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

# Load matplotlib images inline
%matplotlib inline

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

In [2]:
```
# Set the path to the CIFAR-10 data
cifar10_dir = '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', '
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) KNN's training function consists of just storing the entire training dataset.

(2) KNN's training step is very fast because no computation needs to be performed and can be updated very quickly because you can just add more data; however, it is also very memory-intensive because you need to store the entire training data set. Also, this training style can be thought of as making the prediction step slower, because all computation will have to occur in the prediction step.

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 c
#   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,
```

```
Time to run code: 57.7142288684845
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 a
# 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
```

```
Time to run code: 0.44336795806884766
Difference in L2 distances between your KNN implementations (should be
0): 1.4651847440245846e-10
```

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

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

y_pred = knn.predict_labels(dists_L2_vectorized, k=1)
error = np.count_nonzero(y_pred != y_test) / y_pred.shape[0]

# ================================================================ #
# 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

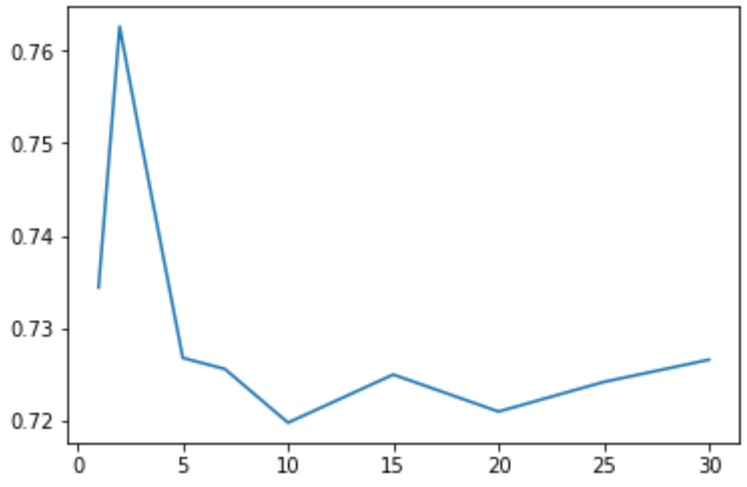
First, we will create the training and validation folds for use in k-fold cross validation.

In [10]:
```
# Create the dataset folds for cross-valdiation.
num_folds = 5

X_train_folds = []
y_train_folds =  []

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

X_train_folds = np.array_split(X_train, num_folds, axis=0)
y_train_folds = np.array_split(y_train, num_folds, axis=0)
# ================================================================ #
# 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.
# ================================================================ #

validation_error = []
for k in ks:
    error_folds_k = 0
    for i in range(0, num_folds):
        X_train_folds_k = X_train_folds[:]
        del X_train_folds_k[i]

        y_train_folds_k = y_train_folds[:]
        del y_train_folds_k[i]

        knn.train(X=np.vstack(X_train_folds_k), y=np.hstack(y_train_folds_k
        dists_L2_vectorized = knn.compute_L2_distances_vectorized(X=np.arra
        y_pred = knn.predict_labels(dists_L2_vectorized, k=k)
        error_folds_k += np.count_nonzero(y_pred != y_train_folds[i]) / y_p

    validation_error.append(error_folds_k / num_folds)


plt.plot(ks, validation_error)
plt.show()
print(validation_error)

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

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



```
[0.7344, 0.7626000000000002, 0.7504000000000001, 0.7267999999999999, 0.72
56, 0.7198, 0.725, 0.721, 0.7242, 0.7266]
Computation time: 47.41
```

Questions:

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

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

Answers:

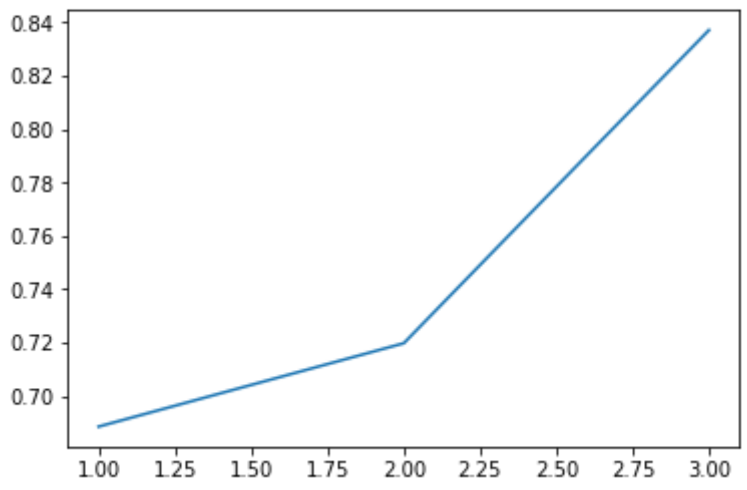
(1) k=10 is the best amongst the tested k's

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

Optimizing the norm

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

In [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 erroi
#   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.
# ================================================================ #

validation_error = []

for norm in norms:
    error_folds_norm = 0
    for i in range(0, num_folds):
        X_train_folds_k = X_train_folds[:]
        del X_train_folds_k[i]

        y_train_folds_k = y_train_folds[:]
        del y_train_folds_k[i]

        knn.train(X=np.vstack(X_train_folds_k), y=np.hstack(y_train_folds_k
        dists_L2 = knn.compute_distances(X=np.array(X_train_folds[i]), norm
        y_pred = knn.predict_labels(dists_L2, k=10)
        error_folds_norm += np.count_nonzero(y_pred != y_train_folds[i]) /

    validation_error.append(error_folds_norm / num_folds)


plt.plot([1, 2, 3], validation_error)
plt.show()
print(validation_error)

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



