一些说明

2019年3月30日 0:06

因为自已在做的过程中遇到一些问题，查了很多资料，希望稍微整理一下，方便在我之后想学这个课又看到我这篇笔记的小白
自己全都一步步做了一遍，加了代码的注释，包括一些函数的解释
大概是一个智障解释版的作业笔记
第一次写作业笔记
其实主要还是自己看……所以写的比较乱，但是尽量按顺序
3/30/19今天遇到一点问题把课翻出来看了一下，感觉受益匪浅
突然发现课程列表里还有一个python tutorial，好好学一下numpy!

配置

2019年3月29日　　23:40

我使用的是win10系统
安装最新python3版本（64位）
打开cs231n的assignment1主页
http://cs231n.github.io/assignments2018/assignment1/
下载代码压缩包
安装适合电脑的Anaconda版本（64位）
做作业时在anaconda的虚拟python环境下做，可以解决繁琐的python库的安装已经方便python环境的管理

```
conda create -n cs231n python=3.6 anaconda
```

to create an environment called `cs231n`.

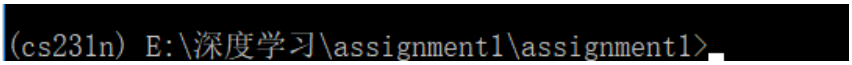
Then, to activate and enter the environment, run

```
source activate cs231n
```

To exit, you can simply close the window, or run

```
source deactivate cs231n
```

根据提示创建环境（就算电脑python版本高于3.7也可以创建一个3.6的环境，该作业发布者说了所有代码在python3.6环境下可以运行）



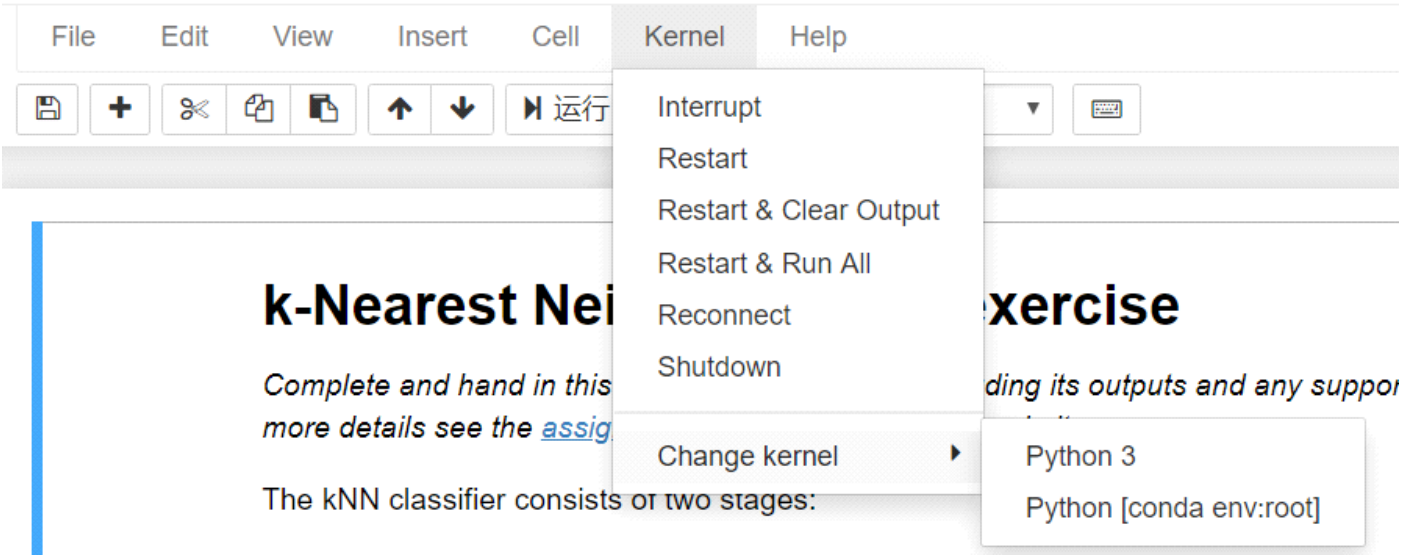
在anaconda prompt（安装anaconda后就能在任务栏找得到）窗口中，进入虚拟环境后前面会显示虚拟环境的名字，没进入之前是（base）
我在win10下的cmd进入虚拟环境不加source　　//activate cs231n
根据作业提示下载CIFA-10数据集

```
cd cs231n/datasets
./get_datasets.sh
```

在assignment1目录下输入 jupyter notebook即可进入jupyter的web编辑器开始做作业

感觉环境基本配置完成了，不知道有没有遗漏，记得一开始还折腾蛮久的。
之前想用linux虚拟机做，做到后面作业报了memory error，以为是虚拟机内存不够，回到win下面，重新配了一边环境做作业还是报memory error，加载一个数据集，10000*32*32*3 float64循环6次，算了一下也就1点多个G，查看任务管理器还有3点多可以用。折腾半天最后也不知道怎么弄好的emmmmm，不过应该在anaconda64位虚拟环境下python也是64位

的应该就没问题。
噢想来了，如果运行作业第一个cell报了importmodel error可以在anaconda prompt里面输入conda install nb_conda_kernels



然后jupyter notebook的kernel会多一项，选择python[conda env:root]重新运行服务器即可。

KNN

2019年3月30日 0:04

接前面配置说到的jupyter notebook打开，选择knn.ipynb

kNN就不说了，比较基础，后面作业要是碰到了难理解的算法再看情况做做解释吧

看着标记、注释、代码一步步做下去就好，每个代码cell都用shift+enter运行一下（必须要一步步运行下去）这里贴上运行结果

In [1]:
```
# Run some setup code for this notebook.
from __future__ import print_function
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt


# This is a bit of magic to make matplotlib figures appear inline in the notebook
# rather than in a new window.
%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# Some more magic so that the notebook will reload external python modules;
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

这是一些前期准备工作，注意看旁边的In【1】每一个cell在运行的时候会变成In【*】，运行结束之后会变成In【数字】，输出结果或者报错信息会显示在cell下面

最开始的时候就是这里报的 importmodel error，配置里面提到的，虽然问题解决的稀里糊涂，但是按照配置里说的做应该么得问题。

In [2]:
```
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

# Cleaning up variables to prevent loading data multiple times (which may cause memory issue)
try:
   del X_train, y_train
   del X_test, y_test
   print('Clear previously loaded data.')
except:
   pass

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,)
```

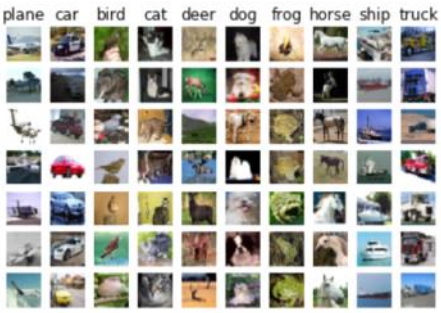
学习一下，删除了之前load的数据，避免内存不够

一开始就是这里报了memory error 贴上这些其实是因为卡太久了，放上来提醒自己（可直接跳过）

