k-Nearest Neighbor (kNN) 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 (https://compsci682-fa19.github.io/assignments2019/assignment1) on the course website.

The kNN classifier consists of two stages:

- During training, the classifier takes the training data and simply remembers it
- During testing, kNN classifies every test image by comparing to all training images and transfering the labels of the k most similar training examples
- The value of k is cross-validated

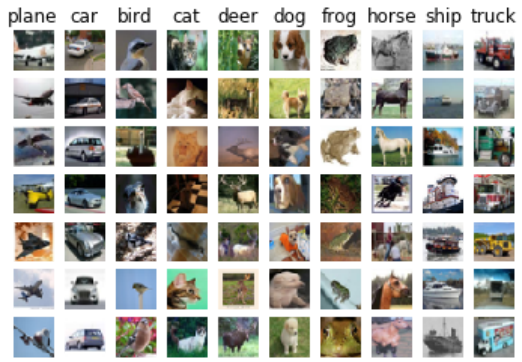
In this exercise you will implement these steps and understand the basic Image Classification pipeline, cross-validation, and gain proficiency in writing efficient, vectorized code.

In [1]:
```
# Run some setup code for this notebook.
from __future__ import print_function

import random
import numpy as np
from cs682.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
```

In [2]:
```
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs682/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,)
```

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



In [4]:
```
# Subsample the data for more efficient code execution in this exercise
num_training = 5000
mask = list(range(num_training))
X_train = X_train[mask]
y_train = y_train[mask]

num_test = 500
mask = list(range(num_test))
X_test = X_test[mask]
y_test = y_test[mask]
```

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)
```

In [6]:
```
from cs682.classifiers import KNearestNeighbor

# Create a kNN classifier instance.
# Remember that training a kNN classifier is a noop:
# the Classifier simply remembers the data and does no further processing
classifier = KNearestNeighbor()
classifier.train(X_train, y_train)
```

We would now like to classify the test data with the kNN classifier. Recall that we can break down this process into two steps:

1. First we must compute the distances between all test examples and all train examples.
2. Given these distances, for each test example we find the k nearest examples and have them vote for the label

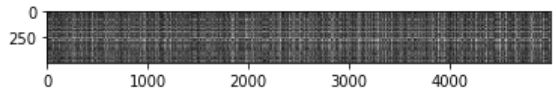
Lets begin with computing the distance matrix between all training and test examples. For example, if there are **Ntr** training examples and **Nte** test examples, this stage should result in a **Nte x Ntr** matrix where each element (i,j) is the distance between the i-th test and j-th train example.

First, open `cs682/classifiers/k_nearest_neighbor.py` and implement the function `compute_distances_two_loops` that uses a (very inefficient) double loop over all pairs of (test, train) examples and computes the distance matrix one element at a time.

In [7]:
```
# Open cs682/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)
```

In [8]:
```
# 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()
```



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:

1. The distinctly bright rows are test examples which are outliers with respect to the training set distribution - very far (having large l2 distances) from all training examples.
2. The distinctly bright columns are training examples which are outliers with respect to the test set distributions - very far (having large l2 distances) from all the test examples.

In [9]:
```
# 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`:

In [10]:
```
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`.

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. and 2.

Your explanation: \ In option 1, we add a constant (-mean) to data which would get cancelled out while computing the L1 distances \ L1' = x1-m - (x2-m) = x1-x2 = L1.

In option 2, we add a constant (-mean) as well as multiply by a constant (1/std_dev). The mean gets cancelled out while computing the L1 distance while all the distances get scaled by a factor of (1/std_dev). This affects all the distances in the same manner and does not affect their ordering which means we get the same set of nearest neighbors.\ L1' = (x1-m)/std_dev - (x2-m)/std_dev = (x1-x2)/std_dev = L1/std_dev

For option 3, lets say our points are (r1 cos(a), r1 sin(a)) and (r2 cos(b), r2 sin(b)). If we rotate the axes by d, the points become (r1 cos(a-d), r1 sin(a-d)) and (r2 cos(b-d), r2 sin(b-d))\ L1' = r1 (cos(a)cos(d) + sin(a)sin(d)) - r2 (cos(b)cos(d) + sin(b)sin(d)) + r1 (sin(a)cos(d) - cos(a)sin(d)) - r2 (sin(b)cos(d) - cos(b)sin(d))\ = (cos(d) - sin(d)) (r1 cos(a) - r2 cos(b) + r1 sin(a) - r2 sin(b)) + 2*r1 sin(a)sin(d) - 2*r2 sin(b)sin(d)\ = (cos(d) - sin(d))*L1 + 2*sin(d)*(r1 sin(a) - s2 sin(b))\ The above shows the L1 distances would change by a variable value which would affect the relative distances and the nearest k neighbors need not remain the same.

In [11]:
```
# 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
```

In [12]:
```
# Now implement the fully vectorized version inside compute_distances_no_loops
# and run the code
dists_two = classifier.compute_distances_no_loops(X_test)

# check that the distance matrix agrees with the one we computed before:
difference = np.linalg.norm(dists - dists_two, 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
```

In [13]:
```
# 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 30.059655 seconds
One loop version took 33.482554 seconds
No loop version took 0.173258 seconds
```

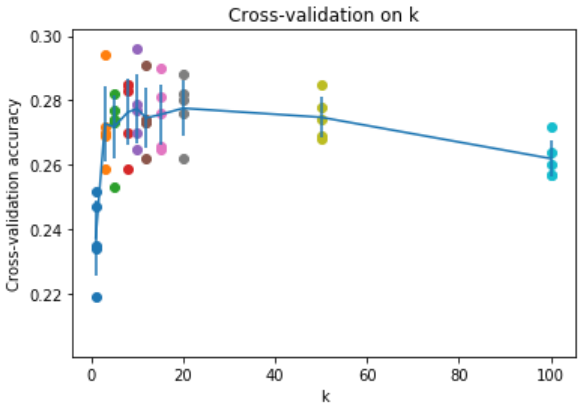
Cross-validation

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 [14]:
```
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.                                #
################################################################################
X_train_folds = np.array_split(X_train, num_folds)
# print(np.array(X_train_folds).shape)
y_train_folds = np.array_split(y_train, num_folds)
# print(np.array(y_train_folds).shape)
################################################################################
#                                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.                               #
################################################################################
for k in k_choices:
#     print (k, "\n")
    accuracies = []
    for validation_fold in range(num_folds):
#         print(validation_fold, "\n")
        num_validate = X_train_folds[validation_fold].shape[0];
        # Select all values from folds where the idx != validation_fold
        X_train_cv = np.array([X_fold for idx,X_fold in enumerate(X_train_folds) if idx!=validation_fo
ld])
        X_train_cv = X_train_cv.reshape(-1, X_train_cv.shape[-1])
        y_train_cv = np.array([y_fold for idx,y_fold in enumerate(y_train_folds) if idx!=validation_fo
ld])
        y_train_cv = y_train_cv.flatten()
#         print(X_train_cv.shape)
#         print(y_train_cv.shape)
        classifier.train(X_train_cv, y_train_cv)

        dists = classifier.compute_distances_no_loops(X_train_folds[validation_fold])
        y_valdiate_pred = classifier.predict_labels(dists, k=k+1)
        num_correct = np.sum(y_valdiate_pred == y_train_folds[validation_fold])
        accuracy = float(num_correct) / num_validate
        accuracies.append(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))
```

```
k = 1, accuracy = 0.235000
k = 1, accuracy = 0.219000
k = 1, accuracy = 0.234000
k = 1, accuracy = 0.247000
k = 1, accuracy = 0.252000
k = 3, accuracy = 0.259000
k = 3, accuracy = 0.270000
k = 3, accuracy = 0.269000
k = 3, accuracy = 0.294000
k = 3, accuracy = 0.272000
k = 5, accuracy = 0.253000
k = 5, accuracy = 0.277000
k = 5, accuracy = 0.274000
k = 5, accuracy = 0.273000
k = 5, accuracy = 0.282000
k = 8, accuracy = 0.259000
k = 8, accuracy = 0.283000
k = 8, accuracy = 0.270000
k = 8, accuracy = 0.285000
k = 8, accuracy = 0.285000
k = 10, accuracy = 0.265000
k = 10, accuracy = 0.296000
k = 10, accuracy = 0.277000
k = 10, accuracy = 0.279000
k = 10, accuracy = 0.270000
k = 12, accuracy = 0.262000
k = 12, accuracy = 0.291000
k = 12, accuracy = 0.273000
k = 12, accuracy = 0.274000
k = 12, accuracy = 0.273000
k = 15, accuracy = 0.265000
k = 15, accuracy = 0.276000
k = 15, accuracy = 0.290000
k = 15, accuracy = 0.281000
k = 15, accuracy = 0.266000
k = 20, accuracy = 0.262000
k = 20, accuracy = 0.280000
k = 20, accuracy = 0.282000
k = 20, accuracy = 0.276000
k = 20, accuracy = 0.288000
k = 50, accuracy = 0.274000
k = 50, accuracy = 0.285000
k = 50, accuracy = 0.278000
k = 50, accuracy = 0.269000
k = 50, accuracy = 0.268000
k = 100, accuracy = 0.257000
k = 100, accuracy = 0.272000
k = 100, accuracy = 0.264000
k = 100, accuracy = 0.257000
k = 100, accuracy = 0.260000
```

In [15]:
```
# 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()
```



In [21]:
```
# 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 = 6