```
[0.6886000000000001, 0.7198, 0.8370000000000001]
Computation time: 1286.79
```

Questions:

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

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

Answers:

(1) L1 has best cross validation error

(2) For norm=L1 and K=10, we have a cross validation error of 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 [13]:
```
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.
# ================================================================ #

knn.train(X=X_train, y=y_train)
dists = knn.compute_distances(X_test, norm=L1_norm)
y_pred = knn.predict_labels(dists, k=10)
error = np.count_nonzero(y_pred != y_test) / y_pred.shape[0]

# ================================================================ #
# 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?

Answer:

Error from k=1 and L2-norm is 0.726, whereas error from k=10 and L1-norm is 0.722, this the error decreased by .004

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

    # Basically, we do (a_n^T*a_n - 2a_n^T*b_m + b_m^T*b_m)^0.5 for every n and m
    # this is equal to (a_n^2 - 2a_n^T*b_m + b_m^2)^0.5
    # which is equal to (A**2.shape(n,m) + B**2.shape(n,m) - 2 * A * B^T)^0.5
    # the shaping can be done using broadcasting
    dists = np.sqrt(np.sum(X**2, axis=1).reshape(X.shape[0], 1) + np.sum(self.X_t
rain**2, axis=1) - 2 * np.matmul(X, self.X_train.transpose()))

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

      top_k_closest = np.array([self.y_train[index] for index in np.argsort(dists
[i])[:k]]) # classes of top k closest
      labels, counts = np.unique(top_k_closest, return_counts=True)
      label_to_count = dict(zip(labels, counts))
      y_pred[i] = max(label_to_count.keys(), key=(lambda key: label_to_count[key
]))

      # ================================================================ #
      # 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 [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
import pdb

# Load matplotlib images inline
%matplotlib inline

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

In [2]:
```
# Set the path to the CIFAR-10 data
cifar10_dir = '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', '
        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]:

```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)
```

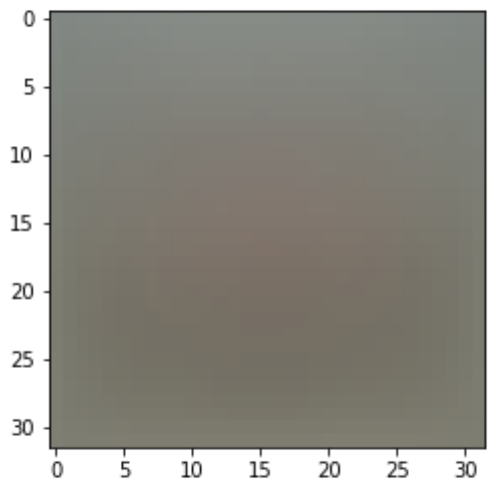
```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
Dev data shape:  (500, 32, 32, 3)
Dev labels shape:  (500,)
```

In [5]:
```
# 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 [6]:
```
# 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 m
plt.show()
```

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



In [7]:
```
# 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 [8]:
```
# third: append the bias dimension of ones (i.e. bias trick) so that our SV
# 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) In SVM, we do mean-subtraction to "center" the data, which helps because in our SVM, we learn a linear classifier, where the slope of the line is what classifies. When the data is centered, the slope has a greater range of possible useful values, whereas if the data is not centered, the very small changes in the slope would have a large impact on the accuracy. KNN doesn't require this because KNN predicts by comparing against all training examples and not by using a linear classifier.