```
def load_CIFAR_batch(filename):
  """ load single batch of cifar """
  with open(filename, 'rb') as f:
    datadict = load_pickle(f)
    X = datadict['data']
    Y = datadict['labels']
    X = X.reshape(10000, 3, 32, 32).transpose(0,2,3,1).astype("float")
    Y = np.array(Y)
    return X, Y

def load_CIFAR10(ROOT):
  """ load all of cifar """
  xs = []
  ys = []
  for b in range(1,6):
    f = os.path.join(ROOT, 'data_batch_%d' % (b, ))
    X, Y = load_CIFAR_batch(f)
    xs.append(X)
    ys.append(Y)
  Xtr = np.concatenate(xs)
  Ytr = np.concatenate(ys)
  del X, Y
  Xte, Yte = load_CIFAR_batch(os.path.join(ROOT, 'test_batch'))
  return Xtr, Ytr, Xte, Yte
```

xs加了6次X，每次10000*3*32*32 类型是float64，240MB*6，大小应该不会超过可用内存，最开始的时候改了循环次数，训练集和测试集都被改成一样大了才不会报memory error……上网查了贼多资料（痛批某DN的水贴！）解决问题过程中，映像是没有改过电脑配置，内存也一直够，好像最后一次是重新配了一下虚拟环境anaconda和python都成了64位貌似就好了。

In [3]:
```
# Visualize some examples from the dataset.
# We show a few examples of training images from each class.
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
num_classes = len(classes)
samples_per_class = 7
for y, cls in enumerate(classes):
    idxs = np.flatnonzero(y_train == y)
    idxs = np.random.choice(idxs, samples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt_idx = i * num_classes + y + 1
        plt.subplot(samples_per_class, num_classes, plt_idx)
        plt.imshow(X_train[idx].astype('uint8'))
        plt.axis('off')
        if i == 0:
            plt.title(cls)
plt.show()
```



之前只会用plt画点图画曲线，这里学下显示图片的显示
挑了一些图片数据集显示看

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]
```

In [5]:
```
# 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)
```

挑了一部分数据用于后面的训练和测试
把图片数据换成2维

In [16]:
```
# Open cs231n/classifiers/k_nearest_neighbor.py and implement
# compute_distances_two_loops.

# Test your implementation:
dists = classifier.compute_distances_two_loops(X_test)
print(dists.shape)
```

```
(500, 5000)
```

这一步做之前，要完成home/cs231n/classifier文件夹下的k_nearest_neighbor.py
里的***two_loop两层循环函数

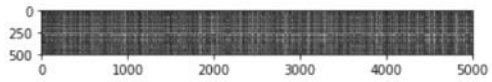
```
self.X_train = X
self.y_train = y
```

注意看该文件里的训练数据集和他们的标签集

```
num_test = X.shape[0]
num_train = self.X_train.shape[0]
dists = np.zeros((num_test, num_train))
for i in range(num_test):
    for j in range(num_train):
        #####################################################################
        # TODO:                                                             #
        # Compute the l2 distance between the ith test point and the jth    #
        # training point, and store the result in dists[i, j]. You should   #
        # not use a loop over dimension.                                    #
        #####################################################################
        dists[i, j] = np.sqrt(np.sum(np.square(self.X_train[j,:] - X[i,:])))
        # 计算两点之间的欧氏距离
        #####################################################################
        #                       END OF YOUR CODE                            #
        #####################################################################
return dists
```

这里用一个两层循环算一下每一个测试集里的数据与所有训练集里的数据的欧式距离

In [17]:
```
# We can visualize the distance matrix: each row is a single test example and
# its distances to training examples
plt.imshow(dists, interpolation='none')
plt.show()
```



输出结果，一张看不清的图

Inline Question #1: Notice the structured patterns in the distance matrix, where some rows or columns are visible brighter. (Note that with the default color scheme black indicates low distances while white indicates high distances.)

- What in the data is the cause behind the distinctly bright rows?
- What causes the columns?

Your Answer: *fill this in.* 某一行偏浅表示该测试样本与所有训练样本距离大 某一列偏浅表示该训练样本与所有测试样本距离大

Question1的答案.....

In [23]:
```
# Now implement the function predict_labels and run the code below:
# We use k = 1 (which is Nearest Neighbor).
y_test_pred = classifier.predict_labels(dists, k=1)
# Compute and print the fraction of correctly predicted examples
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
```

```
Got 137 / 500 correct => accuracy: 0.274000
```

You should expect to see approximately `27%` accuracy. Now lets try out a larger `k`, say `k = 5`:

算了一下27%的正确率，嗯哼，好菜

In [24]:
```
y_test_pred = classifier.predict_labels(dists, k=5)
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
```

```
Got 139 / 500 correct => accuracy: 0.278000
```

You should expect to see a slightly better performance than with `k = 1`.

这一步之前，先完成home/cs231n/classifier文件夹下的k_nearest_neighbor.py里的predict_labels类

```
num_test = dists.shape[0]
y_pred = np.zeros(num_test)
for i in range(num_test):
  # A list of length k storing the labels of the k nearest neighbors to
  # the ith test point.
  closest_y = []
  #########################################################################
  # TODO:                                                                 #
  # Use the distance matrix to find the k nearest neighbors of the ith    #
  # testing point, and use self.y_train to find the labels of these       #
  # neighbors. Store these labels in closest_y.                           #
  # Hint: Look up the function numpy.argsort.                             #
  #########################################################################
  closets_y = self.y_train[np.argsort(dists[i])[:k]]
  # argsort(dists[i])每一列从小到大排序，找到前k个最小的与第i个测试样本的
  # 训练样本的标签
  #########################################################################
  # TODO:                                                                 #
  # Now that you have found the labels of the k nearest neighbors, you    #
  # need to find the most common label in the list closest_y of labels.   #
  # Store this label in y_pred[i]. Break ties by choosing the smaller     #
  # label.                                                                #
  #########################################################################
  y_pred[i] = np.argmax(np.bincount(closets_y))
  # closets是一个1*k矩阵，里面存放的是标签，bincount求出每个标签出现的次数
  # 例如[1,2,1,1,1,1,2]T(转置符号),bincount求出来的是[0,5,2]（值为0,1,2）出
  # 现的次数,然后argmax求出最大值的下标，说明1就是出现次数最多的标签。
  #########################################################################
  #                           END OF YOUR CODE                            #
  #########################################################################
return y_pred
```

Inline Question 2 We can also other distance metrics such as L1 distance. The performance of a Nearest Neighbor classifier that uses L1 distance will not change if (Select all that apply.):

1. The data is preprocessed by subtracting the mean.
2. The data is preprocessed by subtracting the mean and dividing by the standard deviation.
3. The coordinate axes for the data are rotated.
4. None of the above.

