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.

Import the appropriate libraries

In [91]:
```
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 da
from collections import Counter
import operator

# 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-ipy
%load_ext autoreload
%autoreload 2
```

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

In [80]:
```
# Set the path to the CIFAR-10 data
cifar10_dir = '/home/alon/school/c247a/datasets/cifar-10-batches-py' # You ne
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 [81]:

```
# 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', 'sh
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 [82]:

```
# 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 [8]:
```
# Import the KNN class

from nndl import KNN
```

In [9]:
```
# Declare an instance of the knn class.
knn = KNN()

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

Questions

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

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

Answers

(1) The program is caching the entire training dataset, in order to be able to compute distances using the chosen metric for each test example.

(2) Pros: this step is very fast and simple, O(1) training complexity. Cons: if the dataset is large, this takes a lot of memory. Further, the amount of memory needed scales with the number of training examples and the input vector size.

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 [36]:
```
# Implement the function compute_distances() in the KNN class.
# Do not worry about the input 'norm' for now; use the default definition of
#   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, 'f
print('Shape of dists_L2: {}'.format(dists_L2.shape))
```

```
Time to run code: 35.81522274017334
Frobenius norm of L2 distances: 7906696.077040902
Shape of dists_L2: (500, 5000)
```

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 [153…

```
# 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
# 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
```

```
Time to run code: 0.23537254333496094
Difference in L2 distances between your KNN implementations (should be 0): 0.
0
```

Speedup

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

Implementing the prediction

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

In [185…

```
k = 1

correct = 0

num_test = dists_L2_vectorized.shape[0]
y_pred = np.zeros(num_test)

for i in np.arange(num_test):
    sorted_row_k_ixs = np.argsort(dists_L2_vectorized[i])[:k]
    #sorted_row_k = np.sort(dists_L2_vectorized[i])[:k]
    #print(sorted_row_k_ixs)
    nn_class_ixs = [y_train[ix] for ix in sorted_row_k_ixs]
    #print(nn_class_ixs)
    nn_class_top = max(set(nn_class_ixs), key=nn_class_ixs.count)
    #print(nn_class_top)
    #print("predicted class: {}".format(classes[nn_class_top]))
    #print("correct class: {}".format(classes[y_test[i]]))

    y_pred[i] = nn_class_top

    if (nn_class_top == y_test[i]):
        correct += 1
    #print('\n')

print("Total correct: {}".format(correct))
print("Percent correct: {}".format(correct/num_test))
```

```
Total correct: 137
Percent correct: 0.274
```

In [131…

```
# 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
num_errors = 0

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

predicted_labels = knn.predict_labels(dists_L2_vectorized, k=1)
#print(predicted_labels)
for i, lab in enumerate(predicted_labels):
    if (lab != y_test[i]):
        num_errors += 1

error = num_errors/y_test.shape[0]

#pass
# ================================================================ #
# 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 [136…

```
# 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]
# ================================================================ #


fold_size = int(X_train.shape[0]/num_folds)
#print(fold_size)

for i in range(num_folds):
    X_fold = X_train[fold_size*i:fold_size*(i+1)]
    X_train_folds.append(X_fold)
    y_fold = y_train[fold_size*i:fold_size*(i+1)]
    y_train_folds.append(y_fold)


# ================================================================ #
# 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 [150…

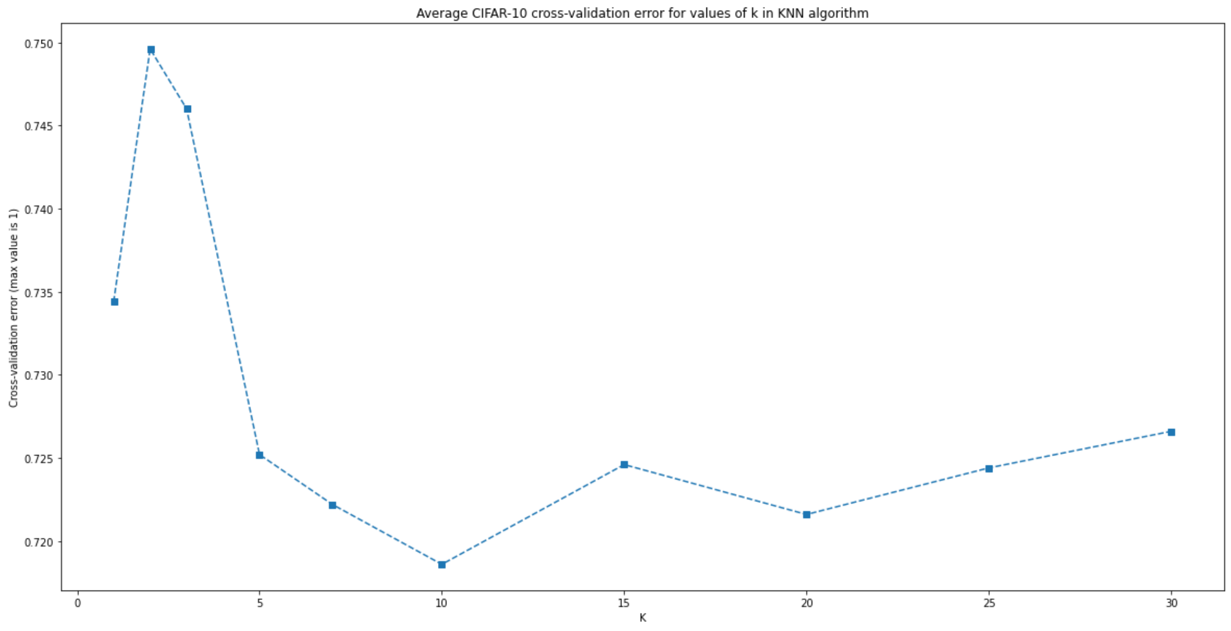
```
#sad = [int(item) for item in np.arange(len(X_train_folds))!=2]
sad = [x for i,x in enumerate(X_train_folds) if i!=3]
bad = np.concatenate(sad,axis=0)
print(bad.shape)
```

```
(4000, 3072)
```

