knn

January 27, 2020

0.1 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.

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.

0.2 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 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
```

In [2]:
```
# Set the path to the CIFAR-10 data
cifar10_dir = '/Users/ApplePro/Desktop/School/GradSchool/Courses/C247/hw2/cifar-10-bat
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', 'truc
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)
```

1 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)
```

1.1 Questions

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

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

1.2 Answers

(1) The function caches the entire dataset in the train and the test groups.

(2) Pros: Fast and simple O(1). Cons: Memory intensive because we need to store all the training data.

1.3 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: 30.84943389892578
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

1.3.1 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 for loop
        # 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): {}'.
```

```
Time to run code: 0.11865115165710449
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.

1.3.2 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

        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.
        # ================================================================ #
        yPredicted=knn.predict_labels(dists_L2_vectorized,1)
        error = np.count_nonzero(y_test-yPredicted)/float(len(y_test))
        # ================================================================ #
        # 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%.

2 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.

2.0.1 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.
         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]
         # ================================================================ #
         #print(X_train.shape)
```

```
        for i in range(num_folds):
            X_train_folds.append(X_train[i*1000:(i+1)*1000,:])
            y_train_folds.append(y_train[i*1000:(i+1)*1000])

        pass
        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #
        #print(y_train.shape)
        #print(X_train.shape)
        #print(y_train_folds[0].shape)
        #print(y_train_folds[0])
        #t=np.concatenate((y_train_folds[0],y_train_folds[1],0)

        #t=map(lambda x: y_train_folds[x], np.setdiff1d(np.arange(num_folds), 0))
        print(y_train_folds[0].shape)
```

```
(1000,)
```

2.0.2 Optimizing the number of nearest neighbors hyperparameter.

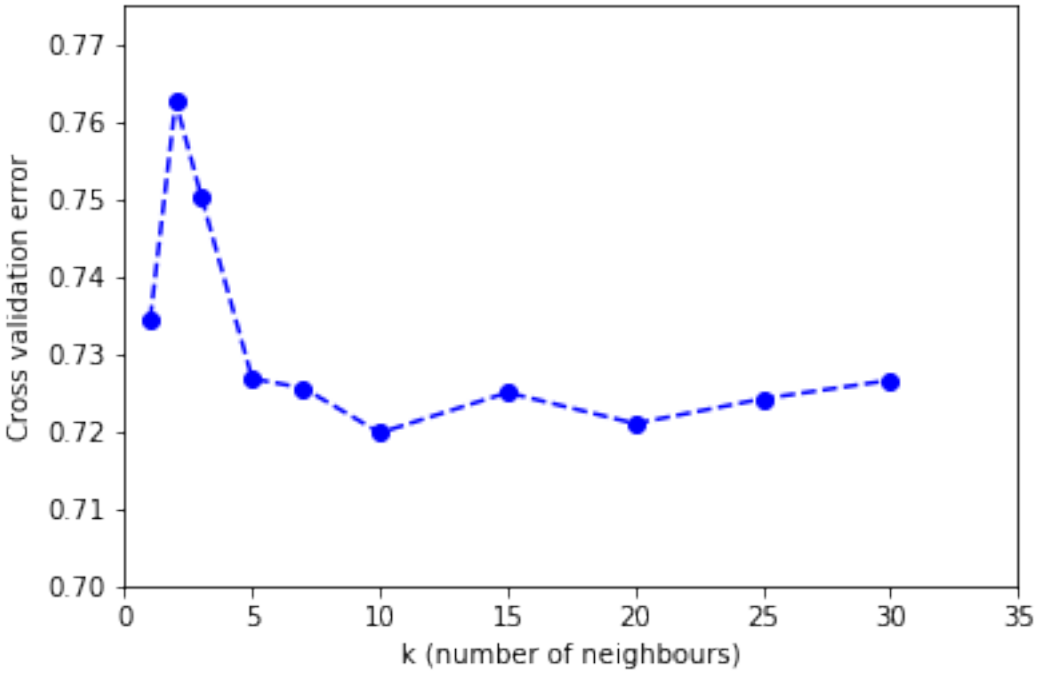
In this section, we select different numbers of nearest neighbors and assess which one has the lowest k-fold cross validation error.

In [11]:
```
time_start =time.time()

ks = [1, 2, 3, 5, 7, 10, 15, 20, 25, 30]
errorAvg=[]
# ================================================================ #
# 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.
# ================================================================ #
num_test_fold = 1000

for k in ks:
    error2 = 0
    error1 = 0
    for i in range(num_folds):
        knn = KNN()

        X_test_kFold = X_train_folds[i]
        y_test_kFold = y_train_folds[i]
```

```
        X_train_kFold = []
        y_train_kFold = []
        #mask = list(range(num_training))
        #del mask[i*1000:(i+1)*1000]

        for j in range(num_folds):
            if i != j:
                X_train_kFold.extend(X_train_folds[j])
                y_train_kFold.extend(y_train_folds[j])