Your Answer: 1.2 *Your explanation*: 前两种处理方法均是线性的，L2范数最小不变，L1范数最小也不会变，但是3是坐标轴旋转，比如向量(1,2)旋转成(0,g根号5),L1从3变成根号5，而L2不变。 ps.关于L0、1、2范数的解释可参考https://blog.csdn.net/Sakura55/article/details/80977090

做个题，了解一下L0、1、2 https://blog.csdn.net/Sakura55/article/details/80977090

In [26]:
```
# Now lets speed up distance matrix computation by using partial vectorization
# with one loop. Implement the function compute_distances_one_loop and run the
# code below:
dists_one = classifier.compute_distances_one_loop(X_test)

# To ensure that our vectorized implementation is correct, we make sure that it
# agrees with the naive implementation. There are many ways to decide whether
# two matrices are similar; one of the simplest is the Frobenius norm. In case
# you haven't seen it before, the Frobenius norm of two matrices is the square
# root of the squared sum of differences of all elements; in other words, reshape
# the matrices into vectors and compute the Euclidean distance between them.
difference = np.linalg.norm(dists - dists_one, ord='fro')
print('Difference was: %f' % (difference, ))
if difference < 0.001:
    print('Good! The distance matrices are the same')
else:
    print('Uh-oh! The distance matrices are different')
```

```
Difference was: 0.000000
Good! The distance matrices are the same
```

这一步之前先完成刚那个py文件里的一层循环函数，显然下面一步也是先完成无循环函数，（当初学计组的时候学到这一部分不以为然，看到后面的比较结果，还是有点小震撼）
学习了一下numpy的一些nb操作
一层循环和无循环代码：

```
def compute_distances_one_loop(self, X):
    """
    Compute the distance between each test point in X and each training point
    in self.X_train using a single loop over the test data.

    Input / Output: Same as compute_distances_two_loops
    """
    num_test = X.shape[0]
    num_train = self.X_train.shape[0]
    dists = np.zeros((num_test, num_train))
    for i in range(num_test):
        #######################################################################
        # TODO:                                                               #
        # Compute the l2 distance between the ith test point and all training #
        # points, and store the result in dists[i, :].                        #
        #######################################################################
        dists[i, :] = np.sqrt(np.sum(np.square(self.X_train - X[i,:]),axis = 1))
        # axis=0沿纵轴操作    axis=1沿横轴操作
        #######################################################################
        #                                 END OF YOUR CODE                    #
        #######################################################################
    return dists


num_test = X.shape[0]
num_train = self.X_train.shape[0]
dists = np.zeros((num_test, num_train))
#########################################################################
# TODO:                                                                 #
# Compute the l2 distance between all test points and all training      #
# points without using any explicit loops, and store the result in      #
# dists.                                                                #
#                                                                       #
# You should implement this function using only basic array operations; #
# in particular you should not use functions from scipy.                #
#                                                                       #
# HINT: Try to formulate the l2 distance using matrix multiplication    #
#       and two broadcast sums.                                         #
#########################################################################
dists = np.multiply(np.dot(X,self.X_train.T),-2)
# 测试集乘训练集的转置，所有元素再乘-2
sq1 = np.sum(np.square(X),axis=1,keepdims = True)
# 按行相加保持维度特性，得到一个列向量
sq2 = np.sum(np.square(self.X_train),axis=1)
dists = np.add(dists,sq1)
dists = np.add(dists,sq2)
dists = np.sqrt(dists)
# 根号((x1-y1)^2+(x2-y2)^) = x1^2+x2^2+y1^2+y2^2+(-2*(x1y1+x2y2))
#########################################################################
#                                 END OF YOUR CODE                      #
#########################################################################
return dists
```

In [35]:
```
# Let's compare how fast the implementations are
def time_function(f, *args):
    """
    Call a function f with args and return the time (in seconds) that it took to execute.
    """
    import time
    tic = time.time()
    f(*args)
    toc = time.time()
    return toc - tic

two_loop_time = time_function(classifier.compute_distances_two_loops, X_test)
print('Two loop version took %f seconds' % two_loop_time)

one_loop_time = time_function(classifier.compute_distances_one_loop, X_test)
print('One loop version took %f seconds' % one_loop_time)

no_loop_time = time_function(classifier.compute_distances_no_loops, X_test)
print('No loop version took %f seconds' % no_loop_time)

# you should see significantly faster performance with the fully vectorized implementation
```

```
Two loop version took 40.776955 seconds
One loop version took 87.773422 seconds
No loop version took 0.320144 seconds
```

对比三种算欧氏距离方法的运行时间，emmmmm整段垮掉。
一层循环竟然比二层循环时间长一倍多……说实话，代码和原csdn博主的一样，人家的一层循环时间只有一半。
我强行解释一下，我觉得……他们也许都是用linux做的作业emmmm可能不同的操作系统，数据在存储器里缓存的替换规则不一样……好tm扯……整段垮掉
但是no loop这个结果确实挺亮眼的，要我直接做我绝对两层循环，看了no loop的代码（里面注释有解释）就是用了一下完全平方差公式，加一点矩阵操作……路还很长啊

下面是交叉验证，确定最佳的K值

Cross-validation

We have implemented the k-Nearest Neighbor classifier but we set the value k = 5 arbitrarily. We will now determine the best value of this hyperparameter with cross-validation.

In [39]:
```
num_folds = 5
k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50, 100]

X_train_folds = []
y_train_folds = []
################################################################################
# TODO:                                                                        #
# Split up the training data into folds. After splitting, X_train_folds and    #
# y_train_folds should each be lists of length num_folds, where                #
# y_train_folds[i] is the label vector for the points in X_train_folds[i].     #
# Hint: Look up the numpy array_split function.                                #
################################################################################
# Your code
X_train_folds = np.array_split(X_train, num_folds)
y_train_folds = np.array_split(y_train, num_folds)
################################################################################
#                                 END OF YOUR CODE                             #
################################################################################

# A dictionary holding the accuracies for different values of k that we find
# when running cross-validation. After running cross-validation,
# k_to_accuracies[k] should be a list of length num_folds giving the different
# accuracy values that we found when using that value of k.
k_to_accuracies = {}
```

```
################################################################################
# TODO:                                                                        #
# Perform k-fold cross validation to find the best value of k. For each        #
# possible value of k, run the k-nearest-neighbor algorithm num_folds times,   #
# where in each case you use all but one of the folds as training data and the #
# last fold as a validation set. Store the accuracies for all fold and all     #
# values of k in the k_to_accuracies dictionary.                               #
################################################################################
# Your code
classifier = KNearestNeighbor()
for k in k_choices:
    accuracies = np.zeros(num_folds)
    for fold in range(num_folds):
        # range是直接开辟一个list,而xrange是每次调用生成一个调用对象
        temp_X = X_train_folds[:]
        temp_y = y_train_folds[:]
        X_validate_fold = temp_X.pop(fold)
        y_validate_fold = temp_y.pop(fold)
        # X_validate为每次取一折的数据

        temp_X = np.array([y for x in temp_X for y in x])
        temp_y = np.array([y for x in temp_y for y in x])
        # 该处temp_X为一个[array[],array[],array[]]类型的数组,
        # x为数组temp_X元素的地址即“array[],array[],array[]”
        # y在循环内为x中每个元素即每个array[]的地址
        # temp_X最终转化为[[],[],[]].T
        classifier.train(temp_X, temp_y)
        # 每次除掉取出的一折，剩下的作为训练集

        y_test_pred = classifier.predict(X_validate_fold, k=k)
        num_correct = np.sum(y_test_pred == y_validate_fold)
        # 分类器做出的结果与取出来的测试集进行比较验证
        accuracy = float(num_correct) / num_test

        accuracies[fold] = accuracy
    k_to_accuracies[k] = accuracies
################################################################################
#                                 END OF YOUR CODE                             #
################################################################################

# Print out the computed accuracies
for k in sorted(k_to_accuracies):
    for accuracy in k_to_accuracies[k]:
        print('k = %d, accuracy = %f' % (k, accuracy))
```