In [176…

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

errs = []

#loop through the ks:
for k in ks:

    errs_k = []

    #loop through the folds:
    for i in range(0, num_folds):
        #obtain validation fold
        X_val_fold = X_train_folds[i]
        y_val_fold = y_train_folds[i]

        #obtain training fold set
        X_train_set = np.concatenate([x for j,x in enumerate(X_train_folds) i
        y_train_set = np.concatenate([x for j,x in enumerate(y_train_folds) i

        # Declare an instance of the knn class.
        knn_inst = 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_inst.train(X=X_train_set, y=y_train_set)

        dists = knn_inst.compute_L2_distances_vectorized(X=X_val_fold)
        predicted_labels = knn_inst.predict_labels(dists, k=k)

        num_errors = 0

        for index, label in enumerate(predicted_labels):
            if label != y_val_fold[index]:
                num_errors += 1

        #print("Number of errors for this step: {}".format(num_errors))
        error = num_errors/y_val_fold.shape[0]

        #print("Cross-validation error for fold {} and k={}: {}".format(i, k,
        errs_k.append(error)

    errs.append(errs_k)

#average errors for each k to plot
average_errs_k = [sum(lst)/len(lst) for lst in errs]
plt.figure(figsize=(20,10))
```

```
Computation time: 35.57
```



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 has the best performance.

(2) 0.7186 or 71.86%

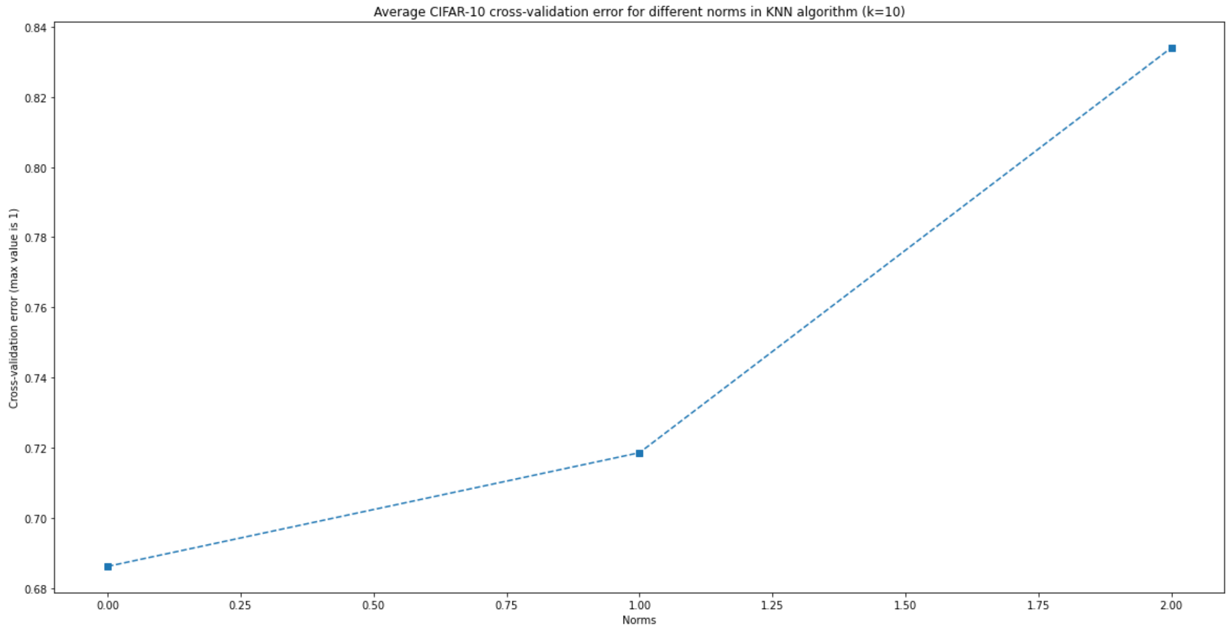
Optimizing the norm

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 [181…

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

errs = []

for norm in norms:
    #print(norm())

    errs_norm = []

    #loop through the folds:
    for i in range(0, num_folds):
        #obtain validation fold
        X_val_fold = X_train_folds[i]
        y_val_fold = y_train_folds[i]

        #obtain training fold set
        X_train_set = np.concatenate([x for j,x in enumerate(X_train_folds) i
        y_train_set = np.concatenate([x for j,x in enumerate(y_train_folds) i

        # Declare an instance of the knn class.
        knn_inst = 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_inst.train(X=X_train_set, y=y_train_set)

        #dists = knn_inst.compute_L2_distances_vectorized(X=X_val_fold)
        dists = knn_inst.compute_distances(X=X_val_fold, norm=norm)
        #print(dists.shape)
        predicted_labels = knn_inst.predict_labels(dists, k=10)

        num_errors = 0

        for index, label in enumerate(predicted_labels):
            if label != y_val_fold[index]:
                num_errors += 1

        #print("Number of errors for this step: {}".format(num_errors))
        error = num_errors/y_val_fold.shape[0]
```

```
Computation time: 738.26
```



Questions:

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

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

In [182…

```
print(average_errs_norm)
```

```
[0.6862000000000001, 0.7186, 0.834]
```

Answers:

(1) The L1 norm.