        X_train_kFold = np.array(X_train_kFold)
        y_train_kFold = np.array(y_train_kFold)

        #X_train_kFold = X_train[mask]
        #y_train_kFold = y_train[mask]

        knn.train(X=X_train_kFold, y=y_train_kFold)

        dists_fold = knn.compute_L2_distances_vectorized(X_test_kFold)
        y_est_fold = knn.predict_labels(dists_fold,k)
#         y_diff_fold = (y_test_kFold - y_est_fold)

        num_correct = np.sum(y_test_kFold == y_est_fold)
        #num_incorrect_fold = np.count_nonzero(y_diff_fold)
        #error1 = num_incorrect_fold / num_test_fold

        error1 = (num_test_fold - num_correct)/num_test_fold

        error2 += error1
    errorAvg.append(error2/num_folds)



for j in np.arange(len(errorAvg)):
    print(errorAvg[j],ks[j])

x_index = ks
y_value = errorAvg
plt.plot(x_index, y_value, 'bo--')
plt.axis([0, 35, 0.7, 0.775])
plt.xlabel('k (number of neighbours)')
plt.ylabel('Cross validation error')
plt.show()
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

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

```
0.7344 1
0.7626000000000002 2
0.7504000000000001 3
0.7267999999999999 5
0.7256 7
0.7198 10
0.725 15
0.721 20
0.7242 25
0.7266 30
```




```
Computation time: 18.49
```

2.1 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?

2.2 Answers:

(1) k=10

(2) 0.7198

2.2.1 Optimizing the norm

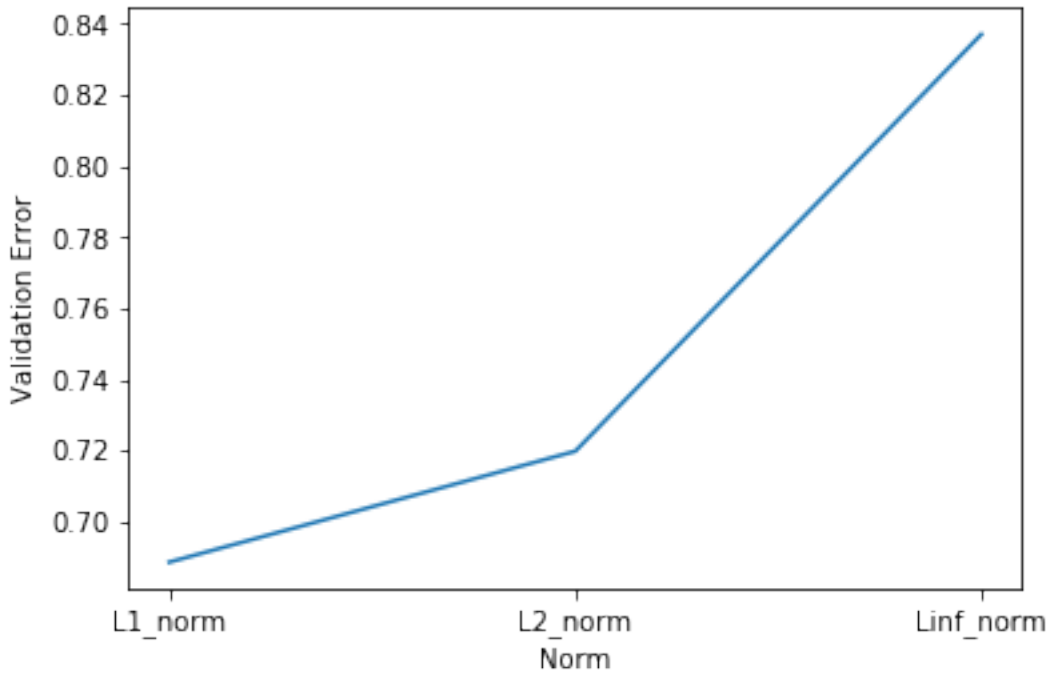
Next, we test three different norms (the 1, 2, and infinity norms) and see which distance metric results in the best cross-validation performance.