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
```

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: 4

Your explanation:\ Option 1 is incorrect because each of the training examples in a 1-NN classifier would have the 0 distance to themselves and hence would always get correctly classified. Thus the training error for a 1-NN classifier would be 0. Depending on the distribution of the data, a 5-NN classifier could also achieve 0 training error if the data from different classes is very well separated such that the 5 closest points to a data point are always from the same class as the data point.

Option 2 is incorrect because the test error is a function of the distribution of test and train datasets and the value of k doesn't necessarily imply better or worse performance.

Option 3 is incorrect because a k-NN classifier could have a non-linear decision boundary. For example, consider a data set consisting of data points of the same class distributed in concentric circles. The decision boundary would also be a collection of concentric circles if the data is well separated.

Option 4 is correct as the classification step requires computing distances from all points in the dataset and then computing the closest k points. Thus, the time to classify is proportional to the size of the training set.

In [ ]:

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 (https://compsci682-fa19.github.io/assignments2019/assignment1/)](https://compsci682-fa19.github.io/assignments2019/assignment1/) 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

In [2]:
```
# Run some setup code for this notebook.
from __future__ import print_function
import random
import numpy as np
from cs682.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
```

CIFAR-10 Data Loading and Preprocessing

In [3]:
```
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs682/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,)
```

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', 'ship', 'truck']
num_classes = len(classes)
samples_per_class = 7
for y, cls in enumerate(classes):
    idxs = np.flatnonzero(y_train == y)
    idxs = np.random.choice(idxs, samples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt_idx = i * num_classes + y + 1
        plt.subplot(samples_per_class, num_classes, plt_idx)
        plt.imshow(X_train[idx].astype('uint8'))
        plt.axis('off')
        if i == 0:
            plt.title(cls)
plt.show()
```

In [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)
```

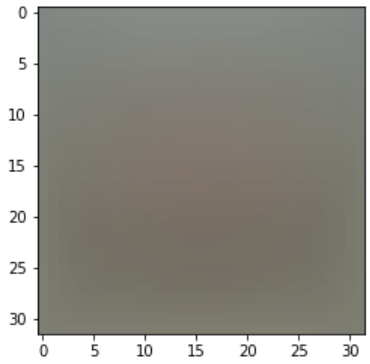
```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

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 mean image
plt.show()
```

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



In [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)
```

SVM Classifier

Your code for this section will all be written inside **cs682/classifiers/linear_svm.py**.

As you can see, we have prefilled the function `svm_loss_naive` which uses for loops to evaluate the multiclass SVM loss function.

In [10]:
```
# Evaluate the naive implementation of the loss we provided for you:
from cs682.classifiers.linear_svm import svm_loss_naive
import time

# generate a random SVM weight matrix of small numbers
W = np.random.randn(3073, 10) * 0.0001

loss, grad = svm_loss_naive(W, X_dev, y_dev, 0.000005)
print('loss: %f' % (loss, ))
```

```
loss: 9.427342
```

The `grad` returned from the function above is right now all zero. Derive and implement the gradient for the SVM cost function and implement it inline inside the function `svm_loss_naive`. You will find it helpful to interleave your new code inside the existing function.

To check that you have correctly implemented the gradient correctly, you can numerically estimate the gradient of the loss function and compare the numeric estimate to the gradient that you computed. We have provided code that does this for you:

In [11]:
```
# 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 cs682.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?
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)
```

```
numerical: 5.643907 analytic: 5.643907, relative error: 4.990735e-12
numerical: -31.707712 analytic: -31.707712, relative error: 2.414137e-12
numerical: -15.094566 analytic: -15.094566, relative error: 6.928050e-12
numerical: 6.590847 analytic: 6.590847, relative error: 6.172296e-11
numerical: -0.195251 analytic: -0.195251, relative error: 4.938329e-11
numerical: 23.630563 analytic: 23.630563, relative error: 1.323223e-11
numerical: -12.854727 analytic: -12.854727, relative error: 8.826560e-12
numerical: -12.317041 analytic: -12.317041, relative error: 2.239764e-11
numerical: 5.725626 analytic: 5.725626, relative error: 2.955116e-11
numerical: 13.582976 analytic: 13.582976, relative error: 2.564645e-11
numerical: 34.546979 analytic: 34.546979, relative error: 1.409610e-11
numerical: -9.059835 analytic: -9.059835, relative error: 9.190069e-12
numerical: 12.961169 analytic: 12.961169, relative error: 4.125498e-11
numerical: 13.339584 analytic: 13.339584, relative error: 4.944858e-12
numerical: 1.247810 analytic: 1.247810, relative error: 2.075525e-11
numerical: 9.514842 analytic: 9.514842, relative error: 2.153143e-11
numerical: -19.383491 analytic: -19.383491, relative error: 4.305561e-12
numerical: -1.687203 analytic: -1.687203, relative error: 5.879986e-11
numerical: 15.154097 analytic: 15.154097, relative error: 2.513574e-11
numerical: -22.494673 analytic: -22.494673, relative error: 1.254444e-11
```

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: While calculating the analytic gradient, if the loss has a small positive value (lets call this f(x) = e). While calculating the numerical gradient, if we use a value of h > e then f(x) - h would cross over the 0 (the location of the kink in SVM loss) and introduce a 0 contribution towards the gradient. Hence, there will be a discrepancy during gradcheck. The same can occur for a small negative value of loss when f(x)-h crosses over 0 and incurs an error in calculation. We could easily get around this by considering fewer datapoints while validating the gradients thereby reducing the chances of our datapoints lying around the kink. Increasing the margin might lead to increase in the frequency of this happening as there would be more changes of a data point not belonging to the correct class being very close to the margin.

In [12]:
```
# 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 cs682.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: 9.427342e+00 computed in 0.142856s
Vectorized loss: 9.427342e+00 computed in 0.009900s
difference: 0.000000
```

In [13]:
```
# 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 0.131190s
Vectorized loss and gradient: computed in 0.003910s
difference: 0.000000
```

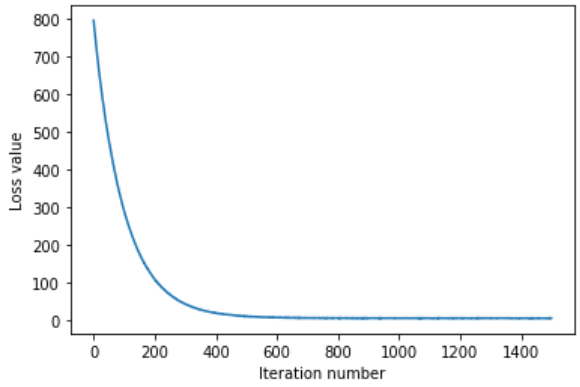
Stochastic Gradient Descent

We now have vectorized and efficient expressions for the loss, the gradient and our gradient matches the numerical gradient. We are therefore ready to do SGD to minimize the loss.

In [14]:
```
# In the file linear_classifier.py, implement SGD in the function
# LinearClassifier.train() and then run it with the code below.
from cs682.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 794.312538
iteration 100 / 1500: loss 287.904655
iteration 200 / 1500: loss 108.227064
iteration 300 / 1500: loss 42.795420
iteration 400 / 1500: loss 19.172778
iteration 500 / 1500: loss 10.514195
iteration 600 / 1500: loss 7.156890
iteration 700 / 1500: loss 5.543028
iteration 800 / 1500: loss 5.214884
iteration 900 / 1500: loss 5.433855
iteration 1000 / 1500: loss 5.367930
iteration 1100 / 1500: loss 5.485583
iteration 1200 / 1500: loss 5.316967
iteration 1300 / 1500: loss 5.635412
iteration 1400 / 1500: loss 5.149256
That took 2.742559s
```

In [15]:
```
# 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()
```



In [16]:
```
# 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), ))
```

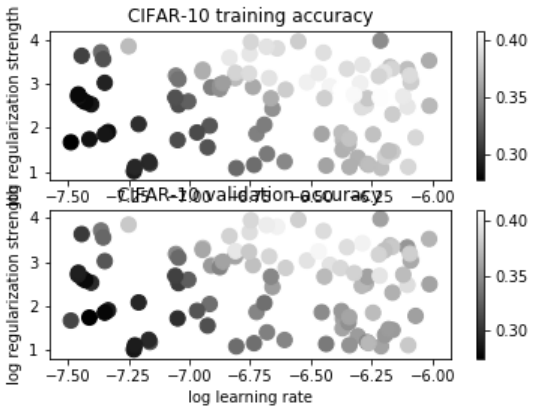
```
training accuracy: 0.371959
validation accuracy: 0.374000
```