(2) 0.6862 or 68.62%

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 [184...

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

#using k = 10 and L1 norm:

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

#train
knn_inst.train(X=X_train, y=y_train)

#compute distances and predict labels
dists = knn_inst.compute_distances(X=X_test, norm=L1_norm)
predicted_labels = knn_inst.predict_labels(dists, k=10)

num_errors = 0

for index, label in enumerate(predicted_labels):
    if label != y_test[index]:
        num_errors += 1

#print("Number of errors for this step: {}".format(num_errors))
error = num_errors/y_test.shape[0]

print("Error rate for k=10 and L1 Norm KNN algorithm: {}".format(error))
print("Improvement with cross validation approach: {}".format(0.726 - error))

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

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

```
Error rate for k=10 and L1 Norm KNN algorithm: 0.714
Improvement with cross validation approach: 0.01200000000000001
Error rate achieved: 0.714
```

Question:

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

Answer:

I got an improvement of 0.012 or 1.2%.

In [ ]:
```
import numpy as np
import pdb
from collections import Counter
import operator


"""
This code was based off of code from cs231n at Stanford University, and modif
"""

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 traini
      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 j
        #   training point using norm(), and store the result in dists[i, j].
        # ================================================================== #

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

        #pass
```

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 [2]:
```
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 [3]:
```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000,
    """
    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 = '/home/alon/school/c247a/datasets/cifar-10-batches-py' # Yo
    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_da
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 [4]:
```
from nndl import Softmax
```

In [5]:
```
# 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 ra

np.random.seed(1)

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

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

In [6]:
```
print(softmax.W.shape)
print(X_train.shape)

print(X_train.dot(softmax.W.T).shape) #49k x 3073 dot 3073 x 10
print(softmax.W.dot(X_train.T).shape) #10 x 3073 dot 3073 x 49k

print(np.max(softmax.W)/.0001)
```

```
(10, 3073)
(49000, 3073)
(49000, 10)
(10, 49000)
4.168117677955094
```

In [7]:
```
# Initialize the loss to zero.
loss = 0.0

#a(x) will be 49k x 10
a = X_train.dot(softmax.W.T)
#print(a.shape)

#placeholder for 49k loss summation terms
row_losses = []

for ix, row in enumerate(a):
    #print(row.shape)
    #y_train[ix] is the correct class index
    #we know that a[y_train[ix]] is then the score of the correct class index
    #and row itself is the 10 scores of the 10 classes in order
    row_loss = -1*(np.log(np.exp(row[y_train[ix]])/np.sum(np.exp(row))))


# #we will be summing these m terms later
# for ix, row, in enumerate(a):
#     row -= np.max(row) #implement a~ to prevent overflow
#     #each row loss is just log(e^(WiT*X)/sum(1,#classes)(WjT*X))
#     row_loss = -1*(np.log(np.exp(row[y[ix]])/np.sum(np.exp(row))))
#     row_losses.append(row_loss)

# #normalize by the number of samples
# loss = np.sum(row_losses)/X.shape[0]
```

Softmax loss

In [86]:
```
## Implement the loss function of the softmax using a for loop over
#  the number of examples

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

In [87]:
```
print(loss)
```

```
2.3277607028048757
```

Question:

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

Answer:

What we are doing in this function is: for each example, we record the ratio of the score for the correct class to the score for all 10 classes summed up. We expect that, on average, this ratio should be 1/10, because our weights are random and should activate evenly over a large number of examples. We are actually taking the log of this ratio, which works out to about -2.3 (but we consider the losses to be positive, so the sign is flipped).

Softmax gradient

In [106…

```
a = softmax.W.dot(X_dev.T) #  numOfClass * numOfSample
a2 = X_dev.dot(softmax.W.T)
a_exp = np.exp(a)
score = np.zeros_like(a_exp)

dLda = score
grad = np.dot(dLda,X_dev)
print(grad.shape)

print(a.shape)
print(a2.shape)
```

```
(10, 3073)
(10, 500)
(500, 10)
```

In [124…

```
## 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)

#print(grad.shape)

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

```
numerical: -1.663829 analytic: -1.663829, relative error: 4.381255e-09
numerical: -0.187085 analytic: -0.187085, relative error: 1.800690e-08
numerical: -0.703666 analytic: -0.703666, relative error: 1.105930e-08
numerical: -2.316506 analytic: -2.316506, relative error: 1.738799e-08
numerical: 0.625989 analytic: 0.625989, relative error: 2.020045e-08
numerical: -1.731080 analytic: -1.731080, relative error: 2.940948e-08
numerical: -0.791582 analytic: -0.791582, relative error: 3.056683e-08
numerical: -4.783476 analytic: -4.783476, relative error: 6.223860e-09
numerical: 1.128093 analytic: 1.128093, relative error: 1.424320e-08
numerical: 3.374623 analytic: 3.374622, relative error: 2.213435e-08
```

A vectorized version of Softmax

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

In [125…

```
import time
```

In [136…

```
## Implement softmax.fast_loss_and_grad which calculates the loss and gradien
#    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.lina

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_vectorize

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

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

```
Normal loss / grad_norm: 2.3393592316203606 / 355.62450083961147 computed in
0.05254864692687988s
Vectorized loss / grad: 2.3393592316203606 / 355.62450083961147 computed in
0.0053462982177734375s
difference in loss / grad: 0.0 /5.276854027086815e-14
```

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?