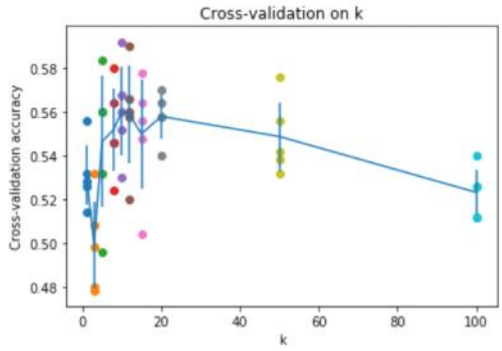
比较基础.....

下面是输出结果:

```
k = 1, accuracy = 0.526000
k = 1, accuracy = 0.514000
k = 1, accuracy = 0.528000
k = 1, accuracy = 0.556000
k = 1, accuracy = 0.532000
k = 3, accuracy = 0.478000
k = 3, accuracy = 0.498000
k = 3, accuracy = 0.480000
k = 3, accuracy = 0.532000
k = 3, accuracy = 0.508000
k = 5, accuracy = 0.496000
k = 5, accuracy = 0.532000
k = 5, accuracy = 0.560000
k = 5, accuracy = 0.584000
k = 5, accuracy = 0.560000
k = 8, accuracy = 0.524000
k = 8, accuracy = 0.564000
k = 8, accuracy = 0.546000
k = 8, accuracy = 0.580000
k = 8, accuracy = 0.546000
k = 10, accuracy = 0.530000
k = 10, accuracy = 0.592000
k = 10, accuracy = 0.552000
k = 10, accuracy = 0.568000
k = 10, accuracy = 0.560000
k = 12, accuracy = 0.520000
k = 12, accuracy = 0.590000
k = 12, accuracy = 0.558000
k = 12, accuracy = 0.566000
k = 12, accuracy = 0.560000
k = 15, accuracy = 0.504000
k = 15, accuracy = 0.578000
```

```
k = 15, accuracy = 0.578000
k = 15, accuracy = 0.556000
k = 15, accuracy = 0.564000
k = 15, accuracy = 0.548000
k = 20, accuracy = 0.540000
k = 20, accuracy = 0.558000
k = 20, accuracy = 0.558000
k = 20, accuracy = 0.564000
k = 20, accuracy = 0.570000
k = 50, accuracy = 0.542000
k = 50, accuracy = 0.576000
k = 50, accuracy = 0.556000
k = 50, accuracy = 0.538000
k = 50, accuracy = 0.532000
k = 100, accuracy = 0.512000
k = 100, accuracy = 0.540000
k = 100, accuracy = 0.526000
k = 100, accuracy = 0.512000
k = 100, accuracy = 0.526000
```

In [40]:
```
# plot the raw observations
for k in k_choices:
    accuracies = k_to_accuracies[k]
    plt.scatter([k] * len(accuracies), accuracies)

# plot the trend line with error bars that correspond to standard deviation
accuracies_mean = np.array([np.mean(v) for k,v in sorted(k_to_accuracies.items())])
accuracies_std = np.array([np.std(v) for k,v in sorted(k_to_accuracies.items())])
plt.errorbar(k_choices, accuracies_mean, yerr=accuracies_std)
plt.title('Cross-validation on k')
plt.xlabel('k')
plt.ylabel('Cross-validation accuracy')
plt.show()
```



画个图.....

In [50]:
```
# Based on the cross-validation results above, choose the best value for k,
# retrain the classifier using all the training data, and test it on the test
# data. You should be able to get above 28% accuracy on the test data.
best_k = 10

classifier = KNearestNeighbor()
classifier.train(X_train, y_train)
y_test_pred = classifier.predict(X_test, k=best_k)

# Compute and display the accuracy
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
```

```
Got 141 / 500 correct => accuracy: 0.282000
```

调下best_k测一下，10最好（其实也就那样……）

Inline Question 3 Which of the following statements about k-Nearest Neighbor (k-NN) are true in a classification setting, and for all k? Select all that apply.

1. The training error of a 1-NN will always be better than that of 5-NN.
2. The test error of a 1-NN will always be better than that of a 5-NN.
3. The decision boundary of the k-NN classifier is linear.
4. The time needed to classify a test example with the k-NN classifier grows with the size of the training set.
5. None of the above.

Your Answer: 1.4 *Your explanation*: 1.1-NN没有训练误差，因为1-NN只用他本身。5-NN会有，因为根据投票结果来判别 2.k越小，表示从数据里面获取的知识就更多，如果存在噪声，过拟合，模型的泛化能力就会越小 3.局部线性 4.显然

Question3做一下

kNN基本结束了，主要是练一下numpy的一些矩阵数组的操作，难度不大。感觉挺有意思的这个作业，再接再厉。

补充一下L1距离

Distance Metric to compare images

L1 distance: $d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$

test image			
56	32	10	18
90	23	128	133
24	26	178	200
2	0	255	220

\-

training image			
10	20	24	17
8	10	89	100
12	16	178	170
4	32	233	112

=

pixel-wise absolute value differences			
46	12	14	1
82	13	39	33
12	10	0	30
2	32	22	108

add → 456

L2距离公式

L2 (Euclidean) distance

$$d_2(I_1, I_2) = \sqrt{\sum_p (I_1^p - I_2^p)^2}$$

SVM

2019年3月31日 21:36

Multiclass Support Vector Machine exercise

Complete and hand in this completed worksheet (including its outputs and any supporting code outside of the worksheet) with your assignment submission. For more details see the assignments page on the course website.

