude (sweeping from 10^-6 to 10-6).

For max_iters=1500, the optimal learning rate is 1.

It's possible that with too small of a learning rate and a num_iter value that is too small, gradient descent will not converge to the global min. This is why it makes sense to sweep values for max_iters until convergence, or to add some epsilon term that determines when to exit gradient descent. The max_iters sweep below was purely experimental.

In [75]:

```
#Find best learning rate and try a couple number max_iters
max_iters = [1500, 2500, 3500]
for max_iter in max_iters:
    print("Testing for max_iter=", max_iter,"...")
    results = sweep_lr(max_iter)
    max_acc = 0
    best_lr = -1
    for key, value in results.items():
        if(value > max_acc):
            max_acc = value
            best_lr = key

    print(results)
    print("Best learning rate is: ", best_lr)
    print("Accuracy is: ", max_acc )
```

```
Testing for max_iter= 1500 ...
{1e-06: 0.13300000000000001, 1e-05: 0.22, 0.0001: 0.2710000000000000
2, 0.001: 0.28499999999999998, 0.01: 0.29299999999999998, 0.1: 0.298
99999999999999, 1.0: 0.315, 10.0: 0.30299999999999999, 100.0: 0.2849
9999999999998, 1000.0: 0.28399999999999997, 10000.0: 0.2959999999999
9999, 100000.0: 0.311, 1000000.0: 0.311}
Best learning rate is:  1.0
Accuracy is:  0.315
Testing for max_iter= 2500 ...
{1e-06: 0.155, 1e-05: 0.23699999999999999, 0.0001: 0.281000000000000
03, 0.001: 0.25800000000000001, 0.01: 0.29499999999999998, 0.1: 0.29
499999999999998, 1.0: 0.29199999999999998, 10.0: 0.29099999999999998
, 100.0: 0.29799999999999999, 1000.0: 0.29399999999999998, 10000.0:
0.29899999999999999, 100000.0: 0.308, 1000000.0: 0.308}
Best learning rate is:  100000.0
Accuracy is:  0.308
Testing for max_iter= 3500 ...
{1e-06: 0.16800000000000001, 1e-05: 0.24099999999999999, 0.0001: 0.2
7500000000000002, 0.001: 0.29999999999999999, 0.01: 0.34300000000000
003, 0.1: 0.35699999999999998, 1.0: 0.30299999999999999, 10.0: 0.320
00000000000001, 100.0: 0.307, 1000.0: 0.34200000000000003, 10000.0:
0.32400000000000001, 100000.0: 0.312, 1000000.0: 0.312}
Best learning rate is:  0.1
Accuracy is:  0.357
```