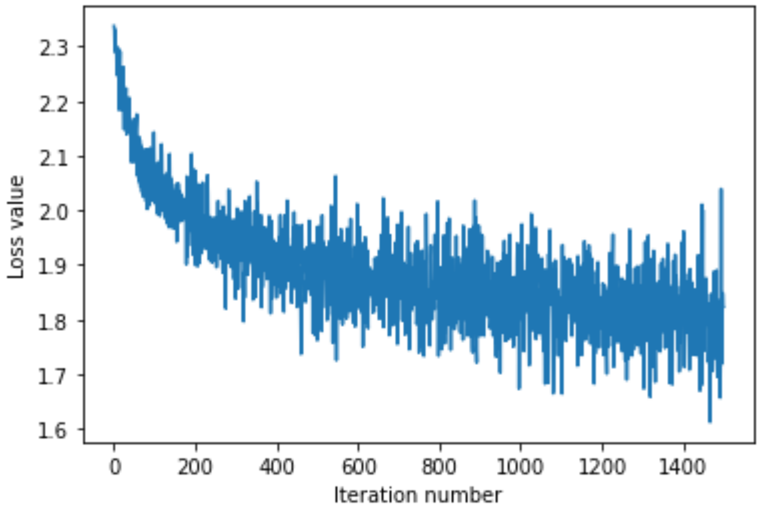
Answer:

They use different loss functions, but otherwise they won't differ.

In [138…

```python
# 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 2.3375919298138235
iteration 100 / 1500: loss 2.0725459949069256
iteration 200 / 1500: loss 2.07223444240482
iteration 300 / 1500: loss 1.9486195973654452
iteration 400 / 1500: loss 1.8399004881304943
iteration 500 / 1500: loss 1.761731290369851
iteration 600 / 1500: loss 1.8206970692535032
iteration 700 / 1500: loss 1.9649850923872147
iteration 800 / 1500: loss 1.8524078293221562
iteration 900 / 1500: loss 1.7920322268634294
iteration 1000 / 1500: loss 1.8339621390492526
iteration 1100 / 1500: loss 1.7752342933754048
iteration 1200 / 1500: loss 1.8010313946050127
iteration 1300 / 1500: loss 1.9009308402355303
iteration 1400 / 1500: loss 1.8833302305951873
That took 8.184735774993896s
```



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

In [142…

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

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)),
```

```
(49000, 10)
training accuracy: 0.3826122448979592
(1000, 10)
validation accuracy: 0.403
```

Optimize the softmax classifier

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

In [143…

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

Out[143…

```
2.220446049250313e-16
```

In [148…

```
# ================================================================ #
# 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 err
#
#   Select the SVM that achieved the best validation error and report
#     its error rate on the test set.
# ================================================================ #

#try learning rates between 10^-6 and 10^-3 in increments of 2.5e-6
learning_rates = np.linspace(1e-8, 1e-5, 100)

best_learning_rate = 0.0
best_val_accuracy = 0.0

for learning_rate in learning_rates:
    softmax.train(X_train, y_train, learning_rate=learning_rate,
                  num_iters=1500, verbose=False)
    y_pred_validation = softmax.predict(X_val)
    val_accuracy = np.mean(np.equal(y_val,y_pred_validation))
    print("Validation accuracy for {}: {}".format(learning_rate, val_accuracy

    if (val_accuracy > best_val_accuracy):
        #update
        best_val_accuracy = val_accuracy
        best_learning_rate = learning_rate

print('\n')

print("best learning rate is: {}".format(best_learning_rate))

#retrain and test on best learning rate
softmax.train(X_train, y_train, learning_rate=best_learning_rate, num_iters=1
y_test_pred = softmax.predict(X_test)
test_accuracy = np.mean(np.equal(y_test,y_test_pred))

print("Test accuracy at best rate is: {}".format(test_accuracy))
print("Test error at best rate is: {}".format(1-test_accuracy))