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 [9]:
```
from nndl.svm import SVM
```

In [10]:
```
# 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

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 [11]:
```python
## 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 15569.977915410243.
```

SVM gradient

In [12]:
```python
## 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
svm.grad_check_sparse(X_dev, y_dev, grad)
```

```
numerical: -2.237781 analytic: -2.237782, relative error: 1.370280e-07
numerical: -6.215028 analytic: -6.215029, relative error: 2.057846e-08
numerical: 3.514615 analytic: 3.514616, relative error: 2.669430e-09
numerical: 5.457790 analytic: 5.457790, relative error: 2.598255e-08
numerical: 2.472639 analytic: 2.472639, relative error: 9.816170e-08
numerical: -4.412693 analytic: -4.412693, relative error: 1.745507e-08
numerical: 15.471785 analytic: 15.471785, relative error: 9.692493e-10
numerical: -12.574649 analytic: -12.574649, relative error: 1.399030e-08
numerical: -3.287478 analytic: -3.287478, relative error: 7.007659e-08
numerical: -11.708351 analytic: -11.708351, relative error: 7.247005e-09
```

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 [13]:
```python
import time
```

```
In [14]: ## 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.li

         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_vectori

         # The losses should match but your vectorized implementation should be much

         print('difference in loss / grad: {} / {}'.format(loss - loss_vectorized, n

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

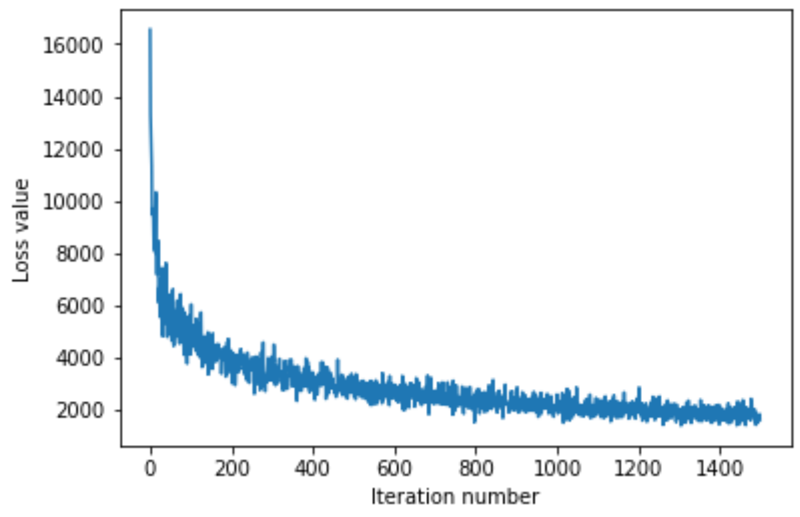
```
Normal loss / grad_norm: 15795.36706359361 / 2205.996997125613 computed i
n 0.13952994346618652s
Vectorized loss / grad: 15795.367063593592 / 2205.996997125613 computed i
n 0.04236173629760742s
difference in loss / grad: 1.8189894035458565e-11 / 7.647796263659292e-12
```

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 [15]: # 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 4017.333137942789
iteration 300 / 1500: loss 3681.9226471953616
iteration 400 / 1500: loss 2732.6164373988995
iteration 500 / 1500: loss 2786.637842464506
iteration 600 / 1500: loss 2837.0357842782664
iteration 700 / 1500: loss 2206.2348687399317
iteration 800 / 1500: loss 2269.0388241169803
iteration 900 / 1500: loss 2543.2378153859204
iteration 1000 / 1500: loss 2566.6921357268266
iteration 1100 / 1500: loss 2182.068905905164
iteration 1200 / 1500: loss 1861.1182244250451
iteration 1300 / 1500: loss 1982.9013858528256
iteration 1400 / 1500: loss 1927.5204158582108
That took 17.0645489692688s
```



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

In [16]:
```
## Implement svm.predict() and use it to compute the training and testing e

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: 0.28530612244897957
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 [17]: # ================================================================ #
         # 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 e
         #
         #   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
         # ================================================================ #
         for learning_rate in [1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1]:
             print('learning rate: {}'.format(learning_rate))
             loss_hist = svm.train(X_train, y_train, learning_rate=learning_rate,
                                   num_iters=1500, verbose=True)
             y_val_pred = svm.predict(X_val)
             print('validation accuracy: {}'.format(np.mean(np.equal(y_val, y_val_pr
             print()

         print('learning rate: {}'.format(0.01))
         loss_hist = svm.train(X_train, y_train, learning_rate=0.01,
                               num_iters=1500, verbose=True)
         y_test_pred = svm.predict(X_test)
         print('test accuracy: {}'.format(np.mean(np.equal(y_test, y_test_pred)), ))
         # ================================================================ #
         # END YOUR CODE HERE
         # ================================================================ #
```