In [12]:
```
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.
# ============================================================ #

error = []
for L in norms:
    error2 = 0
    error1 = 0
    for i in range(num_folds):
        print(str(L),i)
        knn = KNN()

        X_test_kFold = X_train_folds[i]
        y_test_kFold = y_train_folds[i]

        X_train_kFold = []
        y_train_kFold = []
        #mask = list(range(num_training))
        #del mask[i*1000:(i+1)*1000]

        for j in range(num_folds):
            if i != j:
```

```
                        X_train_kFold.extend(X_train_folds[j])
                        y_train_kFold.extend(y_train_folds[j])

                X_train_kFold = np.array(X_train_kFold)
                y_train_kFold = np.array(y_train_kFold)

                #X_train_kFold = X_train[mask]
                #y_train_kFold = y_train[mask]

                knn.train(X=X_train_kFold, y=y_train_kFold)

                dists_fold = knn.compute_distances(X_test_kFold,L)
                y_est_fold = knn.predict_labels(dists_fold,10)
                y_diff_fold = (y_test_kFold - y_est_fold)

                num_correct = np.sum(y_test_kFold == y_est_fold)
                #num_incorrect_fold = np.count_nonzero(y_diff_fold)
                #error1 = num_incorrect_fold / num_test_fold

                error1 = (num_test_fold - num_correct)/num_test_fold

                error2 += error1

            error.append(error2/5)

        print(error)

        plt.figure()
        plt.plot(error)
        plt.xlabel('Norm')
        plt.ylabel('Validation Error')
        plt.xticks(np.arange(3), ['L1_norm', 'L2_norm', 'Linf_norm'])
        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #
        print('Computation time: %.2f'%(time.time()-time_start))
```

```
<function <lambda> at 0x122f33620> 0
<function <lambda> at 0x122f33620> 1
<function <lambda> at 0x122f33620> 2
<function <lambda> at 0x122f33620> 3
<function <lambda> at 0x122f33620> 4
<function <lambda> at 0x11c20aea0> 0
<function <lambda> at 0x11c20aea0> 1
<function <lambda> at 0x11c20aea0> 2
<function <lambda> at 0x11c20aea0> 3
<function <lambda> at 0x11c20aea0> 4
<function <lambda> at 0x122f311e0> 0
```

```
<function <lambda> at 0x122f311e0> 1
<function <lambda> at 0x122f311e0> 2
<function <lambda> at 0x122f311e0> 3
<function <lambda> at 0x122f311e0> 4
[0.6886000000000001, 0.7198, 0.8370000000000001]
Computation time: 642.02
```



2.3 Questions:

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

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

2.4 Answers:

(1) L_1 norm

(2) 0.6886

3 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 [15]: error = 1
         k=10
         num_folds=5
         L1_norm = lambda x: np.linalg.norm(x, ord=1)
         # ================================================================ #
         # YOUR CODE HERE:
         #   Evaluate the testing error of the k-nearest neighbors classifier
         #   for your optimal hyperparameters found by 5-fold cross-validation.
         # ================================================================ #

         knn = KNN()
         knn.train(X=X_train, y=y_train)

         dists_L1=knn.compute_distances(X_test,L1_norm)
         yPredicted=knn.predict_labels(dists_L1,k)
         error = np.count_nonzero(y_test-yPredicted)/float(len(y_test))

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

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

```
Error rate achieved: 0.722
```

3.1 Question:

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

3.2 Answer:

Error rate : 0.722
Improvement: 0.726-0.722= 0.004

4 knn.py

```
In [ ]: import numpy as np
        import pdb

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

        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(X[i]-self.X_train[j])


                # ================================================================ #
                # 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(((X**2).sum(axis=1,keepdims= True ))+ (self.X_train**2).sum(axis=1
        # ================================================================ #
        # 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 = list(self.y_train[np.argsort(dists[i])[:k]])
      y_pred[i] = max(sorted(list(set(closest_y))), key = closest_y.count)


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

    return y_pred
```

svm

January 27, 2020

0.1 This is the svm workbook for ECE C147/C247 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.

0.2 Importing libraries and data setup

In [41]:
```
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 dataset.
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-ipython
%load_ext autoreload
%autoreload 2
```

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

In [42]:
```
# Set the path to the CIFAR-10 data
cifar10_dir = '/Users/ApplePro/Desktop/School/GradSchool/Courses/C247/hw2/cifar-10-ba
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 [43]:
```
# 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', 'tru
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 [44]:
```
# 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)
```

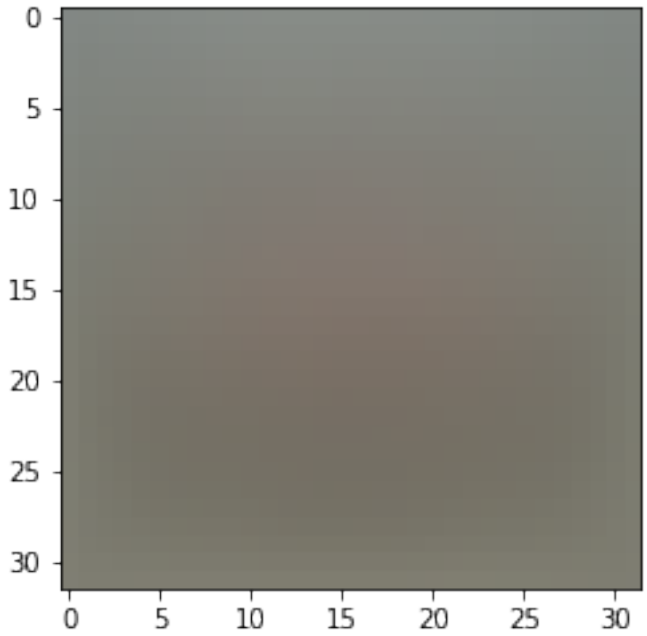
```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
Dev data shape:  (500, 32, 32, 3)
Dev labels shape:  (500,)
```