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

```
Validation accuracy for 1e-08: 0.316
Validation accuracy for 1.1090909090909092e-07: 0.391
Validation accuracy for 2.1181818181818186e-07: 0.409
Validation accuracy for 3.127272727272728e-07: 0.406
Validation accuracy for 4.136363636363637e-07: 0.399
Validation accuracy for 5.145454545454546e-07: 0.397
Validation accuracy for 6.154545454545456e-07: 0.41
Validation accuracy for 7.163636363636365e-07: 0.41
Validation accuracy for 8.172727272727274e-07: 0.416
Validation accuracy for 9.181818181818183e-07: 0.409
Validation accuracy for 1.0190909090909092e-06: 0.407
Validation accuracy for 1.12e-06: 0.415
Validation accuracy for 1.220909090909091e-06: 0.398
Validation accuracy for 1.321818181818182e-06: 0.407
Validation accuracy for 1.4227272727272729e-06: 0.409
Validation accuracy for 1.5236363636363638e-06: 0.41
Validation accuracy for 1.6245454545454547e-06: 0.414
Validation accuracy for 1.7254545454545456e-06: 0.408
Validation accuracy for 1.8263636363636365e-06: 0.417
Validation accuracy for 1.9272727272727277e-06: 0.422
Validation accuracy for 2.0281818181818184e-06: 0.401
Validation accuracy for 2.129090909090909e-06: 0.401
Validation accuracy for 2.2300000000000002e-06: 0.39
Validation accuracy for 2.3309090909090914e-06: 0.399
Validation accuracy for 2.431818181818182e-06: 0.407
Validation accuracy for 2.532727272727273e-06: 0.394
Validation accuracy for 2.633636363636364e-06: 0.389
Validation accuracy for 2.734545454545455e-06: 0.387
Validation accuracy for 2.835454545454546e-06: 0.393
Validation accuracy for 2.9363636363636365e-06: 0.408
Validation accuracy for 3.0372727272727276e-06: 0.396
Validation accuracy for 3.1381818181818188e-06: 0.398
Validation accuracy for 3.2390909090909095e-06: 0.39
Validation accuracy for 3.34e-06: 0.385
Validation accuracy for 3.4409090909090913e-06: 0.38
Validation accuracy for 3.5418181818181825e-06: 0.384
Validation accuracy for 3.642727272727273e-06: 0.384
Validation accuracy for 3.743636363636364e-06: 0.39
Validation accuracy for 3.8445454545454555e-06: 0.377
Validation accuracy for 3.945454545454546e-06: 0.354
Validation accuracy for 4.046363636363637e-06: 0.396
Validation accuracy for 4.147272727272728e-06: 0.382
Validation accuracy for 4.248181818181818e-06: 0.393
Validation accuracy for 4.34909090909091e-06: 0.386
Validation accuracy for 4.450000000000001e-06: 0.402
Validation accuracy for 4.550909090909091e-06: 0.358
Validation accuracy for 4.651818181818183e-06: 0.398
Validation accuracy for 4.752727272727274e-06: 0.372
Validation accuracy for 4.853636363636364e-06: 0.386
Validation accuracy for 4.954545454545455e-06: 0.363
Validation accuracy for 5.055454545454546e-06: 0.365
Validation accuracy for 5.156363636363637e-06: 0.411
Validation accuracy for 5.257272727272728e-06: 0.358
Validation accuracy for 5.358181818181819e-06: 0.365
Validation accuracy for 5.45909090909091e-06: 0.399
Validation accuracy for 5.560000000000001e-06: 0.369
Validation accuracy for 5.660909090909092e-06: 0.388
Validation accuracy for 5.761818181818182e-06: 0.353
Validation accuracy for 5.862727272727273e-06: 0.376
Validation accuracy for 5.963636363636365e-06: 0.368
Validation accuracy for 6.064545454545455e-06: 0.357
Validation accuracy for 6.165454545454546e-06: 0.324
Validation accuracy for 6.266363636363638e-06: 0.384
Validation accuracy for 6.367272727272728e-06: 0.347
Validation accuracy for 6.468181818181819e-06: 0.365
Validation accuracy for 6.56909090909091e-06: 0.353
Validation accuracy for 6.6700000000000005e-06: 0.373
Validation accuracy for 6.770909090909092e-06: 0.329
Validation accuracy for 6.871818181818183e-06: 0.332
Validation accuracy for 6.9727272727272735e-06: 0.361
Validation accuracy for 7.073636363636365e-06: 0.329
Validation accuracy for 7.174545454545456e-06: 0.368
Validation accuracy for 7.2754545454545465e-06: 0.344
Validation accuracy for 7.376363636363637e-06: 0.342
Validation accuracy for 7.477272727272728e-06: 0.363
Validation accuracy for 7.5781818181818195e-06: 0.335
Validation accuracy for 7.679090909090911e-06: 0.319
Validation accuracy for 7.780000000000002e-06: 0.346
Validation accuracy for 7.880909090909092e-06: 0.385
Validation accuracy for 7.981818181818183e-06: 0.326
Validation accuracy for 8.082727272727274e-06: 0.34
Validation accuracy for 8.183636363636365e-06: 0.349
Validation accuracy for 8.284545454545455e-06: 0.334
Validation accuracy for 8.385454545454546e-06: 0.33
Validation accuracy for 8.486363636363637e-06: 0.283
Validation accuracy for 8.587272727272729e-06: 0.325
Validation accuracy for 8.68818181818182e-06: 0.271
Validation accuracy for 8.78909090909091e-06: 0.311
Validation accuracy for 8.890000000000001e-06: 0.309
Validation accuracy for 8.990909090909092e-06: 0.331
Validation accuracy for 9.091818181818183e-06: 0.296
Validation accuracy for 9.192727272727273e-06: 0.349
Validation accuracy for 9.293636363636366e-06: 0.306
Validation accuracy for 9.394545454545457e-06: 0.335
Validation accuracy for 9.495454545454547e-06: 0.313
Validation accuracy for 9.596363636363638e-06: 0.387
Validation accuracy for 9.697272727272729e-06: 0.281
Validation accuracy for 9.79818181818182e-06: 0.317
Validation accuracy for 9.89909090909091e-06: 0.315
Validation accuracy for 1e-05: 0.318


best learning rate is: 1.9272727272727277e-06
iteration 0 / 1500: loss 2.315555637393837
iteration 100 / 1500: loss 1.7712962614279315
iteration 200 / 1500: loss 1.7543143009221092
iteration 300 / 1500: loss 1.7031198154929883
iteration 400 / 1500: loss 1.7412277519253723
iteration 500 / 1500: loss 1.7432401553830528
iteration 600 / 1500: loss 1.7801006474444518
iteration 700 / 1500: loss 1.7449522103257158
iteration 800 / 1500: loss 1.6594433616120778
iteration 900 / 1500: loss 1.7277990566303787
iteration 1000 / 1500: loss 1.8170872351623095
iteration 1100 / 1500: loss 1.6747728730555336
iteration 1200 / 1500: loss 1.7716615871190304
iteration 1300 / 1500: loss 1.6645849787228046
iteration 1400 / 1500: loss 1.6354491488436766
Test accuracy at best rate is: 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 minibatch
    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 mea
      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
    #   (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)

    #placeholder for m loss terms
    row_losses = []

    #we will be summing these m terms later
    for ix, row, in enumerate(a):
        row -= np.max(row) #implement a~ to prevent overflow
        #each row loss is just log(e^(WiT*X)/sum(1,#classes)(WjT*X))
        #row_loss = -1*(np.log(np.exp(row[y[ix]])/np.sum([np.exp(item) for it
        row_loss = -1*(np.log(np.exp(row[y[ix]])/np.sum(np.exp(row))))
        row_losses.append(row_loss)

    #normalize by the number of samples
    loss = np.sum(row_losses)/X.shape[0]
```

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.