In [45]:
```
# 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 = [4e-7,5e-7, 6e-7, 7e-7, 7.5e-7, 8e-7, 8.5e-7, 9e-7, 9.5e-7, 1e-6]
# regularization_strengths = [1e1,5e1,1e2,5e2,1e3,5e3, 1e4, 2e4]s

# 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   #
# code with a larger value for num_iters.                                      #
################################################################################
# Setting the seed to ensure consistent behavior across runs with same params
np.random.seed(123)
num_trials = 100
for t in range(num_trials):
#   print ('learning rate: %f, regularization strength: %f' %(lr, rs))
    lr = 10 ** np.random.uniform (-7.5, -6)
    rs = 10 ** np.random.uniform (1,4)
    svm = LinearSVM()
    loss_hist = svm.train(X_train, y_train, learning_rate=lr, reg=rs,
                num_iters=2000, batch_size=400, verbose=False)
    y_train_pred = svm.predict(X_train)
    train_acc = np.mean(y_train == y_train_pred)
#   print('training accuracy: %f' % (train_acc))
    y_val_pred = svm.predict(X_val)
    val_acc = np.mean(y_val == y_val_pred)
#   print('validation accuracy: %f' % (val_acc))
    results [(lr, rs)] = (train_acc, val_acc)
    if (val_acc > best_val):
        best_val = val_acc
        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)
```

```
lr 3.251255e-08 reg 4.668452e+01 train accuracy: 0.277184 val accuracy: 0.311000
lr 3.478788e-08 reg 5.137424e+02 train accuracy: 0.289204 val accuracy: 0.289000
lr 3.489476e-08 reg 5.721079e+02 train accuracy: 0.281510 val accuracy: 0.291000
lr 3.601438e-08 reg 4.329056e+03 train accuracy: 0.310653 val accuracy: 0.305000
lr 3.634694e-08 reg 4.207594e+02 train accuracy: 0.281388 val accuracy: 0.291000
lr 3.727577e-08 reg 3.857726e+02 train accuracy: 0.281898 val accuracy: 0.279000
lr 3.874409e-08 reg 5.461223e+01 train accuracy: 0.286000 val accuracy: 0.274000
lr 3.941344e-08 reg 3.390967e+02 train accuracy: 0.288122 val accuracy: 0.297000
lr 4.325203e-08 reg 5.212938e+03 train accuracy: 0.331776 val accuracy: 0.337000
lr 4.416854e-08 reg 3.617168e+03 train accuracy: 0.318265 val accuracy: 0.329000
lr 4.480725e-08 reg 7.048268e+01 train accuracy: 0.288469 val accuracy: 0.278000
lr 4.482619e-08 reg 1.051723e+03 train accuracy: 0.294755 val accuracy: 0.320000
lr 4.631187e-08 reg 8.015565e+01 train accuracy: 0.283694 val accuracy: 0.288000
lr 5.619378e-08 reg 7.101248e+03 train accuracy: 0.372755 val accuracy: 0.381000
lr 5.905107e-08 reg 1.022389e+01 train accuracy: 0.291959 val accuracy: 0.286000
lr 5.964594e-08 reg 1.278224e+01 train accuracy: 0.297367 val accuracy: 0.289000
lr 6.182823e-08 reg 1.207073e+02 train accuracy: 0.300020 val accuracy: 0.294000
lr 6.800233e-08 reg 1.708376e+01 train accuracy: 0.298224 val accuracy: 0.296000
lr 6.874855e-08 reg 1.522251e+01 train accuracy: 0.306102 val accuracy: 0.315000
lr 8.744384e-08 reg 4.855520e+02 train accuracy: 0.318551 val accuracy: 0.309000
lr 8.800857e-08 reg 1.525428e+03 train accuracy: 0.343735 val accuracy: 0.347000
lr 8.944845e-08 reg 5.222336e+01 train accuracy: 0.315980 val accuracy: 0.298000
lr 9.013629e-08 reg 1.255389e+03 train accuracy: 0.335571 val accuracy: 0.330000
lr 9.020229e-08 reg 3.310228e+02 train accuracy: 0.316633 val accuracy: 0.306000
lr 9.893984e-08 reg 3.961656e+02 train accuracy: 0.326286 val accuracy: 0.324000
lr 1.075437e-07 reg 7.783641e+01 train accuracy: 0.315020 val accuracy: 0.319000
lr 1.132648e-07 reg 1.888104e+03 train accuracy: 0.365939 val accuracy: 0.363000
lr 1.135399e-07 reg 2.088986e+03 train accuracy: 0.368571 val accuracy: 0.374000
lr 1.188303e-07 reg 3.583559e+01 train accuracy: 0.319694 val accuracy: 0.317000
lr 1.212182e-07 reg 1.109603e+02 train accuracy: 0.321612 val accuracy: 0.333000
lr 1.229384e-07 reg 8.309898e+02 train accuracy: 0.342286 val accuracy: 0.356000
lr 1.334196e-07 reg 7.648317e+02 train accuracy: 0.350347 val accuracy: 0.344000
lr 1.341470e-07 reg 9.542580e+02 train accuracy: 0.352714 val accuracy: 0.369000
lr 1.372996e-07 reg 1.022020e+03 train accuracy: 0.357592 val accuracy: 0.361000
lr 1.461940e-07 reg 3.897038e+03 train accuracy: 0.395306 val accuracy: 0.386000
lr 1.541637e-07 reg 1.339302e+03 train accuracy: 0.371571 val accuracy: 0.361000
lr 1.548008e-07 reg 1.771345e+03 train accuracy: 0.384980 val accuracy: 0.397000
lr 1.560149e-07 reg 1.205858e+01 train accuracy: 0.330571 val accuracy: 0.332000
lr 1.759830e-07 reg 1.707677e+03 train accuracy: 0.389102 val accuracy: 0.381000
lr 1.769711e-07 reg 4.778015e+03 train accuracy: 0.394163 val accuracy: 0.392000
lr 1.790853e-07 reg 8.932212e+03 train accuracy: 0.385735 val accuracy: 0.389000
lr 1.826721e-07 reg 8.478981e+02 train accuracy: 0.362918 val accuracy: 0.376000
lr 1.845477e-07 reg 1.426261e+01 train accuracy: 0.332592 val accuracy: 0.333000
lr 1.889344e-07 reg 7.298913e+01 train accuracy: 0.342408 val accuracy: 0.345000
lr 2.035732e-07 reg 1.172258e+02 train accuracy: 0.340898 val accuracy: 0.333000
lr 2.078649e-07 reg 6.924201e+03 train accuracy: 0.392347 val accuracy: 0.400000
lr 2.086973e-07 reg 2.527523e+01 train accuracy: 0.339265 val accuracy: 0.342000
lr 2.129682e-07 reg 1.342818e+03 train accuracy: 0.392449 val accuracy: 0.388000
lr 2.165040e-07 reg 4.664767e+02 train accuracy: 0.365020 val accuracy: 0.346000
lr 2.196915e-07 reg 2.704577e+02 train accuracy: 0.350980 val accuracy: 0.358000
lr 2.295802e-07 reg 6.368351e+03 train accuracy: 0.389061 val accuracy: 0.403000
lr 2.465536e-07 reg 1.692118e+01 train accuracy: 0.346612 val accuracy: 0.345000
lr 2.490223e-07 reg 7.355208e+02 train accuracy: 0.381633 val accuracy: 0.384000
lr 2.807149e-07 reg 9.202233e+03 train accuracy: 0.381367 val accuracy: 0.385000
lr 3.046898e-07 reg 1.079173e+03 train accuracy: 0.398939 val accuracy: 0.395000
lr 3.402921e-07 reg 1.736155e+03 train accuracy: 0.396714 val accuracy: 0.405000
lr 3.505127e-07 reg 7.218019e+01 train accuracy: 0.360551 val accuracy: 0.348000
lr 3.563784e-07 reg 5.091546e+03 train accuracy: 0.390347 val accuracy: 0.391000
lr 3.594539e-07 reg 4.665094e+02 train accuracy: 0.382388 val accuracy: 0.373000
lr 3.654896e-07 reg 1.373534e+01 train accuracy: 0.360286 val accuracy: 0.339000
lr 3.789462e-07 reg 8.655728e+02 train accuracy: 0.401429 val accuracy: 0.395000
lr 4.187680e-07 reg 7.297520e+02 train accuracy: 0.407020 val accuracy: 0.389000
lr 4.229739e-07 reg 7.705630e+01 train accuracy: 0.373306 val accuracy: 0.355000
lr 4.268465e-07 reg 3.109065e+01 train accuracy: 0.366776 val accuracy: 0.361000
lr 4.281898e-07 reg 5.203413e+01 train accuracy: 0.371082 val accuracy: 0.360000
lr 4.451041e-07 reg 2.180340e+03 train accuracy: 0.388796 val accuracy: 0.392000
lr 4.520011e-07 reg 4.252222e+01 train accuracy: 0.367939 val accuracy: 0.351000
lr 4.549080e-07 reg 5.627173e+01 train accuracy: 0.376939 val accuracy: 0.362000
lr 4.558419e-07 reg 1.286003e+01 train accuracy: 0.366327 val accuracy: 0.352000
lr 4.772910e-07 reg 5.604356e+02 train accuracy: 0.403633 val accuracy: 0.386000
lr 4.928970e-07 reg 2.082830e+01 train accuracy: 0.366469 val accuracy: 0.357000
lr 5.050506e-07 reg 1.025022e+03 train accuracy: 0.399020 val accuracy: 0.410000
lr 5.122127e-07 reg 2.883737e+02 train accuracy: 0.389939 val accuracy: 0.372000
lr 5.153676e-07 reg 2.686289e+03 train accuracy: 0.393490 val accuracy: 0.403000
lr 5.330233e-07 reg 3.240582e+02 train accuracy: 0.389796 val accuracy: 0.369000
lr 5.393970e-07 reg 1.505624e+01 train accuracy: 0.369857 val accuracy: 0.344000
```

```
lr 5.408464e-07 reg 4.649674e+02 train accuracy: 0.378143 val accuracy: 0.360000
lr 5.462577e-07 reg 1.431611e+01 train accuracy: 0.373878 val accuracy: 0.372000
lr 5.701436e-07 reg 7.038480e+01 train accuracy: 0.375551 val accuracy: 0.374000
lr 6.023699e-07 reg 1.480942e+03 train accuracy: 0.384837 val accuracy: 0.387000
lr 6.070402e-07 reg 4.885039e+02 train accuracy: 0.405612 val accuracy: 0.396000
lr 6.117351e-07 reg 9.572465e+03 train accuracy: 0.348286 val accuracy: 0.360000
lr 6.149036e-07 reg 1.213045e+01 train accuracy: 0.374082 val accuracy: 0.361000
lr 6.341515e-07 reg 1.837654e+03 train accuracy: 0.392469 val accuracy: 0.376000
lr 6.382017e-07 reg 2.197040e+01 train accuracy: 0.383551 val accuracy: 0.355000
lr 6.603845e-07 reg 2.987210e+01 train accuracy: 0.374204 val accuracy: 0.357000
lr 6.701202e-07 reg 2.410460e+03 train accuracy: 0.374612 val accuracy: 0.377000
lr 6.830692e-07 reg 2.209866e+03 train accuracy: 0.384857 val accuracy: 0.381000
lr 6.854474e-07 reg 2.020754e+02 train accuracy: 0.381102 val accuracy: 0.368000
lr 6.864593e-07 reg 2.847938e+01 train accuracy: 0.379449 val accuracy: 0.356000
lr 7.591890e-07 reg 5.169790e+02 train accuracy: 0.396102 val accuracy: 0.374000
lr 7.887476e-07 reg 2.162084e+03 train accuracy: 0.373735 val accuracy: 0.354000
lr 7.899653e-07 reg 3.022374e+02 train accuracy: 0.394020 val accuracy: 0.392000
lr 7.958802e-07 reg 1.284899e+01 train accuracy: 0.386388 val accuracy: 0.381000
lr 8.082795e-07 reg 1.553257e+03 train accuracy: 0.388510 val accuracy: 0.384000
lr 8.145460e-07 reg 1.077362e+03 train accuracy: 0.381163 val accuracy: 0.364000
lr 8.199476e-07 reg 1.129479e+03 train accuracy: 0.390531 val accuracy: 0.395000
lr 8.535624e-07 reg 5.961523e+01 train accuracy: 0.386653 val accuracy: 0.359000
lr 9.702025e-07 reg 3.358292e+03 train accuracy: 0.364816 val accuracy: 0.370000
lr 9.738339e-07 reg 3.141903e+02 train accuracy: 0.371510 val accuracy: 0.361000
best validation accuracy achieved during cross-validation: 0.410000
```