In [45]:
```
# 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 [46]:
```
# 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 image
plt.show()
```

```
[130.64189796 135.98173469 132.47391837 130.05569388 135.34804082
 131.75402041 130.96055102 136.14328571 132.47636735 131.48467347]
```



In [47]:
```
# 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 [48]:
```
# 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)
```

0.3 Question:

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

0.4 Answer:

(1) knn is not a linear classifier like SVM. Normalize the distribution to zero centered may speed up the calculation in SVM, but knn compares point to point so mean-subtraction will only shift the data in vector space without reducing anything.

0.5 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 [49]:
```
from nndl.svm import SVM
```

In [53]:
```
# 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 random s

np.random.seed(1)

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

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

SVM loss

```
In [57]: ## 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 15569.977915410187.
```

SVM gradient

```
In [59]: ## 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 implem
         svm.grad_check_sparse(X_dev, y_dev, grad)
```

```
numerical: -7.583533 analytic: -7.583533, relative error: 3.738598e-08
numerical: -7.346214 analytic: -7.346214, relative error: 1.069195e-08
numerical: -3.026962 analytic: -3.026962, relative error: 2.825360e-08
numerical: 5.622965 analytic: 5.622965, relative error: 5.122295e-09
numerical: -1.013095 analytic: -1.013096, relative error: 3.579369e-07
numerical: 1.941761 analytic: 1.941762, relative error: 6.872900e-08
numerical: 4.561661 analytic: 4.561661, relative error: 6.833220e-09
numerical: -6.070590 analytic: -6.070590, relative error: 1.701165e-08
numerical: 2.199344 analytic: 2.199343, relative error: 1.396891e-07
numerical: -14.115310 analytic: -14.115310, relative error: 1.337289e-08
```

0.6 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 [61]: import time
```

```
In [70]: ## 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)
toc = time.time()
print('Normal loss / grad_norm: {} / {} computed in {}s'.format(loss, np.linalg.norm(g

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

# The losses should match but your vectorized implementation should be much faster.
print('difference in loss / grad: {} / {}'.format(loss - loss_vectorized, np.linalg.no

# You should notice a speedup with the same output, i.e., differences on the order of
```

```
Normal loss / grad_norm: 14656.132351172842 / 2018.6983390228343 computed in 0.032856941223144
Vectorized loss / grad: 14656.132351172808 / 2018.6983390228343 computed in 0.0038621425628662
difference in loss / grad: 3.456079866737127e-11 / 3.632777079343923e-12
```