In this exercise you will:

- implement a fully-vectorized **loss function** for the SVM
- implement the fully-vectorized expression for its **analytic gradient**
- **check your implementation** using numerical gradient
- use a validation set to **tune the learning rate and regularization** strength
- **optimize** the loss function with **SGD**
- **visualize** the final learned weights

多类支持向量机：
完成一个基于SVM的全向量化loss function
解析梯度全向量化表示
用数值梯度验证所求梯度 //表述不太清楚，之后再解释
用验证集去调优学习率和正则化强度
用SGD（随机梯度下降）优化loss function
可视化学习得到的权重

前三个cell的代码以及结果和前面的kNN一样，这里从第四个cell开始

In [7]:
```
# 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)
```

```
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,)
```

选出训练集，验证集，测试集，然后选出一小部分dev开发集，后面的训练基本都基于这个小开发集

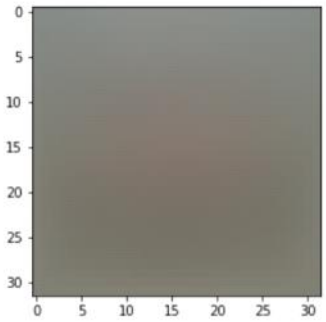
In [8]:
```
# Preprocessing: reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_val = np.reshape(X_val, (X_val.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

# 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)
```

转换成二维

In [9]:
```
# Preprocessing: subtract the mean image
# first: compute the image mean based on the training data
mean_image = np.mean(X_train, axis=0)
print(mean_image[:10]) # print a few of the elements
plt.figure(figsize=(4,4))
plt.imshow(mean_image.reshape((32,32,3)).astype('uint8')) # visualize the mean image
plt.show()
```

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



看一下图片数据的平均值

In [10]:
```
# 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 [11]:
```
# 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)
```

每一列表示一个样本在N维度上的特征，求出每一列的平均值，再减去平均值做预处理
训练集和测试集加上一维偏置维度（优化时可以同时考虑w和b，f（x） = wx+b）

完成 **cs231n/classifiers/linear_svm.py**里面的svm_loss_naive

```
dW = np.zeros(W.shape) # initialize the gradient as zero

# compute the loss and the gradient
num_classes = W.shape[1]
num_train = X.shape[0]
loss = 0.0
for i in range(num_train):
  scores = X[i].dot(W)
  correct_class_score = scores[y[i]]
  # 第i个样本对应某个标签的分数这里scores是一个向量，每个元素表示
  # 若该样本分类为该元素的下标，得分为该元素的值[1,4,6,3,2]第i个样本
  # 被分类为1,2,3,4,5得分分别为1,4,6,3,2
  for j in range(num_classes):
    if j == y[i]:
      continue
    margin = scores[j] - correct_class_score + 1 # note delta = 1
    # 设置delta等于1表示希望正确分类的分数至少比错误分类的分数大1
    if margin > 0:
      loss += margin
      dW[:,y[i]] += -X[i,:].T
      dW[:,j] += X[i,:].T
# Right now the loss is a sum over all training examples, but we want it
# to be an average instead so we divide by num_train.
loss /= num_train
dW /= num_train
# Add regularization to the loss.
loss += reg * np.sum(W * W)
dW += reg * W
```

这是一个朴素的方法，并没有用全向量化，之后会有比较，然后这里把下一个任务求梯度dW的代码也一起截进来来了，因为做的时候会发现dW和loss 一起求会更方便

这里小小的解释一下：

在给出类别预测前的输出结果是实数值， 也即根据 score function 得到的 score（$s = f(x_i, W)$），

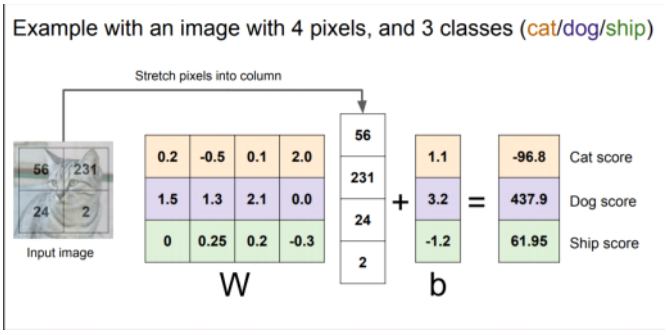
$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + \Delta), \quad \Delta = 1 \text{（一般情况下）}$$

- y_i 表示真实的类别，s_{y_i} 在真实类别上的得分；
- $s_j, j \neq y_i$ 在其他非真实类别上的得分，也即预测错误时的得分；

则在全体训练样本上的平均损失为：

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i$$

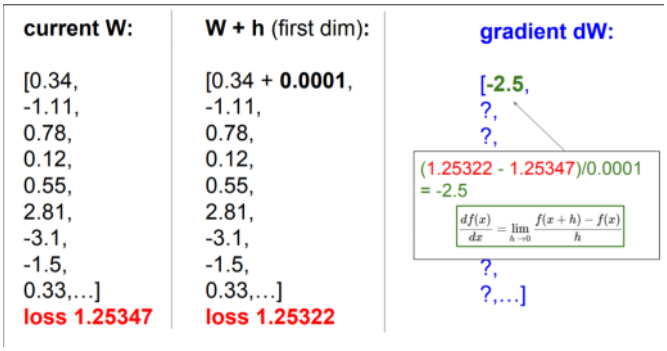
这是一个SVM的loss function的解释



这是一个求某一次迭代的loss的过程
原本的W是一个3*4的矩阵（在numpy里面维度和高等代数里面学到的矩阵维度正好相反）W的第i行×X的每一列（即每一个样本）的到的值为这个样本分到第i类所得到的分数，在代码里面W加了一维变成3*5的矩阵，假设原始训练集X的维度为4*1000（4个维度，1000个样本，预处理后变成5* 1000）原本得到3*1000的矩阵，现在仍然是3*1000，这样做就不用再额外加一个b的列向量。应该是很有道理……hhh

在loss function里面delta取1，在某种意义上来讲这个1是一个随机取的值，（说实话不是很懂，但是只要设置一个大于0的值就行）

然后关于求梯度矩阵dW，注意到loss function L，这不是一个连续可导函数，因此解析解的方法可能会有问题，正常来讲求梯度可以用数值的方法来求，即f（x+h）-f(x-h)/2h这里h可以用一个比较小的值如0.0000001



但是使用这种方法时需要遍历整个W矩阵，会很浪费时间，因此我们可以假设函数L可导，求出梯度矩阵dW之后再随机选取某几维的值与数值方法求出来的值进行比较，如果差距很小，则可认为我们的求导得到的梯度矩阵可行。

至于为什么求导后得到的梯度会是-xi.T（j == y[i]）和xi.T(j != y[i])呢

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$
$$= \sum_{j \neq y_i} \left[\max(0, w_j^T x_i - w_{y_i}^T x_i + 1) \right]$$

L对wj和wyi求偏导即可

求出loss和dW之后加上一个正则化处理，看该课的解释是以后会介绍.....
回到notebook的代码：

In [27]:
```
# Once you've implemented the gradient, recompute it with the code below
# and gradient check it with the function we provided for you