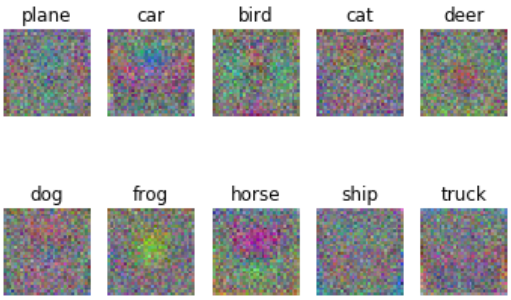
In [46]:
```
# Visualize the cross-validation results
import math
x_scatter = [math.log10(x[0]) for x in results]
y_scatter = [math.log10(x[1]) for x in results]

# 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()
```



In [47]:
```
# 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: 0.388000
```

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



Inline question 2:

Describe what your visualized SVM weights look like, and offer a brief explanation for why they look they way that they do.

Your answer: The weights look like a combination of various different classes with dominant colors and gradients of the correct class. This reflects the various tones in the images that the classifier is using to differentiate the correct class from all other classes, thereby trying to find tones that exist in one and not in others.

In [ ]:

Softmax 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 (https://compsci682-fa19.github.io/assignments2019/assignment1/) on the course website.

This exercise is analogous to the SVM exercise. You will:

- implement a fully-vectorized **loss function** for the Softmax classifier
- implement the fully-vectorized expression for its **analytic gradient**
- **check your implementation** with 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

In [3]:
```
from __future__ import print_function
import random
import numpy as np
from cs682.data_utils import load_CIFAR10
import matplotlib.pyplot as plt


%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'

# for auto-reloading extenrnal modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

In [4]:
```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000, num_dev=500):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    it for the linear classifier. These are the same steps as we used for the
    SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = 'cs682/datasets/cifar-10-batches-py'

    X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

    # subsample the data
    mask = list(range(num_training, num_training + num_validation))
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = list(range(num_training))
    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


# 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

# Invoke the above function to get our data.
X_train, y_train, X_val, y_val, X_test, y_test, X_dev, y_dev = get_CIFAR10_data()
print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
print('dev data shape: ', X_dev.shape)
print('dev labels shape: ', y_dev.shape)
```

```
Train data shape:  (49000, 3073)
Train labels shape:  (49000,)
Validation data shape:  (1000, 3073)
Validation labels shape:  (1000,)
Test data shape:  (1000, 3073)
Test labels shape:  (1000,)
dev data shape:  (500, 3073)
dev labels shape:  (500,)
```

Softmax Classifier

Your code for this section will all be written inside **cs682/classifiers/softmax.py**.

In [5]:
```
# First implement the naive softmax loss function with nested loops.
# Open the file cs682/classifiers/softmax.py and implement the
# softmax_loss_naive function.

from cs682.classifiers.softmax import softmax_loss_naive
import time

# Generate a random softmax weight matrix and use it to compute the loss.
W = np.random.randn(3073, 10) * 0.0001
loss, grad = softmax_loss_naive(W, X_dev, y_dev, 0.0)

# As a rough sanity check, our loss should be something close to -log(0.1).
print('loss: %f' % loss)
print('sanity check: %f' % (-np.log(0.1)))
```

```
loss: 2.378209
sanity check: 2.302585
```

Inline Question 1:

Why do we expect our loss to be close to -log(0.1)? Explain briefly.**

Your answer: With random weight initialization with values in the range 0.0001, the scores for each of the classes would be very small values in a similar range. The exponentiation of such a small value would be very close to 1 and hence the softmax term becomes -log(s/(10*s)) = -log(0.1).

In [6]:
```
# Complete the implementation of softmax_loss_naive and implement a (naive)
# version of the gradient that uses nested loops.
loss, grad = softmax_loss_naive(W, X_dev, y_dev, 0.0)

# As we did for the SVM, use numeric gradient checking as a debugging tool.
# The numeric gradient should be close to the analytic gradient.
from cs682.gradient_check import grad_check_sparse
f = lambda w: softmax_loss_naive(w, X_dev, y_dev, 0.0)[0]
grad_numerical = grad_check_sparse(f, W, grad, 10)

# similar to SVM case, do another gradient check with regularization
loss, grad = softmax_loss_naive(W, X_dev, y_dev, 5e1)
f = lambda w: softmax_loss_naive(w, X_dev, y_dev, 5e1)[0]
grad_numerical = grad_check_sparse(f, W, grad, 10)
```