Importing libraries and data setup

In [2]:
```
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 da
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-ipy
%load_ext autoreload
%autoreload 2
```

In [3]:
```
# Set the path to the CIFAR-10 data
cifar10_dir = '/home/alon/school/c247a/datasets/cifar-10-batches-py' # You ne
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 [4]:
```
# 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', 'sh
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 [5]:
```
# 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)
```

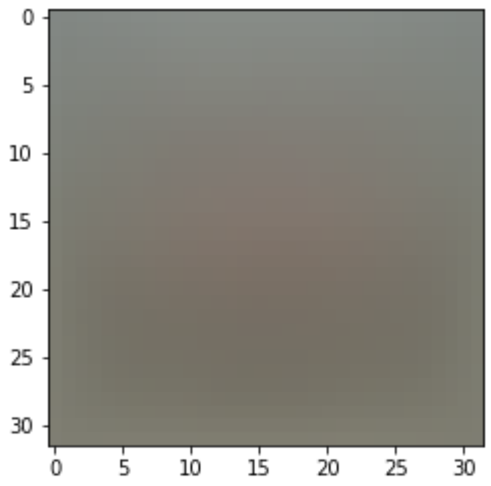
```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
Dev data shape:  (500, 32, 32, 3)
Dev labels shape:  (500,)
```

In [6]:
```
# 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 [7]:
```
# 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 mea
plt.show()
```

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



In [8]:
```
# 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 [9]:
```
# third: append the bias dimension of ones (i.e. bias trick) so that our SVM
# only has to worry about optimizing a single weight matrix W.
X_train = np.hstack([X_train, np.ones((X_train.shape[0], 1))])
X_val = np.hstack([X_val, np.ones((X_val.shape[0], 1))])
X_test = np.hstack([X_test, np.ones((X_test.shape[0], 1))])
X_dev = np.hstack([X_dev, np.ones((X_dev.shape[0], 1))])

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

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

Question:

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

Answer:

(1) For the SVM, we do this to center each image relative to the rest of the dataset. This probably helps to keep gradients reasonable. Each input gets the same weights applied to it, so we want them to be trained on a "centered" dataset. For the KNN classifier, we don't need to do this, because we actually use the distance between input examples in order to calculate the output, and shifting everything by the same amount wouldn't change the relative distances or the outcome.

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 [10]:
```
from nndl.svm import SVM
```

In [11]:
```
# 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

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 [12]:
```
## 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 15569.977915410242.
```

In [13]:
```
loss = svm.loss(X_dev, y_dev)
print('The dev set loss is {}.'.format(loss))
```

```
The dev set loss is 15584.739555166354.
```

SVM gradient

In [14]:
```
#print(X_train.shape)
```

In [15]:
```
## 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 yo
svm.grad_check_sparse(X_dev, y_dev, grad)
```

```
numerical: -8.522250 analytic: -8.522249, relative error: 2.391779e-08
numerical: 6.125786 analytic: 6.125786, relative error: 1.320482e-08
numerical: -6.109695 analytic: -6.109695, relative error: 3.120147e-09
numerical: 5.110966 analytic: 5.110965, relative error: 7.267097e-08
numerical: 6.859735 analytic: 6.859735, relative error: 1.082414e-08
numerical: 4.314657 analytic: 4.314657, relative error: 2.091458e-08
numerical: 9.326083 analytic: 9.326082, relative error: 1.878566e-08
numerical: -15.829804 analytic: -15.829804, relative error: 1.659177e-08
numerical: 2.708752 analytic: 2.708752, relative error: 4.687877e-08
numerical: -15.970575 analytic: -15.970574, relative error: 6.427133e-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 [16]:
```
import time
```

In [17]:
```
#try to remove the j loop first

loss = 0.0

for i in range(0, X_train.shape[0]):
    scores = np.dot(X_train[i], svm.W.T) #(num_classes x 1 vector)
    correct_class_score = scores[y_train[i]] #scalar

    #print(y_dev[i])
    temp = (scores - correct_class_score)
    #print(temp.shape)
    z_js = np.ones_like(scores) + temp
    z_js[z_js < 0] = 0

    #now just need to sum up all the incorrect class terms in each row
    row_sum = np.sum(z_js[np.arange(len(z_js)) != y_train[i]])
    loss += row_sum

loss /= X_train.shape[0]

print(loss)
```

```
15569.977915410242
```

In [18]:
```
#now remove i loop

scores = np.dot(X_train, svm.W.T) #(num_samples x num_classes vector)
correct_class_score = np.choose(y_train, scores.T)
#print(correct_class_score.shape)

#now we want z_js to be num_samples x num_classes

#temp = (scores - correct_class_score.reshape(X_train.shape[0],1))
z_js = np.maximum(0, 1 + (scores - correct_class_score.reshape(X_train.shape[
#z_js[z_js < 0] = 0
#print(z_js.shape)

z_js[np.arange(X_train.shape[0]), y_train] = 0
row_sums = np.sum(z_js, axis=1)
#row_sums = np.sum(z_js, axis=1) - np.choose(y_train, z_js.T)
# print(row_sums.shape)
# print(z_js[y_train].shape)

loss = np.sum(row_sums)/X_train.shape[0]
print(loss)
```

```
15569.97791541023
```

In [21]:
```
## 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.lina