```
learning rate: 1e-06
iteration 0 / 1500: loss 17772.834341836522
iteration 100 / 1500: loss 14780.267680784524
iteration 200 / 1500: loss 15075.635289979482
iteration 300 / 1500: loss 14839.594710902102
iteration 400 / 1500: loss 13792.279947184972
iteration 500 / 1500: loss 13657.987712622155
iteration 600 / 1500: loss 12401.422542976215
iteration 700 / 1500: loss 14551.880696294287
iteration 800 / 1500: loss 11866.563137872732
iteration 900 / 1500: loss 11604.224785862243
iteration 1000 / 1500: loss 12401.100658679432
iteration 1100 / 1500: loss 12527.734572806692
iteration 1200 / 1500: loss 10772.40405009249
iteration 1300 / 1500: loss 11598.917808630456
iteration 1400 / 1500: loss 10195.066636726382
validation accuracy: 0.158

learning rate: 1e-05
iteration 0 / 1500: loss 23298.531266498594
iteration 100 / 1500: loss 14358.246986907223
iteration 200 / 1500: loss 13318.214004290094
iteration 300 / 1500: loss 9511.451188326495
iteration 400 / 1500: loss 9656.339130875695
iteration 500 / 1500: loss 8570.866583531062
iteration 600 / 1500: loss 10609.262599603697
iteration 700 / 1500: loss 7810.51591088207
iteration 800 / 1500: loss 7110.797153744181
iteration 900 / 1500: loss 7246.924312358823
iteration 1000 / 1500: loss 7631.926511942877
iteration 1100 / 1500: loss 7250.779561846437
iteration 1200 / 1500: loss 7091.836948900854
iteration 1300 / 1500: loss 7079.380840941409
iteration 1400 / 1500: loss 6363.476743143491
validation accuracy: 0.172

learning rate: 0.0001
iteration 0 / 1500: loss 17434.679795253487
iteration 100 / 1500: loss 7182.478280261159
iteration 200 / 1500: loss 6649.587592162183
iteration 300 / 1500: loss 5125.807516631966
iteration 400 / 1500: loss 5443.689542470026
iteration 500 / 1500: loss 4134.158509323795
iteration 600 / 1500: loss 3609.0871693979925
iteration 700 / 1500: loss 4065.9117916523855
iteration 800 / 1500: loss 3823.1231163265884
iteration 900 / 1500: loss 4367.424151822208
iteration 1000 / 1500: loss 4123.190459235119
iteration 1100 / 1500: loss 3499.7795586868697
iteration 1200 / 1500: loss 3624.974854069729
iteration 1300 / 1500: loss 3880.960353934499
iteration 1400 / 1500: loss 3877.5708952772356
validation accuracy: 0.249

learning rate: 0.001
iteration 0 / 1500: loss 17726.97708162432
iteration 100 / 1500: loss 3993.5313843936437
iteration 200 / 1500: loss 3302.868798593412
iteration 300 / 1500: loss 2918.841268364429
iteration 400 / 1500: loss 2501.9415467794665
iteration 500 / 1500: loss 2544.3633522725886
iteration 600 / 1500: loss 3014.148281082504
iteration 700 / 1500: loss 2813.957096554755
iteration 800 / 1500: loss 2557.0873015992775
iteration 900 / 1500: loss 1989.3584996672423
iteration 1000 / 1500: loss 2260.6714692896467
iteration 1100 / 1500: loss 1632.215258818069
iteration 1200 / 1500: loss 1780.0244682220027
iteration 1300 / 1500: loss 1849.7555073251692
iteration 1400 / 1500: loss 2120.8059634599686
validation accuracy: 0.266

learning rate: 0.01
iteration 0 / 1500: loss 17439.114360917345
iteration 100 / 1500: loss 20026.28540088886
iteration 200 / 1500: loss 17084.722833943473
iteration 300 / 1500: loss 28856.128334494573
iteration 400 / 1500: loss 11897.13339210161
iteration 500 / 1500: loss 13998.344330680375
iteration 600 / 1500: loss 15577.41818175864
iteration 700 / 1500: loss 13406.674685569573
iteration 800 / 1500: loss 13353.254296805462
iteration 900 / 1500: loss 11540.603600221883
iteration 1000 / 1500: loss 20529.126891335487
iteration 1100 / 1500: loss 18963.55037857191
iteration 1200 / 1500: loss 25299.747843521014
iteration 1300 / 1500: loss 17509.667212414766
iteration 1400 / 1500: loss 14270.853309556525
validation accuracy: 0.306