```
numerical: -0.078319 analytic: -0.078319, relative error: 3.021400e-07
numerical: -2.730777 analytic: -2.730777, relative error: 2.790426e-09
numerical: -0.047185 analytic: -0.047185, relative error: 3.914673e-07
numerical: 0.671479 analytic: 0.671479, relative error: 1.064990e-07
numerical: -0.932518 analytic: -0.932518, relative error: 2.339729e-08
numerical: 0.437400 analytic: 0.437400, relative error: 6.794517e-08
numerical: -0.376330 analytic: -0.376330, relative error: 5.018106e-08
numerical: -0.503356 analytic: -0.503356, relative error: 3.716490e-08
numerical: 0.561375 analytic: 0.561375, relative error: 4.432825e-08
numerical: -0.673301 analytic: -0.673301, relative error: 4.080552e-08
numerical: 0.739985 analytic: 0.739985, relative error: 8.303052e-09
numerical: 0.999494 analytic: 0.999494, relative error: 3.116367e-08
numerical: -3.115712 analytic: -3.115712, relative error: 9.551139e-09
numerical: -0.214271 analytic: -0.214271, relative error: 1.673878e-07
numerical: 3.611589 analytic: 3.611589, relative error: 1.273046e-08
numerical: -0.400190 analytic: -0.400190, relative error: 3.922155e-08
numerical: 0.586929 analytic: 0.586929, relative error: 1.090298e-08
numerical: 1.714036 analytic: 1.714036, relative error: 1.378063e-08
numerical: -0.210448 analytic: -0.210448, relative error: 3.865159e-10
numerical: 2.001744 analytic: 2.001744, relative error: 2.666238e-09
```

In [7]:
```
# Now that we have a naive implementation of the softmax loss function and its gradient,
# implement a vectorized version in softmax_loss_vectorized.
# The two versions should compute the same results, but the vectorized version should be
# much faster.
tic = time.time()
loss_naive, grad_naive = softmax_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('naive loss: %e computed in %fs' % (loss_naive, toc - tic))

from cs682.classifiers.softmax import softmax_loss_vectorized
tic = time.time()
loss_vectorized, grad_vectorized = softmax_loss_vectorized(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('vectorized loss: %e computed in %fs' % (loss_vectorized, toc - tic))

# As we did for the SVM, we use the Frobenius norm to compare the two versions
# of the gradient.
grad_difference = np.linalg.norm(grad_naive - grad_vectorized, ord='fro')
print('Loss difference: %f' % np.abs(loss_naive - loss_vectorized))
print('Gradient difference: %f' % grad_difference)
```

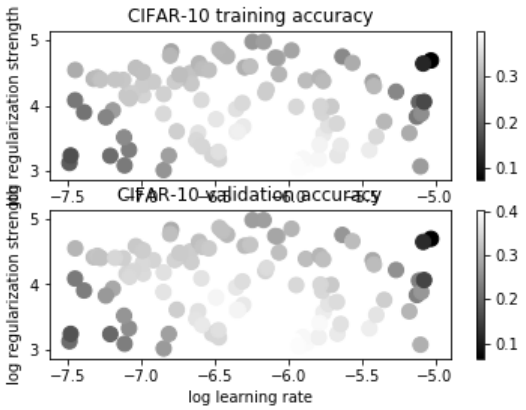
```
naive loss: 2.378209e+00 computed in 12.622023s
vectorized loss: 2.378209e+00 computed in 0.075435s
Loss difference: 0.000000
Gradient difference: 0.000000
```

In [16]:
```
# 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 over 0.35 on the validation set.
from cs682.classifiers import Softmax
results = {}
best_val = -1
best_softmax = None
# learning_rates = [3e-7, 4e-7, 5e-7, 6e-7, 7e-7]
# regularization_strengths = [1e3, 5e3, 1e4, 2e4, 3e4, 4e4]

################################################################################
# TODO:                                                                        #
# Use the validation set to set the learning rate and regularization strength. #
# This should be identical to the validation that you did for the SVM; save    #
# the best trained softmax classifer in best_softmax.                          #
################################################################################
np.random.seed(123)
N = X_train.shape[0]
D = X_train.shape[1]
C = W.shape[1]
num_trials = 100
for t in range(num_trials):
#   print ('learning rate: %f, regularization strength: %f' %(lr, rs))
    lr = 10 ** np.random.uniform (-7.5, -5)
    rs = 10 ** np.random.uniform (3,5)
    softmax = Softmax()
    loss = softmax.train(X_train, y_train, learning_rate=lr, reg=rs,
                  num_iters=1000, verbose=False)
    y_train_pred = softmax.predict (X_train)
    train_acc = np.mean(y_train_pred == y_train)

    y_val_pred = softmax.predict (X_val)
    val_acc = np.mean(y_val_pred == y_val)

    results[(lr, rs)] = (train_acc, val_acc)
    if val_acc > best_val:
        best_val = val_acc
        best_softmax = softmax

################################################################################
#                              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)
```

```
lr 3.258080e-08 reg 1.348217e+03 train accuracy: 0.191082 val accuracy: 0.209000
lr 3.321815e-08 reg 1.751938e+03 train accuracy: 0.174918 val accuracy: 0.176000
lr 3.559932e-08 reg 1.225499e+04 train accuracy: 0.226776 val accuracy: 0.227000
lr 3.569976e-08 reg 3.549747e+04 train accuracy: 0.292531 val accuracy: 0.306000
lr 4.099158e-08 reg 8.048387e+03 train accuracy: 0.220857 val accuracy: 0.231000
lr 4.766177e-08 reg 2.596421e+04 train accuracy: 0.307796 val accuracy: 0.320000
lr 5.050069e-08 reg 2.733333e+04 train accuracy: 0.313347 val accuracy: 0.328000
lr 5.314633e-08 reg 2.521249e+04 train accuracy: 0.321653 val accuracy: 0.322000
lr 5.759198e-08 reg 6.727783e+03 train accuracy: 0.242776 val accuracy: 0.268000
lr 6.176492e-08 reg 1.735001e+03 train accuracy: 0.212673 val accuracy: 0.204000
lr 6.427551e-08 reg 8.594780e+03 train accuracy: 0.273327 val accuracy: 0.288000
lr 6.663849e-08 reg 2.534973e+04 train accuracy: 0.324367 val accuracy: 0.350000
lr 7.575469e-08 reg 2.505885e+04 train accuracy: 0.326551 val accuracy: 0.340000
lr 7.664551e-08 reg 3.318633e+03 train accuracy: 0.251102 val accuracy: 0.263000
lr 7.726900e-08 reg 1.237432e+03 train accuracy: 0.229082 val accuracy: 0.233000
lr 7.968840e-08 reg 1.446854e+04 train accuracy: 0.328776 val accuracy: 0.346000
lr 8.284518e-08 reg 2.132810e+03 train accuracy: 0.234286 val accuracy: 0.253000
lr 8.346735e-08 reg 1.301628e+04 train accuracy: 0.328939 val accuracy: 0.355000
lr 8.716513e-08 reg 1.584042e+04 train accuracy: 0.335592 val accuracy: 0.350000
lr 9.214246e-08 reg 3.833329e+04 train accuracy: 0.316449 val accuracy: 0.332000
lr 1.018775e-07 reg 3.275782e+04 train accuracy: 0.318694 val accuracy: 0.337000
lr 1.026013e-07 reg 2.098975e+04 train accuracy: 0.334061 val accuracy: 0.353000
lr 1.031189e-07 reg 3.328637e+04 train accuracy: 0.321102 val accuracy: 0.338000
lr 1.033242e-07 reg 1.461397e+04 train accuracy: 0.343633 val accuracy: 0.356000
lr 1.212109e-07 reg 2.340142e+04 train accuracy: 0.332265 val accuracy: 0.347000
lr 1.419320e-07 reg 7.636723e+03 train accuracy: 0.349612 val accuracy: 0.367000
lr 1.425447e-07 reg 1.522328e+04 train accuracy: 0.339980 val accuracy: 0.363000
lr 1.425751e-07 reg 1.047021e+03 train accuracy: 0.262286 val accuracy: 0.269000
lr 1.546193e-07 reg 1.749254e+03 train accuracy: 0.276286 val accuracy: 0.278000
lr 1.576817e-07 reg 5.891604e+04 train accuracy: 0.307102 val accuracy: 0.319000
lr 1.602274e-07 reg 6.825114e+04 train accuracy: 0.283490 val accuracy: 0.294000
lr 1.694010e-07 reg 2.353230e+04 train accuracy: 0.334449 val accuracy: 0.345000
lr 1.749800e-07 reg 4.100961e+03 train accuracy: 0.336959 val accuracy: 0.338000
lr 1.795360e-07 reg 2.595024e+04 train accuracy: 0.320041 val accuracy: 0.336000
lr 1.827506e-07 reg 3.035897e+04 train accuracy: 0.316224 val accuracy: 0.334000
lr 2.168210e-07 reg 3.201254e+04 train accuracy: 0.322653 val accuracy: 0.337000
lr 2.200686e-07 reg 2.197883e+04 train accuracy: 0.328490 val accuracy: 0.338000
lr 2.375602e-07 reg 2.989391e+03 train accuracy: 0.348429 val accuracy: 0.368000
lr 2.446575e-07 reg 3.883925e+03 train accuracy: 0.361837 val accuracy: 0.363000
lr 2.480178e-07 reg 4.489681e+04 train accuracy: 0.304327 val accuracy: 0.318000
lr 2.573337e-07 reg 1.236507e+04 train accuracy: 0.351469 val accuracy: 0.366000
lr 2.597383e-07 reg 2.407501e+03 train accuracy: 0.345837 val accuracy: 0.374000
lr 2.615648e-07 reg 3.740476e+04 train accuracy: 0.314551 val accuracy: 0.326000
lr 2.889722e-07 reg 3.543106e+04 train accuracy: 0.318286 val accuracy: 0.335000
lr 3.042534e-07 reg 1.771001e+03 train accuracy: 0.343306 val accuracy: 0.351000
lr 3.238884e-07 reg 6.296680e+03 train accuracy: 0.366367 val accuracy: 0.390000
lr 3.357618e-07 reg 1.992543e+03 train accuracy: 0.358265 val accuracy: 0.362000
lr 3.385263e-07 reg 1.546869e+03 train accuracy: 0.342898 val accuracy: 0.350000
lr 3.415061e-07 reg 7.319371e+04 train accuracy: 0.303714 val accuracy: 0.320000
lr 3.593983e-07 reg 6.504733e+04 train accuracy: 0.302347 val accuracy: 0.320000
lr 3.849291e-07 reg 5.759985e+04 train accuracy: 0.301776 val accuracy: 0.309000
lr 4.433648e-07 reg 8.990631e+03 train accuracy: 0.360633 val accuracy: 0.372000
lr 4.479656e-07 reg 4.042973e+03 train accuracy: 0.376714 val accuracy: 0.396000
lr 4.569945e-07 reg 4.084633e+03 train accuracy: 0.379714 val accuracy: 0.389000
lr 4.892519e-07 reg 1.109545e+04 train accuracy: 0.349490 val accuracy: 0.358000
lr 4.951370e-07 reg 4.856733e+03 train accuracy: 0.372204 val accuracy: 0.374000
lr 5.071041e-07 reg 3.887843e+04 train accuracy: 0.298531 val accuracy: 0.313000
lr 5.523354e-07 reg 4.033332e+04 train accuracy: 0.317286 val accuracy: 0.326000
lr 5.731314e-07 reg 9.626114e+04 train accuracy: 0.265020 val accuracy: 0.274000
lr 6.821113e-07 reg 9.621291e+04 train accuracy: 0.276286 val accuracy: 0.275000
lr 7.004375e-07 reg 1.596999e+04 train accuracy: 0.334204 val accuracy: 0.347000
lr 7.146074e-07 reg 1.777744e+04 train accuracy: 0.335388 val accuracy: 0.357000
lr 8.174599e-07 reg 5.482741e+04 train accuracy: 0.279061 val accuracy: 0.285000
lr 8.490516e-07 reg 5.395744e+04 train accuracy: 0.290735 val accuracy: 0.298000
lr 8.684032e-07 reg 3.070606e+04 train accuracy: 0.304816 val accuracy: 0.318000
lr 1.056483e-06 reg 7.080378e+04 train accuracy: 0.283082 val accuracy: 0.304000
lr 1.110817e-06 reg 1.374484e+03 train accuracy: 0.398898 val accuracy: 0.404000
lr 1.136732e-06 reg 9.745731e+03 train accuracy: 0.348531 val accuracy: 0.352000
lr 1.188787e-06 reg 1.156345e+03 train accuracy: 0.394449 val accuracy: 0.399000
lr 1.263509e-06 reg 1.559978e+03 train accuracy: 0.389143 val accuracy: 0.394000
lr 1.320554e-06 reg 2.454567e+04 train accuracy: 0.313673 val accuracy: 0.316000
lr 1.341472e-06 reg 1.306028e+03 train accuracy: 0.394408 val accuracy: 0.397000
lr 1.615486e-06 reg 1.558226e+03 train accuracy: 0.388082 val accuracy: 0.383000
lr 1.641789e-06 reg 2.585399e+03 train accuracy: 0.383122 val accuracy: 0.396000
lr 1.644336e-06 reg 2.738173e+04 train accuracy: 0.303143 val accuracy: 0.307000
lr 1.662881e-06 reg 9.937517e+03 train accuracy: 0.344939 val accuracy: 0.350000
```