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_vectorize

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

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

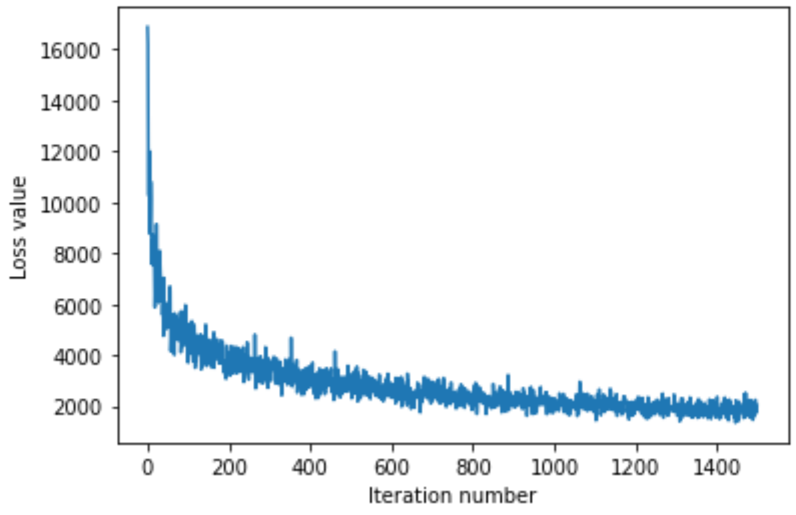
```
Normal loss / grad_norm: 15584.739555166354 / 2017.1983969632684 computed in
0.043683767318725586s
Vectorized loss / grad: 15584.73955516636 / 2017.1983969632681 computed in 0.
0040471553802490234s
difference in loss / grad: -5.4569682106375694e-12 / 3.618512721556333e-12
```

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 [22]:
```
# 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 16878.859625643898
iteration 100 / 1500: loss 3698.374429445832
iteration 200 / 1500: loss 3749.3722795434505
iteration 300 / 1500: loss 3232.2016048804485
iteration 400 / 1500: loss 2786.9285054561765
iteration 500 / 1500: loss 2911.8902158068004
iteration 600 / 1500: loss 2696.12348093388
iteration 700 / 1500: loss 2959.5756376002887
iteration 800 / 1500: loss 2512.8602634753483
iteration 900 / 1500: loss 2105.966992162575
iteration 1000 / 1500: loss 2313.4288377704447
iteration 1100 / 1500: loss 1732.4754464411596
iteration 1200 / 1500: loss 2114.4851353256367
iteration 1300 / 1500: loss 2050.9948002316255
iteration 1400 / 1500: loss 1814.3598110234702
That took 7.542051076889038s
```



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

In [23]:
```
## Implement svm.predict() and use it to compute the training and testing err

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: 0.29612244897959183
validation accuracy: 0.303
```

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 [24]:
```
# ================================================================ #
# 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 err
#
#   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
# ================================================================ #

#try learning rates between 10^-6 and 10^-3 in increments of 2.5e-6
learning_rates = np.linspace(1e-6, 1e-1, 100)

best_learning_rate = 0.0
best_val_accuracy = 0.0

for learning_rate in learning_rates:
    svm.train(X_train, y_train, learning_rate=learning_rate,
                  num_iters=1500, verbose=False)
    y_pred_validation = svm.predict(X_val)
    val_accuracy = np.mean(np.equal(y_val,y_pred_validation))
    print("Validation accuracy for {}: {}".format(learning_rate, val_accuracy

    if (val_accuracy > best_val_accuracy):
        #update
        best_val_accuracy = val_accuracy
        best_learning_rate = learning_rate

print('\n')

print("best learning rate is: {}".format(best_learning_rate))

#retrain and test on best learning rate
svm.train(X_train, y_train, learning_rate=best_learning_rate, num_iters=1500,
y_test_pred = svm.predict(X_test)
test_accuracy = np.mean(np.equal(y_test,y_test_pred))

print("Test accuracy at best rate is: {}".format(test_accuracy))


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