learning rate: 0.1
iteration 0 / 1500: loss 15370.641063153469
iteration 100 / 1500: loss 150542.27179736446
iteration 200 / 1500: loss 274117.2538056461
iteration 300 / 1500: loss 123071.95138949061
iteration 400 / 1500: loss 155051.4597033335
iteration 500 / 1500: loss 134865.15341798545
iteration 600 / 1500: loss 120845.00936800771
iteration 700 / 1500: loss 168879.85332251273
iteration 800 / 1500: loss 196772.13993255346
iteration 900 / 1500: loss 129220.75171202901
iteration 1000 / 1500: loss 155807.71852033742
iteration 1100 / 1500: loss 131514.71749096332
iteration 1200 / 1500: loss 141571.6569927082
iteration 1300 / 1500: loss 161303.4897508109
iteration 1400 / 1500: loss 122190.6488423454
validation accuracy: 0.262

learning rate: 1
iteration 0 / 1500: loss 17834.89814950957
iteration 100 / 1500: loss 1158266.333167801
iteration 200 / 1500: loss 1786261.3171935224
iteration 300 / 1500: loss 1724946.6334517535
iteration 400 / 1500: loss 1140259.1475829678
iteration 500 / 1500: loss 1090646.4067051795
iteration 600 / 1500: loss 1843775.6248034923
iteration 700 / 1500: loss 1389840.3792271346
iteration 800 / 1500: loss 1684029.6508396333
iteration 900 / 1500: loss 2311462.6616987674
iteration 1000 / 1500: loss 1759591.1423630107
iteration 1100 / 1500: loss 936372.791510725
iteration 1200 / 1500: loss 1209672.535284187
iteration 1300 / 1500: loss 1598376.8037788253
iteration 1400 / 1500: loss 1501289.462830457
validation accuracy: 0.289

learning rate: 0.01
iteration 0 / 1500: loss 15754.992051210502
iteration 100 / 1500: loss 18381.617872154166
iteration 200 / 1500: loss 11726.910059578619
iteration 300 / 1500: loss 10283.978413134835
iteration 400 / 1500: loss 15012.203770867709
iteration 500 / 1500: loss 13323.311549204533
iteration 600 / 1500: loss 14413.097266453857
iteration 700 / 1500: loss 14467.422483423972
iteration 800 / 1500: loss 17488.75701978225
iteration 900 / 1500: loss 10194.2939965192
iteration 1000 / 1500: loss 9706.228906685465
iteration 1100 / 1500: loss 10518.140099153923
iteration 1200 / 1500: loss 23726.206699223956
iteration 1300 / 1500: loss 7353.385283170138
iteration 1400 / 1500: loss 15801.919263539872
test accuracy: 0.259
```

```
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

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

              loss += np.sum(np.maximum([0], 1 + np.matmul(self.W, X[i]) - np.matmul(self
        .W[y[i]], X[i]))) - 1
            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.
            # ================================================================ #
              loss += np.sum(np.maximum([0], 1 + np.matmul(self.W, X[i]) - np.matmul(self
        .W[y[i]], X[i]))) - 1

              indicators = np.sign(np.maximum([0], 1 + np.matmul(self.W, X[i]) - np.matmu
        l(self.W[y[i]], X[i])))
              indicators[y[i]] = 0
              grad += indicators.reshape(indicators.shape[0], 1) * X[i]
              grad[y[i]] += -np.sum(indicators.reshape(indicators.shape[0], 1) * X[i], ax
        is=0)

            # ================================================================ #
            # 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) + ab
        s(grad_analytic))
              print('numerical: %f analytic: %f, relative error: %e' % (grad_numerical, g
        rad_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 SVM loss WITHOUT any for loops.
    # ================================================================ #
    loss = (np.sum(np.maximum([0], 1 + np.matmul(X, self.W.transpose()) - np.sum(
np.multiply(self.W[y], X), axis=1).reshape(X.shape[0], 1))) - X.shape[0]) / X.sha
pe[0]
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #



    # ================================================================ #
    # YOUR CODE HERE:
    #   Calculate the SVM grad WITHOUT any for loops.
    # ================================================================ #
    indicators = np.sign(np.maximum([0], 1 + np.matmul(X, self.W.transpose()) - n
p.sum(np.multiply(self.W[y], X), axis=1).reshape(X.shape[0], 1)))
    indicators[range(X.shape[0]), y] = 0
    indicators[range(X.shape[0]), y] = -np.sum(indicators, axis=1)
    grad = np.matmul(indicators.transpose(), X)/X.shape[0]
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

    return loss, grad

  def train(self, X, y, learning_rate=1e-3, num_iters=100,
            batch_size=200, verbose=False):
    """
    Train this linear classifier using stochastic gradient descent.