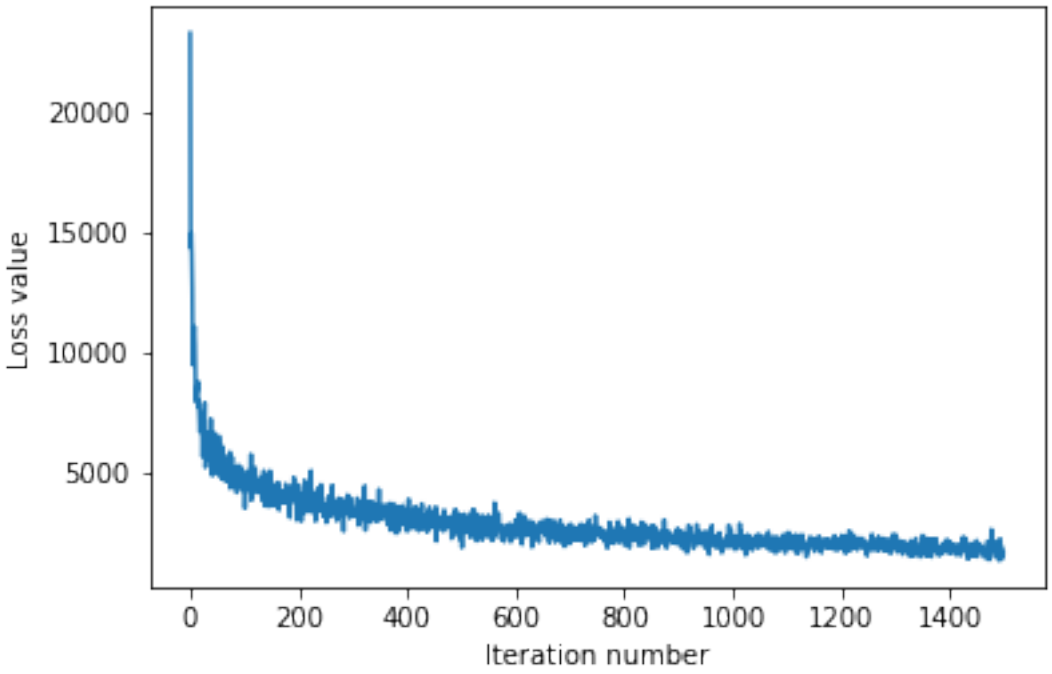
0.7 Stochastic gradient descent

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 [75]:
```
# 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 23298.531266498594
iteration 100 / 1500: loss 4405.409952058734
iteration 200 / 1500: loss 4507.282152274323
iteration 300 / 1500: loss 3419.5478719395205
iteration 400 / 1500: loss 2562.7996515515506
iteration 500 / 1500: loss 2837.187093495071
iteration 600 / 1500: loss 3035.636281513894
iteration 700 / 1500: loss 2402.010168899141
iteration 800 / 1500: loss 2012.0001740308526
```

```
iteration 900 / 1500: loss 2113.8507743765795
iteration 1000 / 1500: loss 2068.423094764378
iteration 1100 / 1500: loss 2260.8503230367
iteration 1200 / 1500: loss 1976.4065776896903
iteration 1300 / 1500: loss 2446.5993445957706
iteration 1400 / 1500: loss 1667.9116041013592
That took 2.003600835800171s
```



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

In [76]:
```
## Implement svm.predict() and use it to compute the training and testing error.

y_train_pred = svm.predict(X_train)
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)), ))
```

```
training accuracy: 0.29567346938775513
validation accuracy: 0.267
```

0.8 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 [77]:
```
# ================================================================ #
# 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
# ================================================================ #
rates = [10**i for i in range(-5,6)]

for rate in rates:
    svm.train(X_train, y_train, learning_rate=rate, num_iters=1500, verbose=False)
    y_val_pred = svm.predict(X_val)
    print('rate:',rate,'validation accuracy: {}'.format(np.mean(np.equal(y_val, y_val
print('-'*20)

Best = [0,0,0] # rate, train accuracy, test accuracy
for rate in np.linspace(0.0001, 0.1,30):
    svm.train(X_train, y_train, learning_rate=rate, num_iters=1500, verbose=False)
    y_val_pred = svm.predict(X_val)

    if np.mean(np.equal(y_val, y_val_pred)) > Best[1]:
        Best[0] = rate
        Best[1] = np.mean(np.equal(y_val, y_val_pred))
    print('rate:',rate,'validation accuracy: {}'.format(np.mean(np.equal(y_val, y_val

print('-'*20)

svm.train(X_train, y_train, learning_rate=Best[0], num_iters=1500, verbose=False)
y_test_pred = svm.predict(X_test)
Best[2] = np.mean(np.equal(y_test, y_test_pred))
print('Best rate is',Best[0], 'Test Set Error Rate is', 1-Best[2])
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

```
rate: 1e-05 validation accuracy: 0.212
rate: 0.0001 validation accuracy: 0.271
rate: 0.001 validation accuracy: 0.304
rate: 0.01 validation accuracy: 0.254
rate: 0.1 validation accuracy: 0.297
rate: 1 validation accuracy: 0.29
rate: 10 validation accuracy: 0.252
rate: 100 validation accuracy: 0.285
```

```
rate: 1000 validation accuracy: 0.278
rate: 10000 validation accuracy: 0.307
rate: 100000 validation accuracy: 0.332
--------------------
rate: 0.0001 validation accuracy: 0.265
rate: 0.0035448275862068964 validation accuracy: 0.28
rate: 0.006989655172413793 validation accuracy: 0.303
rate: 0.010434482758620689 validation accuracy: 0.295
rate: 0.013879310344827585 validation accuracy: 0.304
rate: 0.017324137931034482 validation accuracy: 0.28
rate: 0.02076896551724138 validation accuracy: 0.326
rate: 0.024213793103448275 validation accuracy: 0.26
rate: 0.02765862068965517 validation accuracy: 0.282
rate: 0.031103448275862068 validation accuracy: 0.292
rate: 0.03454827586206897 validation accuracy: 0.296
rate: 0.03799310344827586 validation accuracy: 0.301
rate: 0.04143793103448276 validation accuracy: 0.29
rate: 0.04488275862068966 validation accuracy: 0.237
rate: 0.048327586206896554 validation accuracy: 0.296
rate: 0.05177241379310345 validation accuracy: 0.315
rate: 0.05521724137931035 validation accuracy: 0.325
rate: 0.05866206896551725 validation accuracy: 0.305
rate: 0.06210689655172414 validation accuracy: 0.314
rate: 0.06555172413793103 validation accuracy: 0.325
rate: 0.06899655172413793 validation accuracy: 0.293
rate: 0.07244137931034483 validation accuracy: 0.31
rate: 0.07588620689655172 validation accuracy: 0.275
rate: 0.07933103448275862 validation accuracy: 0.327
rate: 0.08277586206896552 validation accuracy: 0.3
rate: 0.08622068965517242 validation accuracy: 0.308
rate: 0.08966551724137932 validation accuracy: 0.256
rate: 0.0931103448275862 validation accuracy: 0.319
rate: 0.0965551724137931 validation accuracy: 0.317
rate: 0.1 validation accuracy: 0.309
--------------------
Best rate is 0.07933103448275862 Test Set Error Rate is 0.768
```

1 svm.py