```
Validation accuracy for 1e-06: 0.142
Validation accuracy for 0.0010110909090909: 0.29
Validation accuracy for 0.0020211818181818182: 0.308
Validation accuracy for 0.003031272727272727: 0.282
Validation accuracy for 0.004041363636363636: 0.294
Validation accuracy for 0.005051454545454546: 0.334
Validation accuracy for 0.006061545454545454: 0.317
Validation accuracy for 0.007071636363636363: 0.307
Validation accuracy for 0.008081727272727272: 0.362
Validation accuracy for 0.00909181818181818: 0.262
Validation accuracy for 0.01010190909090909: 0.313
Validation accuracy for 0.011111999999999999: 0.275
Validation accuracy for 0.012122090909090907: 0.289
Validation accuracy for 0.013132181818181817: 0.329
Validation accuracy for 0.014142272727272726: 0.317
Validation accuracy for 0.015152363636363636: 0.288
Validation accuracy for 0.016162454545454546: 0.286
Validation accuracy for 0.017172545454545454: 0.285
Validation accuracy for 0.018182636363636363: 0.325
Validation accuracy for 0.019192727272727274: 0.277
Validation accuracy for 0.020202818181818183: 0.297
Validation accuracy for 0.02121290909090909: 0.264
Validation accuracy for 0.022223: 0.28
Validation accuracy for 0.023233090909090908: 0.279
Validation accuracy for 0.024243181818181817: 0.308
Validation accuracy for 0.02525327272727273: 0.285
Validation accuracy for 0.026263363636363637: 0.289
Validation accuracy for 0.027273454545454545: 0.325
Validation accuracy for 0.028283545454545454: 0.317
Validation accuracy for 0.029293636363636362: 0.32
Validation accuracy for 0.030303727272727274: 0.301
Validation accuracy for 0.03131381818181818: 0.327
Validation accuracy for 0.03232390909090909: 0.301
Validation accuracy for 0.033334: 0.263
Validation accuracy for 0.03434409090909091: 0.275
Validation accuracy for 0.03535418181818182: 0.29
Validation accuracy for 0.036364272727272724: 0.356
Validation accuracy for 0.037374363636363636: 0.283
Validation accuracy for 0.03838445454545455: 0.343
Validation accuracy for 0.03939454545454545: 0.335
Validation accuracy for 0.040404636363636365: 0.303
Validation accuracy for 0.04141472727272727: 0.301
Validation accuracy for 0.04242481818181818: 0.348
Validation accuracy for 0.04343490909090909: 0.317
Validation accuracy for 0.044445: 0.309
Validation accuracy for 0.04545509090909091: 0.301
Validation accuracy for 0.046465181818181815: 0.291
Validation accuracy for 0.04747527272727273: 0.327
Validation accuracy for 0.04848536363636363: 0.277
Validation accuracy for 0.049495454545454544: 0.278
Validation accuracy for 0.050505545454545456: 0.317
Validation accuracy for 0.05151563636363636: 0.313
Validation accuracy for 0.05252572727272727: 0.319
Validation accuracy for 0.05353581818181818: 0.268
Validation accuracy for 0.05454590909090909: 0.253
Validation accuracy for 0.055556: 0.324
Validation accuracy for 0.056566090909090906: 0.323
Validation accuracy for 0.05757618181818182: 0.309
Validation accuracy for 0.05858627272727272: 0.275
Validation accuracy for 0.059596363636363635: 0.309
Validation accuracy for 0.06060645454545455: 0.295
Validation accuracy for 0.06161654545454545: 0.281
Validation accuracy for 0.06262663636363636: 0.32
Validation accuracy for 0.06363672727272728: 0.328
Validation accuracy for 0.06464681818181818: 0.339
Validation accuracy for 0.06565690909090909: 0.264
Validation accuracy for 0.066667: 0.295
Validation accuracy for 0.06767709090909091: 0.311
Validation accuracy for 0.06868718181818181: 0.292
Validation accuracy for 0.06969727272727272: 0.331
Validation accuracy for 0.07070736363636364: 0.303
Validation accuracy for 0.07171745454545454: 0.31
Validation accuracy for 0.07272754545454545: 0.281
Validation accuracy for 0.07373763636363637: 0.364
Validation accuracy for 0.07474772727272727: 0.286
Validation accuracy for 0.07575781818181818: 0.306
Validation accuracy for 0.0767679090909091: 0.322
Validation accuracy for 0.077778: 0.288
Validation accuracy for 0.0787880909090909: 0.281
Validation accuracy for 0.07979818181818181: 0.344
Validation accuracy for 0.08080827272727273: 0.351
Validation accuracy for 0.08181836363636363: 0.274
Validation accuracy for 0.08282845454545454: 0.294
Validation accuracy for 0.08383854545454546: 0.292
Validation accuracy for 0.08484863636363636: 0.233
Validation accuracy for 0.08585872727272727: 0.285
Validation accuracy for 0.08686881818181819: 0.333
Validation accuracy for 0.08787890909090909: 0.343
Validation accuracy for 0.088889: 0.341
Validation accuracy for 0.0898990909090909: 0.337
Validation accuracy for 0.09090918181818182: 0.322
Validation accuracy for 0.09191927272727272: 0.282
Validation accuracy for 0.09292936363636363: 0.247
Validation accuracy for 0.09393945454545455: 0.297
Validation accuracy for 0.09494954545454545: 0.306
Validation accuracy for 0.09595963636363636: 0.288
Validation accuracy for 0.09696972727272726: 0.301
Validation accuracy for 0.09797981818181818: 0.279
Validation accuracy for 0.09898990909090909: 0.323
Validation accuracy for 0.1: 0.302


best learning rate is: 0.07373763636363637
iteration 0 / 1500: loss 16873.642067946166
iteration 100 / 1500: loss 101960.88214204952
iteration 200 / 1500: loss 79732.3597810737
iteration 300 / 1500: loss 79777.50339944643
iteration 400 / 1500: loss 157746.99857367005
iteration 500 / 1500: loss 130226.62841713522
iteration 600 / 1500: loss 146234.47744294538
iteration 700 / 1500: loss 162221.31380072358
iteration 800 / 1500: loss 141522.04199477856
iteration 900 / 1500: loss 157165.3695134706
iteration 1000 / 1500: loss 180628.80032503017
iteration 1100 / 1500: loss 72546.5969149875
iteration 1200 / 1500: loss 185275.660914508
iteration 1300 / 1500: loss 98091.18846531605
iteration 1400 / 1500: loss 74251.38742926178
```

In [184…

```
test_accuracy = np.mean(np.equal(y_test,y_test_pred))
print(test_accuracy)
```

```
0.27
```

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

"""
This code was based off of code from cs231n at Stanford University, and modif
"""
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,
        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 minibatch
    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 mea
      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 is 1/num_train * sum over all examples(sum over all classes not
        scores = np.dot(X[i], self.W.T)
        correct_class_score = scores[y[i]]
        hinge_terms = []
        for j in range(0, num_classes):
            if j == y[i]:
                continue
            else:
                class_score = scores[j]
                term = max(0, 1 + class_score - correct_class_score)
                hinge_terms.append(term)
```