```
lr 1.703983e-06 reg 7.937453e+03 train accuracy: 0.345714 val accuracy: 0.352000
lr 1.734487e-06 reg 2.412066e+04 train accuracy: 0.311224 val accuracy: 0.319000
lr 1.742501e-06 reg 3.734897e+03 train accuracy: 0.372000 val accuracy: 0.379000
lr 1.878426e-06 reg 5.054512e+03 train accuracy: 0.360490 val accuracy: 0.367000
lr 1.966854e-06 reg 9.815506e+03 train accuracy: 0.331388 val accuracy: 0.332000
lr 2.149954e-06 reg 5.393259e+03 train accuracy: 0.347959 val accuracy: 0.357000
lr 2.167211e-06 reg 1.760490e+03 train accuracy: 0.366408 val accuracy: 0.375000
lr 2.280421e-06 reg 2.482905e+03 train accuracy: 0.370122 val accuracy: 0.363000
lr 2.340774e-06 reg 5.699793e+04 train accuracy: 0.248490 val accuracy: 0.264000
lr 2.740301e-06 reg 4.567705e+04 train accuracy: 0.293204 val accuracy: 0.315000
lr 3.574047e-06 reg 2.125066e+03 train accuracy: 0.374041 val accuracy: 0.381000
lr 3.762056e-06 reg 2.342654e+04 train accuracy: 0.278735 val accuracy: 0.289000
lr 3.839631e-06 reg 2.013811e+04 train accuracy: 0.307980 val accuracy: 0.314000
lr 3.983799e-06 reg 3.419115e+03 train accuracy: 0.349612 val accuracy: 0.368000
lr 4.508858e-06 reg 5.783580e+03 train accuracy: 0.306776 val accuracy: 0.338000
lr 5.427528e-06 reg 1.660644e+04 train accuracy: 0.245510 val accuracy: 0.256000
lr 6.683194e-06 reg 3.864594e+03 train accuracy: 0.292245 val accuracy: 0.306000
lr 7.515159e-06 reg 6.861880e+03 train accuracy: 0.228551 val accuracy: 0.238000
lr 7.776557e-06 reg 1.141131e+04 train accuracy: 0.201347 val accuracy: 0.235000
lr 7.954041e-06 reg 1.195963e+03 train accuracy: 0.272633 val accuracy: 0.299000
lr 8.060313e-06 reg 7.657722e+03 train accuracy: 0.254306 val accuracy: 0.266000
lr 8.301652e-06 reg 4.454291e+04 train accuracy: 0.122449 val accuracy: 0.122000
lr 8.424696e-06 reg 1.163300e+04 train accuracy: 0.167184 val accuracy: 0.156000
lr 9.378184e-06 reg 4.972532e+04 train accuracy: 0.073776 val accuracy: 0.065000
best validation accuracy achieved during cross-validation: 0.404000
```

In [17]:
```
# 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()
```



In [18]:
```
# evaluate on test set
# Evaluate the best softmax on test set
y_test_pred = best_softmax.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('softmax on raw pixels final test set accuracy: %f' % (test_accuracy, ))
```

```
softmax on raw pixels final test set accuracy: 0.376000
```

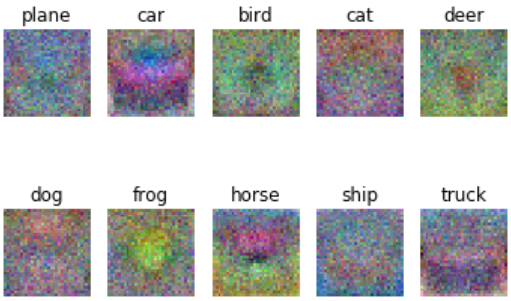
Inline Question - *True or False*

It's possible to add a new datapoint to a training set that would leave the SVM loss unchanged, but this is not the case with the Softmax classifier loss.

Your answer: True

Your explanation: Any datapoint which has a score beyond the margin with respect to the correct class score will contribute a 0 towards the loss. Adding such a datapoint would leave the SVM loss unchanged. However, in the case of softmax, this datapoint would have a non-zero contribution towards the loss.

In [19]:
```
# Visualize the learned weights for each class
w = best_softmax.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])
```



In [ ]:

Implementing a Neural Network

In this exercise we will develop a neural network with fully-connected layers to perform classification, and test it out on the CIFAR-10 dataset.

In [1]:
```
# A bit of setup

from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt

from cs682.classifiers.neural_net import TwoLayerNet


%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'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2

def rel_error(x, y):
    """ returns relative error """
    return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

We will use the class `TwoLayerNet` in the file `cs682/classifiers/neural_net.py` to represent instances of our network. The network parameters are stored in the instance variable `self.params` where keys are string parameter names and values are numpy arrays. Below, we initialize toy data and a toy model that we will use to develop your implementation.