In [ ]:

```
import numpy as np
import pdb

"""
This code was based off of code from cs231n at Stanford University, and modified
for ece239as at UCLA.
"""
class SVM(object):

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

  def init_weights(self, dims):
    """
        Initializes the weight matrix of the SVM.  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)

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

    # compute the loss and the gradient
    num_classes = self.W.shape[0]
    num_train = X.shape[0]
    loss = 0.0

    # ================================================================ #
    # YOUR CODE HERE:
    #   Calculate the normalized SVM loss, and store it as '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 np.arange(num_train):
      predictions = X[i].dot(self.W.T)
      gnd_truth = predictions[y[i]]
      counter = 0

      for j in range(0, num_classes):
        #j != y from summation
        if(j==y[i]):
          continue

        #margin calculation
        w = predictions[j] - gnd_truth + 1
        if(w > 0):
          counter+=1
          loss += w



    loss /= num_train # get mean


    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.
      """

    # compute the loss and the gradient

    num_classes = self.W.shape[0]
    num_train = X.shape[0]
    loss = 0.0
    grad = np.zeros_like(self.W)
#    print(self.W.shape[1])
    for i in np.arange(num_train):
    # ================================================================ #
    # YOUR CODE HERE:
    #   Calculate the SVM loss and the gradient.  Store the gradient in
    #   the variable grad.
    # ================================================================ #
      predictions = X[i].dot(self.W.T)
      gnd_truth = predictions[y[i]]
      counter = 0
      for j in range(0, num_classes):
        #j != y from summation
        if(j==y[i]):
          continue

        #margin calculation
        w = predictions[j] - gnd_truth + 1
        if(w > 0):
          counter+=1
          grad[j,:] += X[i]
          loss += w


      grad[y[i], :] += (-counter)*X[i]

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


    loss /= num_train
    grad /= num_train

    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) + a
bs(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
    num_train = X.shape[0]
#     print(num_train)
#     print("X shape", X.shape)
    # ================================================================== #
    # YOUR CODE HERE:
    #   Calculate the SVM loss WITHOUT any for loops.
    # ================================================================== #
    predictions = X.dot(self.W.T)
    gnd_truth = predictions[np.arange(num_train), y]
    hinge = np.maximum(0, predictions - gnd_truth[:, np.newaxis] + 1)

    hinge[np.arange(num_train), y] = 0
    loss = np.sum(hinge)

    loss /= num_train
    # ==================================================================== #
    # END YOUR CODE HERE
    # ==================================================================== #



    # ==================================================================== #
    # YOUR CODE HERE:
    #   Calculate the SVM grad WITHOUT any for loops.
    # ==================================================================== #

    """
    -We have 3073 features, and 500 examples. There are 10 classes we can classi
fy into.
    -X is shape (500, 3073).
    -The margins will be shape (500, 10), where each row corresponds to the marg
in for one
    of the 500 examples. Index (i, j) corresponds to the margin of example i for
class j.
    -We need to make a new matrix that has a 1 wherever the margin is > 0, and t
hen sum across the
    columns.
    """
    #Need to look at the margins and if > 0, we're going to need to add 1 to the
counter.
    #Then, we multiply -counter by X

#    print("margin shape", margins.shape)
    counter_matrix = np.zeros(hinge.shape) #row = example, col = class

    #place a 1 wherever hinge loss are > 0.
    counter_matrix[hinge >  0 ] = 1

    #Sum across the classes
    counter = np.sum(counter_matrix, axis=1)

#    print(counter)
 #   print(counter.shape)
#    print(counter_matrix)

    #Need to subtract the hinge loss values from each of the points in the count
er matrix
    ex_idx = np.arange(num_train)
    counter_matrix[ex_idx, y] = -counter #FIX THIS ERROR

    #Take dot product of X and the errors to see how we need to update the weigh
ts
    grad = (X.T.dot(counter_matrix)).T
    grad /= num_train



    # ================================================================ #
    # 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 numbe
r of classes

    self.init_weights(dims=[np.max(y) + 1, X.shape[1]]) # initializes the weight
s 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.
      # ================================================================ #
      rand_indices = np.random.choice(np.arange(num_train), batch_size)
#       print(rand_indices)
```

```
      X_batch = X[rand_indices]

      y_batch = y[rand_indices]
#      print(X.shape)
 #     print(X_batch.shape)
      # ================================================================ #
      # 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.

    N = the number of examples

    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[1])


    # ================================================================ #
    # YOUR CODE HERE:
    #   Predict the labels given the training data with the parameter self.W.
    # ================================================================ #
    #y will be size N.
    # X=(N x D)  W=(C x D) where C = #of classes
    #Result will be (N x C)
    multi_class_preds = (X).dot(self.W.T)

    #find the highest ranking class value among the 10 classes -> columns so axi
s=1
    y_pred = np.argmax(multi_class_preds, axis=1)
    # ================================================================== #
    # END YOUR CODE HERE
    # ================================================================== #

    return y_pred
```

This is the softmax workbook for ECE 239AS Assignment #2

Please follow the notebook linearly to implement a softmax classifier.

Please print out the workbook entirely when completed.

We thank Serena Yeung & Justin Johnson for permission to use code written for the CS 231n class (cs231n.stanford.edu). These are the functions in the cs231n folders and code in the jupyer notebook to preprocess and show the images. The classifiers used are based off of code prepared for CS 231n as well.