```
In [ ]: import numpy as np
        import pdb

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

  for i in np.arange(num_train):

  # ============================================================= #
  # 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.)
  # ============================================================= #
      scores=np.dot(X[i,:],self.W.T)
      correct_class_score=scores[y[i]]
      for j in np.arange(num_classes):
          if y[i] == j:
              continue
          loss+=max(0,1+scores[j]-scores[y[i]])
  loss=loss/num_train
```

```
    # ================================================================ #
    # 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.
        """

    # 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)

    for i in np.arange(num_train):
    # ================================================================ #
    # YOUR CODE HERE:
        #   Calculate the SVM loss and the gradient.  Store the gradient in
    #   the variable grad.
    # ================================================================ #
        scores=np.dot(X[i,:],self.W.T)
        correct_class_score=scores[y[i]]
        for j in np.arange(num_classes):
            if y[i] == j:
                continue
            margin=max(0,1+scores[j]-scores[y[i]])
            if margin >0:
                loss+=margin
                grad[y[i],:] -= X[i,:]
                grad[j,:] += X[i,:]
    loss=loss/num_train
    grad=grad/num_train
    # ================================================================ #
    # 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) + abs(grad
      print('numerical: %f analytic: %f, relative error: %e' % (grad_numerical, grad_an

  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 SVM loss WITHOUT any for loops.
    # ============================================================= #
    #num_train=X.shape[0]
    #scores=X.dot(self.W.T)
    #yi_scores=scores[np.arange(scores.shape[0]),y]
    #margins=np.maximum(0,scores-np.matrix(yi_scores).T+1)
    #margins[np.arange(num_train),y] = 0
    #loss = np.mean(np.sum(margins, axis=1))

    a = X.dot(self.W.T) # sample * class
    num_train = a.shape[0]
    ay = a[range(num_train), y]
    m = np.maximum(0, (a.T - ay).T +1) # - ones_like(a)[range(a.shape[0]), y]
    m[range(a.shape[0]), y] = 0
        # m[range(a.shape[0]), y] = 0
    loss = np.sum(m)/a.shape[0]

    # ============================================================= #
```

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



    # ================================================================ #
    # YOUR CODE HERE:
    #   Calculate the SVM grad WITHOUT any for loops.
    # ================================================================ #
    #binary = margins
    #binary[margins > 0] = 1
    #row_sum = np.sum(binary, axis=1)
    #binary[np.arange(num_train), y] = -row_sum.T
    #grad=np.dot(binary.T,X)
    #grad/=num_train

    m[range(a.shape[0]), y] = 0
    b = m # sample * class
    b[m > 0] = 1
    row_sum = np.sum(b, axis=1)
    b[range(num_train), y] = -row_sum.T
    grad = b.T.dot(X)/num_train # class * feature

    # ================================================================ #
    # 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 c

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

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


  # ========================================================================= #
  # 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 = self.W - grad* learning_rate
        # ========================================================================= #
  # 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 with the parameter self.W.
  # ================================================================ #
  y_pred = np.argmax(X.dot(self.W.T),axis=1)
  # ================================================================ #
  # END YOUR CODE HERE
  # ================================================================ #

  return y_pred
```

softmax

January 27, 2020

0.1 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.

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 [1]:
```
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

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

In [2]:
```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000, num_dev=5
    """
    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 = '/Users/ApplePro/Desktop/School/GradSchool/Courses/C247/hw2/cifar-10
    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,)
```

0.2 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 [37]: from nndl import Softmax
```

```
In [39]: # 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 see

         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 [40]: ## Implement the loss function of the softmax using a for loop over
         #  the number of examples
         print(y_train[0])
         loss = softmax.loss(X_train, y_train)
```

```
6
(49000,)
(49000, 3073)
(10, 3073)
(49000, 10)
```

```
In [41]: print(loss)
```

```
2.3277607028048966
```

0.3 Question:

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

0.4 Answer:

2.3~ [-ln(0.1)] which means one in the ten given classes is what our currentl loss function is calculating. It make sense as the hyperparameters are rondomly chosen from normal distribution (np.random.normal[]). So 1/10 probability of choosing the right class.