# Compute the loss and its gradient at W.
loss, grad = svm_loss_naive(W, X_dev, y_dev, 0.0)

# Numerically compute the gradient along several randomly chosen dimensions, and
# compare them with your analytically computed gradient. The numbers should match
# almost exactly along all dimensions.
from cs231n.gradient_check import grad_check_sparse
f = lambda w: svm_loss_naive(w, X_dev, y_dev, 0.0)[0]
grad_numerical = grad_check_sparse(f, W, grad)

# do the gradient check once again with regularization turned on
# you didn't forget the regularization gradient did you?
print('turn on reg')
loss, grad = svm_loss_naive(W, X_dev, y_dev, 5e1)
f = lambda w: svm_loss_naive(w, X_dev, y_dev, 5e1)[0]
grad_numerical = grad_check_sparse(f, W, grad)
```

```
<function <lambda> at 0x000001EF84CAD268>
numerical: -22.745992 analytic: -22.745992, relative error: 1.422867e-11
numerical: -20.754212 analytic: -20.754212, relative error: 8.073722e-12
numerical: -7.888957 analytic: -7.888957, relative error: 2.655239e-11
numerical: 15.256024 analytic: 15.256024, relative error: 9.650425e-12
numerical: -9.051240 analytic: -9.051240, relative error: 1.842163e-11
numerical: -12.158796 analytic: -12.158796, relative error: 4.137356e-11
numerical: -7.023781 analytic: -7.023781, relative error: 2.468743e-11
```

```
numerical: 15.256024 analytic: 15.256024, relative error: 9.650425e-12
numerical: -9.051240 analytic: -9.051240, relative error: 1.842163e-11
numerical: -12.158796 analytic: -12.158796, relative error: 4.137356e-11
numerical: -7.023781 analytic: -7.023781, relative error: 2.468743e-11
numerical: 9.172769 analytic: 9.172769, relative error: 4.313530e-11
numerical: -13.067263 analytic: -13.067263, relative error: 1.836753e-11
numerical: 2.551772 analytic: 2.529200, relative error: 4.442472e-03
turn on reg
numerical: -21.564201 analytic: -21.561846, relative error: 5.460961e-05
numerical: 19.992880 analytic: 19.999128, relative error: 1.562290e-04
numerical: 27.950028 analytic: 27.945387, relative error: 8.302970e-05
numerical: -14.510542 analytic: -14.507828, relative error: 9.353513e-05
numerical: -12.151671 analytic: -12.154353, relative error: 1.103375e-04
numerical: 3.184191 analytic: 3.186646, relative error: 3.854344e-04
numerical: 9.347640 analytic: 9.342472, relative error: 2.764959e-04
numerical: 21.094101 analytic: 21.095883, relative error: 4.222621e-05
numerical: -6.109513 analytic: -6.115938, relative error: 5.255090e-04
numerical: -1.680411 analytic: -1.683834, relative error: 1.017263e-03
```

这是梯度的对比结果（前半部分不加对比结果）其实不连续也么的问题.....

Inline Question 1:
It is possible that once in a while a dimension in the gradcheck will not match exactly. What could such a discrepancy be caused by? Is it a reason for concern? What is a simple example in one dimension where a gradient check could fail? How would change the margin affect of the frequency of this happening? *Hint: the SVM loss function is not strictly speaking differentiable*

Your Answer: *fill this in.*
loss function不是连续可导函数,在不连续点可能会出现差异，dh取足够小的值就能通过梯度检查。

加下来完成 cs231n/classifiers/linear_svm.py里面的svm_loss_vectorized

```
  scores = X.dot(W)
  num_classes = W.shape[1]
  num_train = X.shape[0]

  scores_correct = scores[np.arange(num_train),y]
  # 这里生成的scores_correct矩阵元素是每个样本的分类正确的分数
  # scores里面存的是每个样本分到某个类得的分数
  scores_correct = np.reshape(scores_correct,(num_train,-1))
  # scores_correct矩阵变成shape（num_train,1）
  margins = scores - scores_correct + 1
  # scores每一列是每个样本分到某一类的分数，每一列每个元素减去该样本分类正确
  # 得到的分数再加1比如[3.2,5.1,-1.7]分类正确为3.2
  margins = np.maximum(0,margins)
  # 经过处理后得到[1,2.9，0]
  margins[np.arange(num_train),y] = 0
  # 分类正确置0得到[0,2.9,0]即loss为2.9
  loss += np.sum(margins) / num_train
  loss += 0.5 * reg * np.sum(W * W)
  #############################################################################
  # TODO:
  # Implement a vectorized version of the gradient for the structured SVM      #
  # loss, storing the result in dW.                                             #
  #
  # Hint: Instead of computing the gradient from scratch, it may be easier     #
  # to reuse some of the intermediate values that you used to compute the      #
  # loss.
  #############################################################################
  margins[margins > 0] = 1
  row_sum = np.sum(margins, axis = 1)
  margins[np.arange(num_train),y] = -row_sum
  dW += np.dot(X.T,margins)/num_train + reg * W
  #############################################################################
  #                                 END OF YOUR CODE
  #############################################################################
```

一些高端的numpy数组操作，回到notebook

```
In [30]: # Next implement the function svm_loss_vectorized; for now only compute the loss;
         # we will implement the gradient in a moment.
         tic = time.time()
         loss_naive, grad_naive = svm_loss_naive(W, X_dev, y_dev, 0.000005)
         toc = time.time()
         print('Naive loss: %e computed in %fs' % (loss_naive, toc - tic))

         from cs231n.classifiers.linear_svm import svm_loss_vectorized
         tic = time.time()
         loss_vectorized, _ = svm_loss_vectorized(W, X_dev, y_dev, 0.000005)
         toc = time.time()
         print('Vectorized loss: %e computed in %fs' % (loss_vectorized, toc - tic))

         # The losses should match but your vectorized implementation should be much faster.
         print('difference: %f' % (loss_naive - loss_vectorized))

         Naive loss: 8.896348e+00 computed in 0.145121s
         Vectorized loss: 8.896348e+00 computed in 0.002999s
         difference: 0.000000
```

两种方法的比较

In [31]:
```
# Complete the implementation of svm_loss_vectorized, and compute the gradient
# of the loss function in a vectorized way.