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

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

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

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

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

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

    # ================================================================ #
    # YOUR CODE HERE:
    #   Predict the labels given the training data with the parameter self.W.
    # ================================================================ #
    y_pred = np.argsort(np.matmul(X, self.W.transpose()), axis=1)[:X.shape[0], se
lf.W.shape[0] - 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 [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
            """
            Load the CIFAR-10 dataset from disk and perform preprocessing to prepar
            it for the linear classifier. These are the same steps as we used for t
            SVM, but condensed to a single function.
            """
            # Load the raw CIFAR-10 data
            cifar10_dir = '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))
            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_
        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 [3]:
```python
from nndl import Softmax
```

In [4]:
```python
# 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

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 [5]:
```python
## Implement the loss function of the softmax using a for loop over
#  the number of examples

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

In [6]:
```python
print(loss)
```

```
2.3277607028048966
```

Question:

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

Answer:

Random chance is 0.1, and so we should expect the loss to be ln(10/1), which it is (ln(10) is approximately 2.3).

Softmax gradient

In [7]:
```
## 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
softmax.grad_check_sparse(X_dev, y_dev, grad)
```

```
numerical: 0.521283 analytic: 0.521283, relative error: 4.486898e-08
numerical: 1.487680 analytic: 1.487680, relative error: 1.262243e-08
numerical: -0.632911 analytic: -0.632911, relative error: 8.090488e-09
numerical: 0.643084 analytic: 0.643084, relative error: 1.800101e-08
numerical: 1.049247 analytic: 1.049247, relative error: 6.407224e-08
numerical: 1.680929 analytic: 1.680929, relative error: 2.309006e-08
numerical: -0.161634 analytic: -0.161634, relative error: 1.753171e-07
numerical: -2.457132 analytic: -2.457132, relative error: 4.540432e-09
numerical: 1.227282 analytic: 1.227282, relative error: 5.211303e-09
numerical: -2.886716 analytic: -2.886716, relative error: 9.843126e-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 [8]:
```
import time
```

In [9]:
```
## Implement softmax.fast_loss_and_grad which calculates the loss and gradi
#    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.li

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_vectori

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

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

```
Normal loss / grad_norm: 2.321805367940105 / 329.3714006915056 computed i
n 0.15343189239501953s
Vectorized loss / grad: 2.321805367940105 / 329.3714006915056 computed in
0.03251814842224121s
difference in loss / grad: 0.0 /2.1406619004282536e-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:

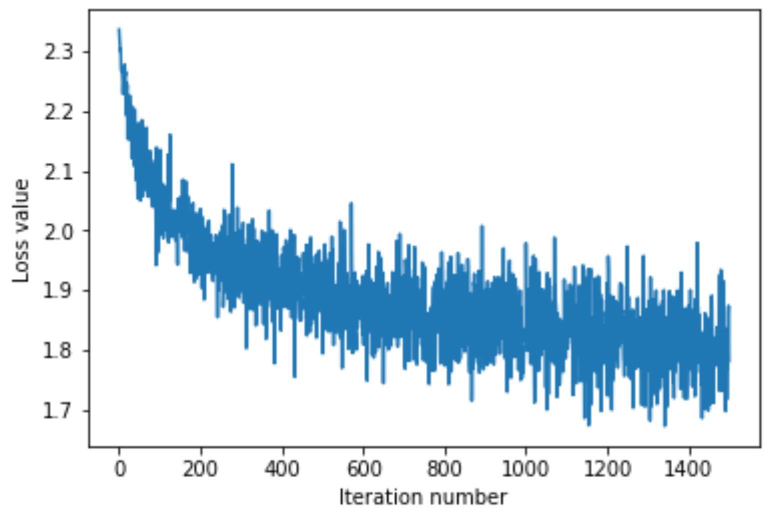
How should the softmax gradient descent training step differ from the svm training step, if at all?

Answer:

It is the same.

In [10]:
```
# Implement softmax.train() by filling in the code to extract a batch of da
# 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 1.8532611454359387
iteration 700 / 1500: loss 1.8353062223725827
iteration 800 / 1500: loss 1.829389246882764
iteration 900 / 1500: loss 1.899215853035748
iteration 1000 / 1500: loss 1.9783503540252299
iteration 1100 / 1500: loss 1.8470797913532633
iteration 1200 / 1500: loss 1.8411450268664082
iteration 1300 / 1500: loss 1.79104024957921
iteration 1400 / 1500: loss 1.8705803029382257
That took 10.551152229309082s
```



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

In [11]:
```
## Implement softmax.predict() and use it to compute the training and testi