Softmax gradient

```
In [48]: ## 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 implem
         softmax.grad_check_sparse(X_dev, y_dev, grad)
```

```
before (10, 500)
After (10, 500)
The end
numerical: -3.099203 analytic: -3.099204, relative error: 1.128370e-08
numerical: 0.799125 analytic: 0.799125, relative error: 4.943867e-08
numerical: -2.286678 analytic: -2.286678, relative error: 4.027054e-08
numerical: -0.913192 analytic: -0.913192, relative error: 1.223424e-08
numerical: 1.609597 analytic: 1.609597, relative error: 7.771982e-10
numerical: -1.768594 analytic: -1.768594, relative error: 4.054308e-08
numerical: 0.704470 analytic: 0.704470, relative error: 3.005293e-09
numerical: -0.608301 analytic: -0.608301, relative error: 5.939298e-08
numerical: 1.674017 analytic: 1.674016, relative error: 5.825551e-08
numerical: 0.056773 analytic: 0.056773, relative error: 4.050249e-07
```

0.5 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 [49]: import time
```

```
In [51]: ## 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(g

         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.lin
```

```
        # The losses should match but your vectorized implementation should be much faster.
        print('difference in loss / grad: {} /{} '.format(loss - loss_vectorized, np.linalg.no

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

```
Normal loss / grad_norm: 2.3380159773413673 / 335.77608489059077 computed in 0.018710136413574
Vectorized loss / grad: 2.3380159773413665 / 335.77608489059077 computed in 0.0025773048400878
difference in loss / grad: 8.881784197001252e-16 /5.006281901851313e-14
```

0.6 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).

0.7 Question:

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

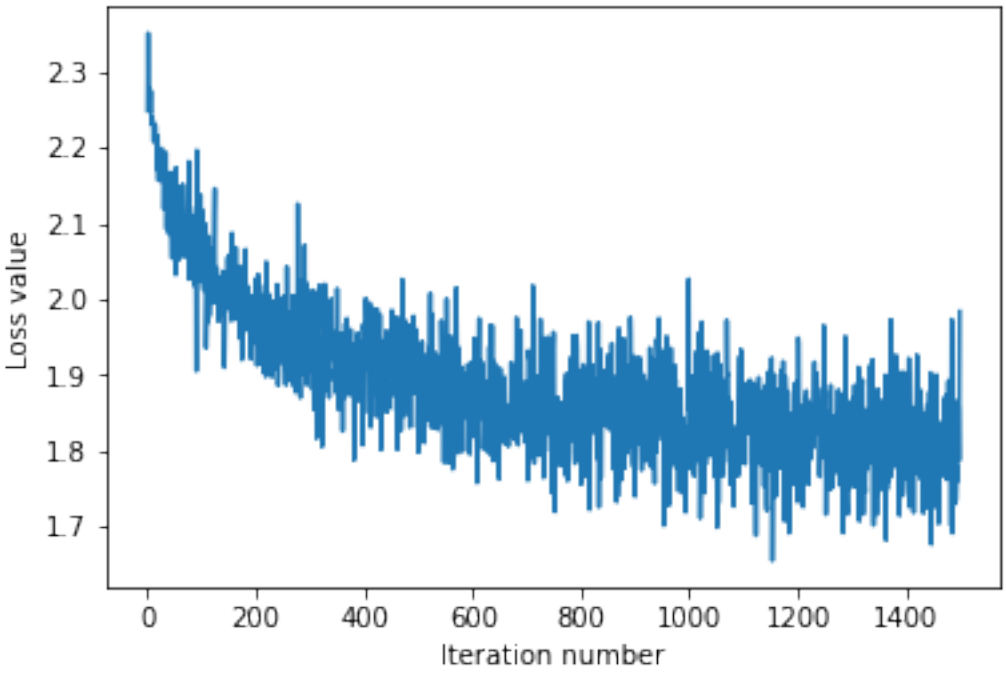
0.8 Answer:

They should differ as they utilize different cost functions.