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

In [176]:

```
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings('ignore')

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

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

In [177]:

```
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
    # Set the path to the CIFAR-10 data
    import os
    cur_dir = os.getcwd()
    cifar10_dir = cur_dir+'/cs231n/datasets/cifar-10-batches-py'
    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))
    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 [178]:

```
from nndl import Softmax
```

In [179]:

```
# 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 rando
m 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 [180]:

```
## Implement the loss function of the softmax using a for loop over
#  the number of examples

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

In [181]:

```
print(X_train.shape)
print("Loss: ", loss)
```

```
(49000, 3073)
Loss:  2.3277607028
```

Question:

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

Answer:

Without training, the softmax classifier was initialized with random weights. From initializaiton, it essentially acts as a random classifier. We expect a random clssifier to do slightly worse than 50% in terms of accuracy. Assuming that this classifier did slightly worse than 50%, 24500 or more training exmaples would be classified incorrectly. These would take on variable ranges depending on the degree of the misclassification. This error is then normalized by the number of training examples, so for each training example we incur a loss of about 2.32. This number is being inflated by improperly classified points at extreme distances (due to hinge loss penalty). It is not closer to 0 because a correct classification has no bonus, since it is max(0, hinge_penalty).

Softmax gradient

In [182]:

```
## 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 i
mplemented the gradient correctly.
softmax.grad_check_sparse(X_dev, y_dev, grad)
```

```
numerical: 0.684953 analytic: 0.684953, relative error: 5.867046e-09
numerical: 1.251212 analytic: 1.251212, relative error: 2.058286e-08
numerical: 1.225199 analytic: 1.225199, relative error: 1.059550e-08
numerical: 3.011935 analytic: 3.011935, relative error: 6.940156e-09
numerical: 0.903300 analytic: 0.903300, relative error: 4.877627e-08
numerical: -0.221147 analytic: -0.221148, relative error: 2.190131e-
07
numerical: 1.124273 analytic: 1.124273, relative error: 3.960280e-08
numerical: -1.214810 analytic: -1.214810, relative error: 2.342095e-
10
numerical: -1.754822 analytic: -1.754822, relative error: 7.381466e-
09
numerical: -3.909095 analytic: -3.909095, relative error: 9.820295e-
09
```

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 [183]:

```
import time
```

In [184]:

```
## 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'), toc - tic))

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(grad_vectorized, 'fro'), toc - tic))

# The losses should match but your vectorized implementation should be much fast
er.
print('difference in loss / grad: {} /{} '.format(loss - loss_vectorized, np.lin
alg.norm(grad - grad_vectorized)))

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

```
Normal loss / grad_norm: 2.316697106110442 / 321.1638255742149 compu
ted in 0.08814406394958496s
Vectorized loss / grad: 2.316697106110443 / 321.1638255742149 comput
ed in 0.008970022201538086s
difference in loss / grad: -8.881784197001252e-16 /2.160941993728576
7e-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?