# The naive implementation and the vectorized implementation should match, but
# the vectorized version should still be much faster.
tic = time.time()
_, grad_naive = svm_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Naive loss and gradient: computed in %fs' % (toc - tic))

tic = time.time()
_, grad_vectorized = svm_loss_vectorized(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Vectorized loss and gradient: computed in %fs' % (toc - tic))

# The loss is a single number, so it is easy to compare the values computed
# by the two implementations. The gradient on the other hand is a matrix, so
# we use the Frobenius norm to compare them.
difference = np.linalg.norm(grad_naive - grad_vectorized, ord='fro')
print('difference: %f' % difference)
```

```
Naive loss and gradient: computed in 0.146592s
Vectorized loss and gradient: computed in 0.004990s
difference: 0.000000
```

这次加上了梯度，全向量化一如既往的优秀

下面开始完成SGD的部分完成home/cs231n/classifier/linearclassifier.py

```
    num_train, dim = X.shape
    num_classes = np.max(y) + 1 # assume y takes values 0...K-1 where K is number of classes
    if self.W is None:
      # lazily initialize W
      self.W = 0.001 * np.random.randn(dim, num_classes)

    # Run stochastic gradient descent to optimize W
    loss_history = []
    for it in range(num_iters):
      X_batch = None
      y_batch = None

      #########################################################################
      # TODO:                                                                 #
      # Sample batch_size elements from the training data and their           #
      # corresponding labels to use in this round of gradient descent.        #
      # Store the data in X_batch and their corresponding labels in           #
      # y_batch; after sampling X_batch should have shape (dim, batch_size)   #
      # and y_batch should have shape (batch_size,)                           #
      #                                                                       #
      # Hint: Use np.random.choice to generate indices. Sampling with         #
      # replacement is faster than sampling without replacement.              #
      #########################################################################

      #########################################################################
      batch_inx = np.random.choice(num_train,batch_size)
      X_batch = X[batch_inx,:]
      y_batch = y[batch_inx]
      #########################################################################
      #                       END OF YOUR CODE                                #
      #########################################################################

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

      # perform parameter update
      #########################################################################
      # TODO:                                                                 #
      # Update the weights using the gradient and the learning rate.          #
      #########################################################################
      self.W = self.W - learning_rate * grad
      #########################################################################
      #                       END OF YOUR CODE                                #
      #########################################################################

      if verbose and it % 100 == 0:
        print('iteration %d / %d: loss %f' % (it, num_iters, loss))

    return loss_history
```

每次从训练集里面采样，作为这一步（一次迭代）的训练集，求出W，dW，然后W减去学习率*dW得到新的W，这样使得每次迭代loss逐渐减小直到收敛。

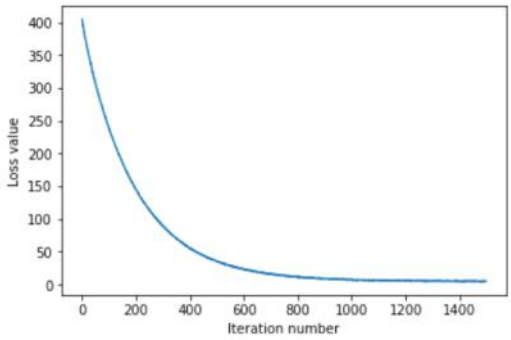
回到notebook

In [34]:
```
# In the file linear_classifier.py, implement SGD in the function
# LinearClassifier.train() and then run it with the code below.
from cs231n.classifiers import LinearSVM
svm = LinearSVM()
tic = time.time()
loss_hist = svm.train(X_train, y_train, learning_rate=1e-7, reg=2.5e4,
                      num_iters=1500, verbose=True)
toc = time.time()
print('That took %fs' % (toc - tic))
```

```
iteration 0 / 1500: loss 405.197025
iteration 100 / 1500: loss 237.707303
iteration 200 / 1500: loss 145.501002
iteration 300 / 1500: loss 89.585784
iteration 400 / 1500: loss 55.523742
iteration 500 / 1500: loss 35.515583
iteration 600 / 1500: loss 23.001688
iteration 700 / 1500: loss 15.642178
iteration 800 / 1500: loss 11.992715
iteration 900 / 1500: loss 8.907278
iteration 1000 / 1500: loss 7.493249
iteration 1100 / 1500: loss 6.306079
iteration 1200 / 1500: loss 5.575116
iteration 1300 / 1500: loss 5.569020
iteration 1400 / 1500: loss 4.968430
That took 9.786952s
```

看得出loss是逐渐减小然后收敛的

In [35]:
```
# A useful debugging strategy is to plot the loss as a function of
# iteration number:
plt.plot(loss_hist)
plt.xlabel('Iteration number')
plt.ylabel('Loss value')
plt.show()
```



这个图很直观了

然后完成predict函数，在刚刚那个py文件里面

```
y_pred = np.zeros(X.shape[0])
###########################################################################
# TODO:
# Implement this method. Store the predicted labels in y_pred.            #
###########################################################################
y_pred = np.argmax(X.dot(self.W),axis=1)
# argmax返回数组最大索引，axis=1按每一行算
###########################################################################
#                                   END OF YOUR CODE
###########################################################################
return y_pred
```

得分最高，即分为某个类

In [40]:
```
# Write the LinearSVM.predict function and evaluate the performance on both the
# training and validation set
y_train_pred = svm.predict(X_train)
print('training accuracy: %f' % (np.mean(y_train == y_train_pred), ))
y_val_pred = svm.predict(X_val)
print('validation accuracy: %f' % (np.mean(y_val == y_val_pred), ))
```

```
training accuracy: 0.382898
validation accuracy: 0.383000
```

训练集和验证集的准确率，看的出比kNN高了一些，期待做到后面会有比较惊艳的效果

下面使用验证集去调参，学习率和正则化强度

In [43]:
```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of about 0.4 on the validation set.
learning_rates = [1e-7, 0.75e-7,1.25e-7,0.5e-7,1.5e-7]
regularization_strengths = [3.5e4,4e4,4.5e4, 5e4,5.5e4,6e4]

# results is dictionary mapping tuples of the form
# (learning_rate, regularization_strength) to tuples of the form
# (training_accuracy, validation_accuracy). The accuracy is simply the fraction
# of data points that are correctly classified.
results = {}
best_val = -1   # The highest validation accuracy that we have seen so far.
best_svm = None # The LinearSVM object that achieved the highest validation rate.