In [52]:
```
# 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 2.2483650583337553
iteration 100 / 1500: loss 2.1182643187377015
iteration 200 / 1500: loss 1.955351315420126
iteration 300 / 1500: loss 1.9587048512507435
iteration 400 / 1500: loss 1.8495037733293611
iteration 500 / 1500: loss 1.797760075469314
iteration 600 / 1500: loss 1.822558641450844
iteration 700 / 1500: loss 1.8674388252253464
```

```
iteration 800 / 1500: loss 1.7553670865183204
iteration 900 / 1500: loss 1.7396877079711515
iteration 1000 / 1500: loss 1.85268364487155
iteration 1100 / 1500: loss 1.7854944930427314
iteration 1200 / 1500: loss 1.7322776347927484
iteration 1300 / 1500: loss 1.838509909050927
iteration 1400 / 1500: loss 1.8741836589582033
That took 3.0625500679016113s
```



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

In [53]:
```
## 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: 0.38142857142857145
validation accuracy: 0.394
```

0.9 Optimize the softmax classifier

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

In [54]:
```
np.finfo(float).eps
```

Out[54]:
```
2.220446049250313e-16
```

In [55]:
```
# ================================================================ #
# 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.
# ================================================================ #
rates = [10**i for i in range(-12,-1)]

Best = [0,0,0] # rate, train accuracy, test accuracy
for rate in rates:
    softmax.train(X_train, y_train, learning_rate=rate, num_iters=1500, verbose=False)
    y_val_pred = softmax.predict(X_val)

    if np.mean(np.equal(y_val, y_val_pred)) > Best[1]:
        Best[0] = rate
        Best[1] = np.mean(np.equal(y_val, y_val_pred))
    print('rate:',rate,'validation accuracy: {}'.format(np.mean(np.equal(y_val, y_val_

print('-'*20)

softmax.train(X_train, y_train, learning_rate=Best[0], num_iters=1500, verbose=False)
y_test_pred = softmax.predict(X_test)
Best[2] = np.mean(np.equal(y_test, y_test_pred))
print('Best rate is',Best[0], 'Test Set Error Rate is', 1-Best[2])
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

```
rate: 1e-12 validation accuracy: 0.133
rate: 1e-11 validation accuracy: 0.059
rate: 1e-10 validation accuracy: 0.134
rate: 1e-09 validation accuracy: 0.154
rate: 1e-08 validation accuracy: 0.301
rate: 1e-07 validation accuracy: 0.394
rate: 1e-06 validation accuracy: 0.412
rate: 1e-05 validation accuracy: 0.301
rate: 0.0001 validation accuracy: 0.272


/Users/ApplePro/Desktop/School/GradSchool/Courses/C247/hw2/nndl/softmax.py:142: RuntimeWarning
```

```
  loss = np.sum(  -np.log(a_exp[np.arange(a.shape[0]), y] / np.sum(a_exp, axis = 1)) )


rate: 0.001 validation accuracy: 0.285
rate: 0.01 validation accuracy: 0.268
--------------------
Best rate is 1e-06 Test Set Error Rate is 0.615
```

1 softmax.py

```
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.)
    # ============================================================ #
    a = X.dot(self.W.T)
    #print(y.shape)
    #print(X.shape)
    #print(self.W.shape)
    #print(a.shape)
    i = 0
    for row in a:
        row -= np.max(row) #To avoid overflow
        loss += - np.log( np.exp(row[y[i]]) /sum(np.exp(row)) )
        i = i+1

    loss = loss/y.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.
    # ============================================================ #
    # dL/dW = dL/da*da/dW
    # dL/da = Score - 1(y == i)
    # da/dW = X'
    a = self.W.dot(X.T) #  numOfClass * numOfSample
    a_exp = np.exp(a)
```

```
    Score = np.zeros_like(a_exp)

    for j in range(Score.shape[1]):
        Score[:,j] = a_exp[:,j]/np.sum(a_exp[:,j])

    Score[y, range(Score.shape[1])] -=1
    dLda = Score
    grad = np.dot(dLda,X)

    grad /= y.shape[0]
    loss = self.loss(X, y)
    # ================================================================ #
    # 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
      print('numerical: %f analytic: %f, relative error: %e' % (grad_numerical, grad_an

  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.
    # ============================================================= #
    a = X.dot(self.W.T) #   numOfSample *numOfClass
    a = (a.T - np.amax(a,axis = 1)).T
    num_train = y.shape[0]
    a_exp = np.exp(a)
    Score = np.zeros_like(a_exp)
    Score = a_exp / np.sum(a_exp, axis = 1, keepdims = True)
    #loss += - np.log( np.exp(row[y[i]]) /sum(np.exp(row)) )
    loss = np.sum(  -np.log(a_exp[np.arange(a.shape[0]), y] / np.sum(a_exp, axis = 1))


    Score[range(num_train),y] -= 1
    dLda = Score
    grad = dLda.T.dot(X)
    grad /= num_train # NumOfClass * NumOfDimension

    loss = loss/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 number of c

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

```
    # 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.
      # ========================================================== #
      index = np.random.choice(np.arange(num_train), batch_size)
      X_batch = X[index]
      y_batch = y[index]
      # ========================================================== #
      # 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 = self.W - grad* learning_rate

          # ========================================================== #
      # 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_pred = np.argmax(X.dot(self.W.T),axis=1)
        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #

        return y_pred
```