In [2]:
```
# Create a small net and some toy data to check your implementations.
# Note that we set the random seed for repeatable experiments.

input_size = 4
hidden_size = 10
num_classes = 3
num_inputs = 5

def init_toy_model():
    np.random.seed(0)
    return TwoLayerNet(input_size, hidden_size, num_classes, std=1e-1)

def init_toy_data():
    np.random.seed(1)
    X = 10 * np.random.randn(num_inputs, input_size)
    y = np.array([0, 1, 2, 2, 1])
    return X, y

net = init_toy_model()
X, y = init_toy_data()
```

Forward pass: compute scores

Open the file `cs682/classifiers/neural_net.py` and look at the method `TwoLayerNet.loss`. This function is very similar to the loss functions you have written for the SVM and Softmax exercises: It takes the data and weights and computes the class scores, the loss, and the gradients on the parameters.

Implement the first part of the forward pass which uses the weights and biases to compute the scores for all inputs.

In [3]:
```
scores = net.loss(X)
print('Your scores:')
print(scores)
print()
print('correct scores:')
correct_scores = np.asarray([
  [-0.81233741, -1.27654624, -0.70335995],
  [-0.17129677, -1.18803311, -0.47310444],
  [-0.51590475, -1.01354314, -0.8504215 ],
  [-0.15419291, -0.48629638, -0.52901952],
  [-0.00618733, -0.12435261, -0.15226949]])
print(correct_scores)
print()

# The difference should be very small. We get < 1e-7
print('Difference between your scores and correct scores:')
print(np.sum(np.abs(scores - correct_scores)))
```

```
Your scores:
[[-0.81233741 -1.27654624 -0.70335995]
 [-0.17129677 -1.18803311 -0.47310444]
 [-0.51590475 -1.01354314 -0.8504215 ]
 [-0.15419291 -0.48629638 -0.52901952]
 [-0.00618733 -0.12435261 -0.15226949]]

correct scores:
[[-0.81233741 -1.27654624 -0.70335995]
 [-0.17129677 -1.18803311 -0.47310444]
 [-0.51590475 -1.01354314 -0.8504215 ]
 [-0.15419291 -0.48629638 -0.52901952]
 [-0.00618733 -0.12435261 -0.15226949]]

Difference between your scores and correct scores:
3.6802720496109664e-08
```

Forward pass: compute loss

In the same function, implement the second part that computes the data and regularizaion loss.

In [4]:
```
loss, _ = net.loss(X, y, reg=0.05)
correct_loss = 1.30378789133

# should be very small, we get < 1e-12
print('Difference between your loss and correct loss:')
print(np.sum(np.abs(loss - correct_loss)))
```

```
Difference between your loss and correct loss:
1.7985612998927536e-13
```

Backward pass

Implement the rest of the function. This will compute the gradient of the loss with respect to the variables `W1`, `b1`, `W2`, and `b2`. Now that you (hopefully!) have a correctly implemented forward pass, you can debug your backward pass using a numeric gradient check:

In [5]:
```
from cs682.gradient_check import eval_numerical_gradient

# Use numeric gradient checking to check your implementation of the backward pass.
# If your implementation is correct, the difference between the numeric and
# analytic gradients should be less than 1e-8 for each of W1, W2, b1, and b2.

loss, grads = net.loss(X, y, reg=0.05)

# these should all be less than 1e-8 or so
for param_name in grads:
    f = lambda W: net.loss(X, y, reg=0.05)[0]
    param_grad_num = eval_numerical_gradient(f, net.params[param_name], verbose=False)
#     print (grads[param_name], "____")
#     print (param_grad_num)
    print('%s max relative error: %e' % (param_name, rel_error(param_grad_num, grads[param_name])))
```

```
b2 max relative error: 3.865091e-11
W2 max relative error: 3.440708e-09
b1 max relative error: 1.555471e-09
W1 max relative error: 3.561318e-09
```

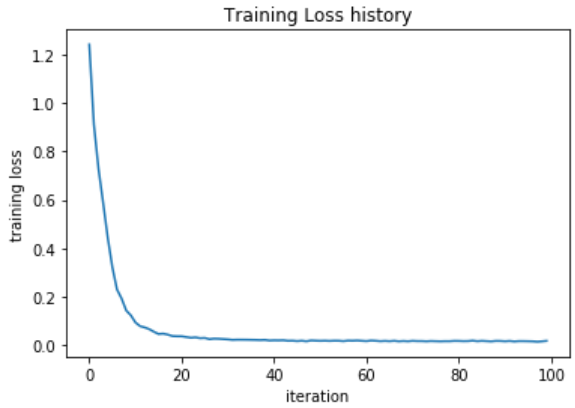
Train the network

To train the network we will use stochastic gradient descent (SGD), similar to the SVM and Softmax classifiers. Look at the function `TwoLayerNet.train` and fill in the missing sections to implement the training procedure. This should be very similar to the training procedure you used for the SVM and Softmax classifiers. You will also have to implement `TwoLayerNet.predict`, as the training process periodically performs prediction to keep track of accuracy over time while the network trains.

Once you have implemented the method, run the code below to train a two-layer network on toy data. You should achieve a training loss less than 0.2.

In [7]:
```
net = init_toy_model()
stats = net.train(X, y, X, y,
            learning_rate=1e-1, reg=5e-6,
            num_iters=100, verbose=False)

print('Final training loss: ', stats['loss_history'][-1])

# plot the loss history
plt.plot(stats['loss_history'])
plt.xlabel('iteration')
plt.ylabel('training loss')
plt.title('Training Loss history')
plt.show()
```

```
Final training loss:  0.017149607938732093
```



Load the data

Now that you have implemented a two-layer network that passes gradient checks and works on toy data, it's time to load up our favorite CIFAR-10 data so we can use it to train a classifier on a real dataset.

In [8]:
```
from cs682.data_utils import load_CIFAR10

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 two-layer neural net 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 = 'cs682/datasets/cifar-10-batches-py'

    X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

    # Subsample the data
    mask = list(range(num_training, num_training + num_validation))
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = list(range(num_training))
    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]

    # 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

    # Reshape data to rows
    X_train = X_train.reshape(num_training, -1)
    X_val = X_val.reshape(num_validation, -1)
    X_test = X_test.reshape(num_test, -1)

    return X_train, y_train, X_val, y_val, X_test, y_test


# 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

# Invoke the above function to get our data.
X_train, y_train, X_val, y_val, X_test, y_test = get_CIFAR10_data()
print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

```
Train data shape:  (49000, 3072)
Train labels shape:  (49000,)
Validation data shape:  (1000, 3072)
Validation labels shape:  (1000,)
Test data shape:  (1000, 3072)
Test labels shape:  (1000,)
```

Train a network

To train our network we will use SGD. In addition, we will adjust the learning rate with an exponential learning rate schedule as optimization proceeds; after each epoch, we will reduce the learning rate by multiplying it by a decay rate.

In [9]:
```
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
net = TwoLayerNet(input_size, hidden_size, num_classes)

# Train the network
stats = net.train(X_train, y_train, X_val, y_val,
            num_iters=1000, batch_size=200,
            learning_rate=1e-4, learning_rate_decay=0.95,
            reg=0.25, verbose=True)

# Predict on the validation set
val_acc = (net.predict(X_val) == y_val).mean()
print('Validation accuracy: ', val_acc)
```

```
iteration 0 / 1000: loss 2.302954
iteration 100 / 1000: loss 2.302550
iteration 200 / 1000: loss 2.297648
iteration 300 / 1000: loss 2.259602
iteration 400 / 1000: loss 2.204170
iteration 500 / 1000: loss 2.118565
iteration 600 / 1000: loss 2.051535
iteration 700 / 1000: loss 1.988466
iteration 800 / 1000: loss 2.006591
iteration 900 / 1000: loss 1.951473
Validation accuracy:  0.287
```

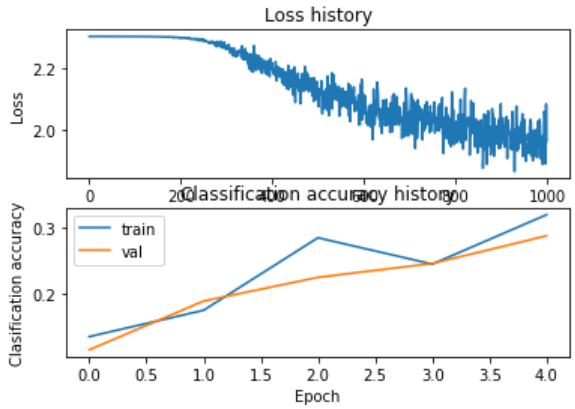
Debug the training

With the default parameters we provided above, you should get a validation accuracy of about 0.29 on the validation set. This isn't very good.

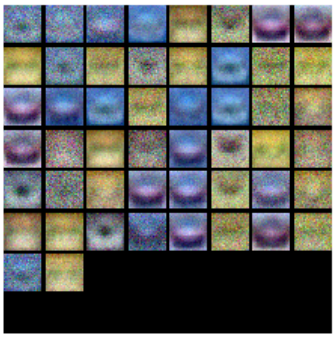
One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

Another strategy is to visualize the weights that were learned in the first layer of the network. In most neural networks trained on visual data, the first layer weights typically show some visible structure when visualized.

In [10]:
```
# Plot the loss function and train / validation accuracies
plt.subplot(2, 1, 1)
plt.plot(stats['loss_history'])
plt.title('Loss history')
plt.xlabel('Iteration')
plt.ylabel('Loss')

plt.subplot(2, 1, 2)
plt.plot(stats['train_acc_history'], label='train')
plt.plot(stats['val_acc_history'], label='val')
plt.title('Classification accuracy history')
plt.xlabel('Epoch')
plt.ylabel('Clasification accuracy')
plt.legend()
plt.show()
```



In [11]:
```
from cs682.vis_utils import visualize_grid