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 [12]:
```
np.finfo(float).eps
```

Out[12]:
```
2.220446049250313e-16
```

In [13]:
```
# ================================================================ #
# 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 e
#
#   Select the SVM that achieved the best validation error and report
#     its error rate on the test set.
# ================================================================ #
for learning_rate in [1e-7, 1e-6, 1e-5, 1e-4]:
    print('learning rate: {}'.format(learning_rate))
    loss_hist = softmax.train(X_train, y_train, learning_rate=learning_rate
                      num_iters=1500, verbose=True)
    y_val_pred = softmax.predict(X_val)
    print('validation accuracy: {}'.format(np.mean(np.equal(y_val, y_val_pr
    print()

print('learning rate: {}'.format(1e-6))
loss_hist = softmax.train(X_train, y_train, learning_rate=1e-6,
                      num_iters=1500, verbose=True)
y_test_pred = softmax.predict(X_test)
print('test accuracy: {}'.format(np.mean(np.equal(y_test, y_test_pred))))
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

```
learning rate: 1e-07
iteration 0 / 1500: loss 2.3353835450891545
iteration 100 / 1500: loss 2.022509394631719
iteration 200 / 1500: loss 1.982172871654982
iteration 300 / 1500: loss 1.9356442081331486
iteration 400 / 1500: loss 1.882893396815689
iteration 500 / 1500: loss 1.8181869697394497
iteration 600 / 1500: loss 1.874513153185746
iteration 700 / 1500: loss 1.8361832500173585
iteration 800 / 1500: loss 1.8584086819212182
iteration 900 / 1500: loss 1.9275087067564147
iteration 1000 / 1500: loss 1.824667969507725
iteration 1100 / 1500: loss 1.7731817984393607
iteration 1200 / 1500: loss 1.8636308568113116
iteration 1300 / 1500: loss 1.924074621260815
iteration 1400 / 1500: loss 1.7846918635831293
validation accuracy: 0.39

learning rate: 1e-06
iteration 0 / 1500: loss 2.4615346985497166
iteration 100 / 1500: loss 1.7515308294429426
iteration 200 / 1500: loss 1.8653151657870888
iteration 300 / 1500: loss 1.7068279724663447
iteration 400 / 1500: loss 1.6919412980959523
iteration 500 / 1500: loss 1.7445602534086055
iteration 600 / 1500: loss 1.9070927441191992
iteration 700 / 1500: loss 1.6266282009657491
iteration 800 / 1500: loss 1.7137070023201977
iteration 900 / 1500: loss 1.6755856266246776
iteration 1000 / 1500: loss 1.8076485575084924
iteration 1100 / 1500: loss 1.7256661402410827
iteration 1200 / 1500: loss 1.698554182531272
iteration 1300 / 1500: loss 1.792082948906467
iteration 1400 / 1500: loss 1.6508555418428204
validation accuracy: 0.415

learning rate: 1e-05
iteration 0 / 1500: loss 2.3790388831757823
iteration 100 / 1500: loss 2.3921785131993816
iteration 200 / 1500: loss 3.27163372235802
iteration 300 / 1500: loss 2.5266749616255084
iteration 400 / 1500: loss 2.5171351792770027
iteration 500 / 1500: loss 2.9346880547320247
iteration 600 / 1500: loss 1.9232171018336048
iteration 700 / 1500: loss 3.44210890971557
iteration 800 / 1500: loss 2.27376749441641
iteration 900 / 1500: loss 2.745258811167521
iteration 1000 / 1500: loss 2.589548028949166
iteration 1100 / 1500: loss 2.737905712614821
iteration 1200 / 1500: loss 2.747594734219702
iteration 1300 / 1500: loss 2.5731286656092824
iteration 1400 / 1500: loss 3.1635404402767886
validation accuracy: 0.27

learning rate: 0.0001
iteration 0 / 1500: loss 2.338799247622096
iteration 100 / 1500: loss 34.98601680024577
iteration 200 / 1500: loss 56.97405897104403
iteration 300 / 1500: loss 14.824554602730215
iteration 400 / 1500: loss 25.6996155749384
iteration 500 / 1500: loss 33.93294068543631
iteration 600 / 1500: loss 24.208229112153166
iteration 700 / 1500: loss 20.911355216047518
iteration 800 / 1500: loss 39.34415640628004
iteration 900 / 1500: loss 16.4359260239477
iteration 1000 / 1500: loss 24.71359284063535
iteration 1100 / 1500: loss 18.564259007064685
iteration 1200 / 1500: loss 28.5426977737122
iteration 1300 / 1500: loss 36.66212965037604
iteration 1400 / 1500: loss 36.16062817387996
validation accuracy: 0.271