################################################################################
# TODO:                                                                        #
# Write code that chooses the best hyperparameters by tuning on the validation #
# set. For each combination of hyperparameters, train a linear SVM on the      #
# training set, compute its accuracy on the training and validation sets, and  #
# store these numbers in the results dictionary. In addition, store the best   #
# validation accuracy in best_val and the LinearSVM object that achieves this  #
# accuracy in best_svm.                                                        #
#                                                                              #
# Hint: You should use a small value for num_iters as you develop your         #
# validation code so that the SVMs don't take much time to train; once you are #
# confident that your validation code works, you should rerun the validation   #

# Your code
for rate in learning_rates:
    for regular in regularization_strengths:
        svm = LinearSVM()
        svm.train(X_train, y_train, learning_rate = rate, reg=regular,
                  num_iters=1000)
        y_train_pred = svm.predict(X_train)
        accuracy_train = np.mean(y_train == y_train_pred)
        y_val_pred = svm.predict(X_val)
        accuracy_val = np.mean(y_val == y_val_pred)
        results[(rate, regular)] = (accuracy_train, accuracy_val)
        if(best_val < accuracy_val):
            best_val = accuracy_val
            best_svm = svm
################################################################################
#                              END OF YOUR CODE                                #
################################################################################

# Print out results.
for lr, reg in sorted(results):
    train_accuracy, val_accuracy = results[(lr, reg)]
    print('lr %e reg %e train accuracy: %f val accuracy: %f' % (
                lr, reg, train_accuracy, val_accuracy))

print('best validation accuracy achieved during cross-validation: %f' % best_val)
```

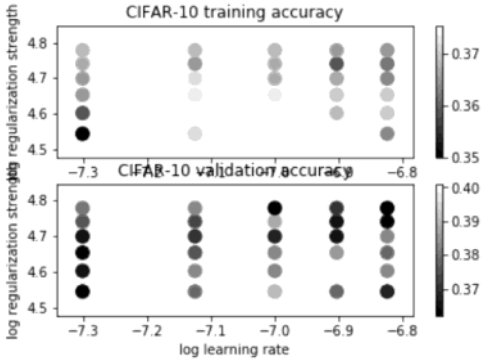
一开始选取一些学习率和正则化强度存在数组里，然后用验证集算出每种组合的loss

```
lr 5.000000e-08 reg 3.500000e+04 train accuracy: 0.349857 val accuracy: 0.363000
lr 5.000000e-08 reg 4.000000e+04 train accuracy: 0.358204 val accuracy: 0.366000
lr 5.000000e-08 reg 4.500000e+04 train accuracy: 0.365286 val accuracy: 0.364000
lr 5.000000e-08 reg 5.000000e+04 train accuracy: 0.365592 val accuracy: 0.365000
lr 5.000000e-08 reg 5.500000e+04 train accuracy: 0.367286 val accuracy: 0.376000
lr 5.000000e-08 reg 6.000000e+04 train accuracy: 0.368224 val accuracy: 0.380000
lr 7.500000e-08 reg 3.500000e+04 train accuracy: 0.370633 val accuracy: 0.379000
lr 7.500000e-08 reg 4.000000e+04 train accuracy: 0.374204 val accuracy: 0.383000
lr 7.500000e-08 reg 4.500000e+04 train accuracy: 0.372735 val accuracy: 0.375000
lr 7.500000e-08 reg 5.000000e+04 train accuracy: 0.371592 val accuracy: 0.370000
lr 7.500000e-08 reg 5.500000e+04 train accuracy: 0.364898 val accuracy: 0.374000
lr 7.500000e-08 reg 6.000000e+04 train accuracy: 0.368245 val accuracy: 0.383000
lr 1.000000e-07 reg 3.500000e+04 train accuracy: 0.375122 val accuracy: 0.389000
lr 1.000000e-07 reg 4.000000e+04 train accuracy: 0.374388 val accuracy: 0.382000
lr 1.000000e-07 reg 4.500000e+04 train accuracy: 0.373163 val accuracy: 0.383000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.367286 val accuracy: 0.368000
lr 1.000000e-07 reg 5.500000e+04 train accuracy: 0.367306 val accuracy: 0.388000
lr 1.000000e-07 reg 6.000000e+04 train accuracy: 0.367796 val accuracy: 0.363000
lr 1.250000e-07 reg 3.500000e+04 train accuracy: 0.375265 val accuracy: 0.378000
lr 1.250000e-07 reg 4.000000e+04 train accuracy: 0.368531 val accuracy: 0.401000
lr 1.250000e-07 reg 4.500000e+04 train accuracy: 0.369061 val accuracy: 0.385000
lr 1.250000e-07 reg 5.000000e+04 train accuracy: 0.366102 val accuracy: 0.366000
lr 1.250000e-07 reg 5.500000e+04 train accuracy: 0.358898 val accuracy: 0.366000
lr 1.250000e-07 reg 6.000000e+04 train accuracy: 0.365694 val accuracy: 0.371000
lr 1.500000e-07 reg 3.500000e+04 train accuracy: 0.363673 val accuracy: 0.367000
lr 1.500000e-07 reg 4.000000e+04 train accuracy: 0.369286 val accuracy: 0.383000
lr 1.500000e-07 reg 4.500000e+04 train accuracy: 0.369612 val accuracy: 0.379000
lr 1.500000e-07 reg 5.000000e+04 train accuracy: 0.363306 val accuracy: 0.383000
lr 1.500000e-07 reg 5.500000e+04 train accuracy: 0.361184 val accuracy: 0.364000
lr 1.500000e-07 reg 6.000000e+04 train accuracy: 0.364776 val accuracy: 0.362000
best validation accuracy achieved during cross-validation: 0.401000
```

输出结果，最优可达到40左右

In [45]:
```
# Visualize the cross-validation results
import math
x_scatter = [math.log10(x[0]) for x in results]
```

In [45]:
```
# Visualize the cross-validation results
import math
x_scatter = [math.log10(x[0]) for x in results]
y_scatter = [math.log10(x[1]) for x in results]

# plot training accuracy
marker_size = 100
colors = [results[x][0] for x in results]
plt.subplot(2, 1, 1)
plt.scatter(x_scatter, y_scatter, marker_size, c=colors)
plt.colorbar()
plt.xlabel('log learning rate')
plt.ylabel('log regularization strength')
plt.title('CIFAR-10 training accuracy')

# plot validation accuracy
colors = [results[x][1] for x in results] # default size of markers is 20
plt.subplot(2, 1, 2)
plt.scatter(x_scatter, y_scatter, marker_size, c=colors)
plt.colorbar()
plt.xlabel('log learning rate')
plt.ylabel('log regularization strength')
plt.title('CIFAR-10 validation accuracy')
plt.show()
```



画个图

In [46]:
```
# Evaluate the best svm on test set
y_test_pred = best_svm.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
```

```
linear SVM on raw pixels final test set accuracy: 0.378000
```

In [47]:
```
# Visualize the learned weights for each class.
# Depending on your choice of learning rate and regularization strength, these may
# or may not be nice to look at.
w = best_svm.W[:-1,:] # strip out the bias
w = w.reshape(32, 32, 3, 10)
w_min, w_max = np.min(w), np.max(w)
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
for i in range(10):
    plt.subplot(2, 5, i + 1)

    # Rescale the weights to be between 0 and 255
    wimg = 255.0 * (w[:, :, :, i].squeeze() - w_min) / (w_max - w_min)
    plt.imshow(wimg.astype('uint8'))
    plt.axis('off')
    plt.title(classes[i])
```

权重矩阵W通过一些列处理，再换回图形数据32*32*3（3为RGB数值）得到模型
记一句课上听到的话，以horse为例子，训练出来的模型，马有两个头，可能在数据集里面有的马头朝左，有的朝右，在线性分类器里面这些都会记录下来，所以会出现双头马的情况（据说在深度学习里面不会有这种情况）

SVM就结束了！