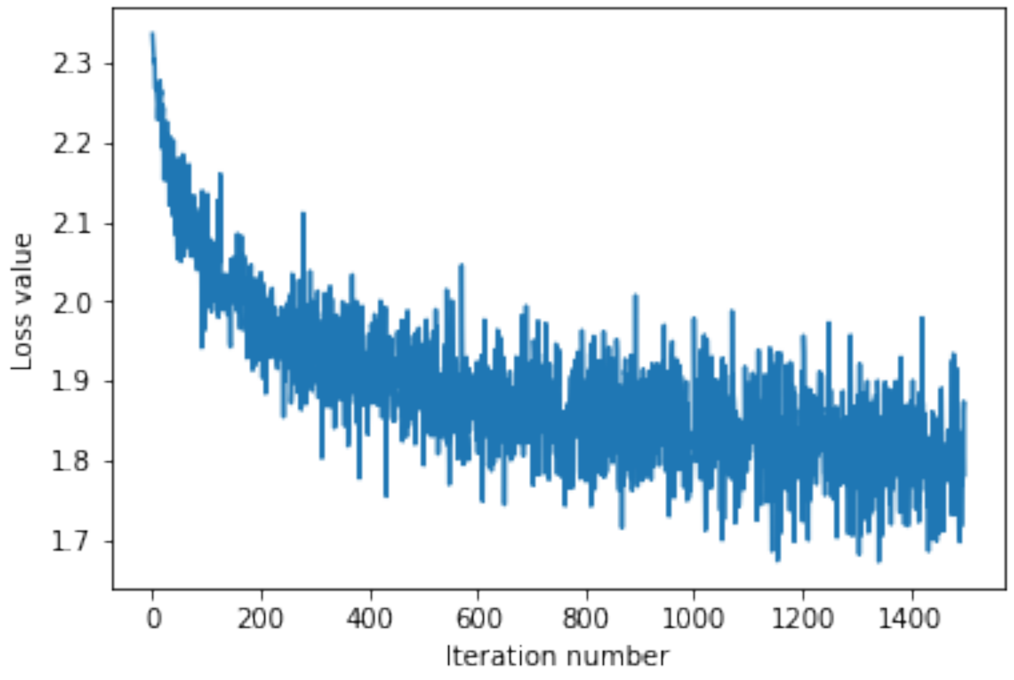
Answer:

The training steps for softmax gradient descent and svm are the same, but the cost functions differ. The SGD steps are the exact same.

In [185]:

```
# 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 2.3365926606637544
iteration 100 / 1500: loss 2.0557222613850827
iteration 200 / 1500: loss 2.0357745120662813
iteration 300 / 1500: loss 1.9813348165609888
iteration 400 / 1500: loss 1.9583142443981612
iteration 500 / 1500: loss 1.8622653073541355
iteration 600 / 1500: loss 1.8532611454359382
iteration 700 / 1500: loss 1.8353062223725827
iteration 800 / 1500: loss 1.8293892468827635
iteration 900 / 1500: loss 1.899215853035748
iteration 1000 / 1500: loss 1.97835035402523
iteration 1100 / 1500: loss 1.8470797913532633
iteration 1200 / 1500: loss 1.8411450268664082
iteration 1300 / 1500: loss 1.7910402495792102
iteration 1400 / 1500: loss 1.8705803029382257
That took 5.789475917816162s
```



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

In [186]:

```
## Implement softmax.predict() and use it to compute the training and testing er
ror.

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: 0.3811428571428571
validation accuracy: 0.398
```

Optimize the softmax classifier

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

In [187]:

```
np.finfo(float).eps
```

Out[187]:

```
2.220446049250313e-16
```

In [188]:

```
# ================================================================ #
# 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.
# ================================================================ #
def sweep_lr(num_iters):
    learning_rates = [1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1e0, 1e1, 1e2, 1e3,
                      1e4, 1e5, 1e6]

    results = {}
    for lr in learning_rates:
        np.random.seed(1)

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

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

        loss_hist = softmax.train(X_train, y_train, learning_rate=lr,
                                  num_iters=num_iters, verbose=False)

        y_val_pred = softmax.predict(X_val)
        val_accuracy = np.mean(np.equal(y_val, y_val_pred))
        results[lr] = val_accuracy

    return results


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

For max_iters=1500, the best learning rate in the set ranging from 1e-6 to 1e6 by orders of magnitude is 1e-6. The associated accuracy is 0.415

In [189]:

```
results = sweep_lr(num_iters=1500)
max_acc = 0
best_lr = -1
for key, value in results.items():
    if(value > max_acc):
        max_acc = value
        best_lr = key

print(results)
print("Best learning rate is: ", best_lr)
print("Accuracy is: ", max_acc )
```

```
{1e-06: 0.41499999999999998, 1e-05: 0.315, 0.0001: 0.283999999999999
97, 0.001: 0.086999999999999994, 0.01: 0.086999999999999994, 0.1: 0.
086999999999999994, 1.0: 0.086999999999999994, 10.0: 0.0869999999999
99994, 100.0: 0.086999999999999994, 1000.0: 0.086999999999999994, 10
000.0: 0.086999999999999994, 100000.0: 0.086999999999999994, 1000000
.0: 0.086999999999999994}
Best learning rate is:  1e-06
Accuracy is:  0.415
```

In [ ]:

```
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.)
    # ================================================================ #



    num_examples = X.shape[0]
    num_classes = self.W.shape[0]
```

```
#    print(num_examples, num_classes)

    #outer summation
    for i in range(0, num_examples):
      a_i = X[i].dot(self.W.T)

      #sum over all
      sigma_j = np.sum(np.exp(a_i))

      # -a_y(i)*X(i) + log(summation)
      probs = lambda idx: np.exp(a_i[idx])/sigma_j
      loss_sum = probs(y[i])#self.get_probs(y[i], a_i, sigma_j)

      #need to maintain negative sign
      loss -= np.log(loss_sum)




    loss /= num_examples

    # ================================================================ #
    # 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.
    # ================================================================ #
    num_examples = X.shape[0]
    num_classes = self.W.shape[0]
#    print(num_examples, num_classes)

    #outer summation
    """
    for i in range(0, num_examples):

      a_i = X[i].dot(self.W.T)

      #sum over all
      sigma_j = np.sum(np.exp(a_i))

      # -a_y(i)*X(i) + log(summation)
      probs = lambda idx: np.exp(a_i[idx])/sigma_j
      loss_sum = probs(y[i])

      #need to maintain negative sign
      loss -= np.log(loss_sum)

      #Gradient
      for c in range(0, num_classes):
        prob_class = probs(c)
        indicator = 0
        if(c == y[i]):
          indicator = 1

        grad_update = (prob_class - indicator)*X[i]

        grad[c, :] += grad_update


    """

    for i in range(len(y)):
        score = self.W.dot(X[i,:].T)
        score -= np.max(score)
        true_score = score[y[i]]
        t_loss = np.exp(true_score) / np.sum(np.exp(score))
        loss -= np.log(t_loss)
        for j in range(self.W.shape[0]):
            indicator = 0
            if (j == y[i]):
                indicator = 1

            grad_update = (np.exp(score[j])/np.sum(np.exp(score)) - indicator)*X
[i]
            grad[j, :] += grad_update

    loss /= num_examples
    grad /= num_examples

    # ================================================================ #
    # 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) + a
bs(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.
    # ================================================================ #
    num_classes = self.W.shape[0]
    num_features = self.W.shape[0]
    num_train = X.shape[0]

    scores = X.dot(self.W.T)
    scores_T = scores.T
    mask = np.zeros(shape = (num_classes, num_train))
    mask[y, range(num_train)] = 1

    gnd_truth = np.sum(np.multiply(mask, scores_T), axis=0)

    total_scores = np.sum(np.exp(scores_T).T, axis=1)
    log_inner = np.log(np.exp(gnd_truth) / total_scores)
    sigma = np.sum(log_inner, axis = 0)
    loss = -sigma
    loss /= num_train


    sigma_same_dims = np.sum(np.exp(scores), axis=1, keepdims=True)
    total_probs = np.exp(scores)/sigma_same_dims
    total_probs[range(X.shape[0]), y] -= 1
    grad = (X.T).dot(total_probs)
    grad /= num_train
    grad = grad.T


    # ================================================================ #
    # 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 numbe
r of classes

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

      # ================================================================ #
      rand_indices = np.random.choice(np.arange(num_train), batch_size)
      X_batch = X[rand_indices]
      y_batch = y[rand_indices]

      # ================================================================ #
      # 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[1])
    # ================================================================ #
    # YOUR CODE HERE:
    #   Predict the labels given the training data.
    # ================================================================ #
    #y will be size N.
    # X=(N x D)  W=(C x D) where C = #of classes
    #Result will be (N x C)
    multi_class_preds = (X).dot(self.W.T)

    #find the highest ranking class value among the 10 classes -> columns so axi
s=1
    y_pred = np.argmax(multi_class_preds, axis=1)
    # ==================================================================== #
    # END YOUR CODE HERE
    # ==================================================================== #

    return y_pred
```