learning rate: 1e-06
iteration 0 / 1500: loss 2.372474398135772
iteration 100 / 1500: loss 1.892919730065111
iteration 200 / 1500: loss 1.790029417712668
iteration 300 / 1500: loss 1.8396019743788241
iteration 400 / 1500: loss 1.6965287331558643
iteration 500 / 1500: loss 1.8498152607388902
iteration 600 / 1500: loss 1.608555393293517
iteration 700 / 1500: loss 1.7768936439173046
iteration 800 / 1500: loss 1.7288467550175795
iteration 900 / 1500: loss 1.7191678304814781
iteration 1000 / 1500: loss 1.8442746025120789
iteration 1100 / 1500: loss 1.7596152530507356
iteration 1200 / 1500: loss 1.7196895935660637
iteration 1300 / 1500: loss 1.720036481810709
iteration 1400 / 1500: loss 1.7480014172362022
test accuracy: 0.392
```

```
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.)
            # ================================================================ #
            for i in np.arange(X.shape[0]):
              loss += np.log(np.sum(np.exp(np.matmul(self.W, X[i])))) - np.matmul(self.W[
        y[i]].transpose(), X[i])

            loss /= X.shape[0]
            # ================================================================ #
            # END YOUR CODE HERE
            # ================================================================ #

            return loss

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

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

            # Initialize the loss and gradient to zero.
```

```
    loss = 0.0
    grad = np.zeros_like(self.W)

    # ================================================================ #
    # YOUR CODE HERE:
    #   Calculate the softmax loss and the gradient. Store the gradient
    #   as the variable grad.
    # ================================================================ #
    for i in np.arange(X.shape[0]):
      loss += np.log(np.sum(np.exp(np.matmul(self.W, X[i])))) - np.matmul(self.W[
y[i]].transpose(), X[i])
    loss /= X.shape[0]

    for i in range(X.shape[0]):
      denominator = np.sum(np.exp(np.matmul(self.W, X[i])))
      for j in range(self.W.shape[0]):
        if j == y[i]:
          grad[j] += -X[i] + np.exp(np.dot(self.W[j], X[i])) * X[i] / denominator
        else:
          grad[j] += np.exp(np.dot(self.W[j], X[i])) * X[i] / denominator
    grad /= X.shape[0]
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

    return loss, grad

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

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

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

      grad_numerical = (fxph - fxmh) / (2 * h)
      grad_analytic = your_grad[ix]
      rel_error = abs(grad_numerical - grad_analytic) / (abs(grad_numerical) + ab
s(grad_analytic))
      print('numerical: %f analytic: %f, relative error: %e' % (grad_numerical, g
rad_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.
    # ================================================================ #
    loss = np.sum(np.log(np.sum(np.exp(np.matmul(self.W, X.T)), axis=0)) - np.sum
(self.W[y] * X, axis=1))
    loss /= X.shape[0]

    base = np.exp(np.matmul(self.W, X.T)) / np.sum(np.exp(np.matmul(X, self.W.T
)), axis=1)
    print(base.shape)
    y[0] = 12
    base[y,np.arange(X.shape[0])] -= 1
    print(base[y])
    grad = np.matmul(base, X)
    grad /= X.shape[0]
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

    return loss, grad

  def train(self, X, y, learning_rate=1e-3, num_iters=100,
            batch_size=200, verbose=False):
    """
    Train this linear classifier using stochastic gradient descent.

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

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

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

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

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

      # ================================================================ #
      # YOUR CODE HERE:
      #   Sample batch_size elements from the training data for use in
      #      gradient descent.  After sampling,
      #     - X_batch should have shape: (dim, batch_size)
      #     - y_batch should have shape: (batch_size,)
      #   The indices should be randomly generated to reduce correlations
      #   in the dataset.  Use np.random.choice.  It's okay to sample with
      #   replacement.
      # ================================================================ #
      indices = np.random.choice(X.shape[0], batch_size)
      X_batch = X[indices,:]
      y_batch = y[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[0])
    # ================================================================ #
    # YOUR CODE HERE:
    #   Predict the labels given the training data.
    # ================================================================ #
    y_pred = np.argsort(np.matmul(X, self.W.transpose()), axis=1)[:X.shape[0], se
lf.W.shape[0] - 1]
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

    return y_pred
```