# Visualize the weights of the network

def show_net_weights(net):
    W1 = net.params['W1']
    W1 = W1.reshape(32, 32, 3, -1).transpose(3, 0, 1, 2)
    plt.imshow(visualize_grid(W1, padding=3).astype('uint8'))
    plt.gca().axis('off')
    plt.show()

show_net_weights(net)
```



Tune your hyperparameters

What's wrong?. Looking at the visualizations above, we see that the loss is decreasing more or less linearly, which seems to suggest that the learning rate may be too low. Moreover, there is no gap between the training and validation accuracy, suggesting that the model we used has low capacity, and that we should increase its size. On the other hand, with a very large model we would expect to see more overfitting, which would manifest itself as a very large gap between the training and validation accuracy.

Tuning. Tuning the hyperparameters and developing intuition for how they affect the final performance is a large part of using Neural Networks, so we want you to get a lot of practice. Below, you should experiment with different values of the various hyperparameters, including hidden layer size, learning rate, numer of training epochs, and regularization strength. You might also consider tuning the learning rate decay, but you should be able to get good performance using the default value.

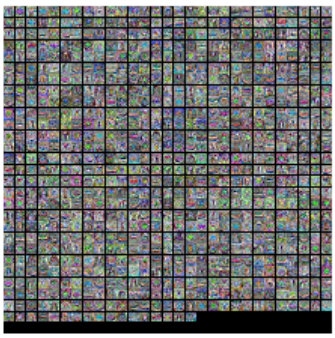
Approximate results. You should be aim to achieve a classification accuracy of greater than 48% on the validation set. Our best network gets over 52% on the validation set.

Experiment: You goal in this exercise is to get as good of a result on CIFAR-10 as you can, with a fully-connected Neural Network. Feel free implement your own techniques (e.g. PCA to reduce dimensionality, or adding dropout, or adding features to the solver, etc.).

In [41]:
```
best_net = None # store the best model into this

#################################################################################
# TODO: Tune hyperparameters using the validation set. Store your best trained  #
# model in best_net.                                                            #
#                                                                               #
# To help debug your network, it may help to use visualizations similar to the  #
# ones we used above; these visualizations will have significant qualitative    #
# differences from the ones we saw above for the poorly tuned network.          #
#                                                                               #
# Tweaking hyperparameters by hand can be fun, but you might find it useful to  #
# write code to sweep through possible combinations of hyperparameters          #
# automatically like we did on the previous exercises.                          #
#################################################################################
input_size = 32 * 32 * 3
num_classes = 10

hidden_sizes = [800] # Also tried 100, 200, 400
# learning_rates = [1e-3, 2e-3]
# regularizations = [0.05, 0.25, 1.25]
batch_sizes = [800, 1600] # Also tried 100, 200, 400
best_val_acc = 0
best_hyperparams = None

np.random.seed(123)
for hs in hidden_sizes:
    for bs in batch_sizes:
        num_trials = 5
        for t in range(num_trials):
            # Train the network
            lr = 10 ** np.random.uniform (-3,-2)
            rs = 10 ** np.random.uniform (-2,-0.5)
            net = TwoLayerNet(input_size, hs, num_classes)
            stats = net.train(X_train, y_train, X_val, y_val,
                        num_iters=2000, batch_size=bs,
                        learning_rate=lr, learning_rate_decay=0.95,
                        reg=rs, verbose=False)

            # Predict on the train set
            train_acc = (net.predict(X_train) == y_train).mean()

            # Predict on the validation set
            val_acc = (net.predict(X_val) == y_val).mean()
            print (val_acc, train_acc, hs, bs, lr, rs)
            if best_val_acc < val_acc:
                best_val_acc = val_acc
                best_net = net
                best_hyper_params = (val_acc, train_acc, hs, bs, lr, rs)
print('Best validation accuracy: ', best_val_acc)
print('Best hyper params: ', best_hyper_params)
#################################################################################
#                               END OF YOUR CODE                                #
#################################################################################
```

```
0.087 0.10026530612244898 800 800 0.00497129099780719 0.02686637067590742
0.506 0.6154693877551021 800 800 0.00280889906844216 0.042636993885010466
0.087 0.10026530612244898 800 800 0.004740475286260089 0.01184755980686724
0.146 0.1550816326530612 800 800 0.004027365789262474 0.11474717519396312
0.534 0.6797142857142857 800 800 0.0023958827548970106 0.02362134176154587
0.518 0.6816734693877551 800 1600 0.0020229985980093684 0.015419954656987641
0.537 0.6313265306122449 800 1600 0.0010316890375639816 0.13838674191166497
0.087 0.10026530612244898 800 1600 0.007435997870359743 0.026299932822681553
0.087 0.10026530612244898 800 1600 0.005910761552109519 0.10312470319012793
0.512 0.5998979591836735 800 1600 0.0033763505833719573 0.07230429643837918
Best validation accuracy:  0.537
Best hyper params:  (0.537, 0.6313265306122449, 800, 1600, 0.0010316890375639816, 0.13838674191166497)
```

In [42]:
```
# visualize the weights of the best network
show_net_weights(best_net)
```



Run on the test set

When you are done experimenting, you should evaluate your final trained network on the test set; you should get above 48%.

In [43]:
```
test_acc = (best_net.predict(X_test) == y_test).mean()
print('Test accuracy: ', test_acc)
```

```
Test accuracy:  0.547
```

Inline Question

Now that you have trained a Neural Network classifier, you may find that your testing accuracy is much lower than the training accuracy. In what ways can we decrease this gap? Select all that apply.

1. Train on a larger dataset.
2. Add more hidden units.
3. Increase the regularization strength.
4. None of the above.

Your answer: 1. and 3.

Your explanation:

Testing accuracy being much lower than training accuracy implies overfitting.

Training on a larger dataset would allow the model to look at more data and would therefore be able to understand the distribution better and generalize.

Adding more hidden units would increase the number of parameters to train for and would result is more overfitting. Therefore, this is not a good option to decrease the gap.

Increasing the regularization strength would also reduce overfitting by penalizing larger values of W. Reducing the range that W can take reduces its ability to stretch in dimensions to fit the data better, thereby reducing overfitting.

In [ ]:
```
```

Image features 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 (https://compsci682-fa19.github.io/assignments2019/assignment1) on the course website.

We have seen that we can achieve reasonable performance on an image classification task by training a linear classifier on the pixels of the input image. In this exercise we will show that we can improve our classification performance by training linear classifiers not on raw pixels but on features that are computed from the raw pixels.

All of your work for this exercise will be done in this notebook.

In [1]:
```
from __future__ import print_function
import random
import numpy as np
from cs682.data_utils import load_CIFAR10
import matplotlib.pyplot as plt


%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'

# for auto-reloading extenrnal modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

Load data

Similar to previous exercises, we will load CIFAR-10 data from disk.

In [2]:
```
from cs682.features import color_histogram_hsv, hog_feature

def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000):
    # Load the raw CIFAR-10 data
    cifar10_dir = 'cs682/datasets/cifar-10-batches-py'

    X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

    # Subsample the data
    mask = list(range(num_training, num_training + num_validation))
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = list(range(num_training))
    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]

    return X_train, y_train, X_val, y_val, X_test, y_test

# 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_val, y_val, X_test, y_test = get_CIFAR10_data()
```

Extract Features

For each image we will compute a Histogram of Oriented Gradients (HOG) as well as a color histogram using the hue channel in HSV color space. We form our final feature vector for each image by concatenating the HOG and color histogram feature vectors.

Roughly speaking, HOG should capture the texture of the image while ignoring color information, and the color histogram represents the color of the input image while ignoring texture. As a result, we expect that using both together ought to work better than using either alone. Verifying this assumption would be a good thing to try for your interests.

The `hog_feature` and `color_histogram_hsv` functions both operate on a single image and return a feature vector for that image. The extract_features function takes a set of images and a list of feature functions and evaluates each feature function on each image, storing the results in a matrix where each column is the concatenation of all feature vectors for a single image.

In [48]:
```
from cs682.features import *

num_color_bins = 25 # Number of bins in the color histogram
feature_fns = [hog_feature, lambda img: color_histogram_hsv(img, nbin=num_color_bins)]
X_train_feats = extract_features(X_train, feature_fns, verbose=True)
X_val_feats = extract_features(X_val, feature_fns)
X_test_feats = extract_features(X_test, feature_fns)

# Preprocessing: Subtract the mean feature
mean_feat = np.mean(X_train_feats, axis=0, keepdims=True)
X_train_feats -= mean_feat
X_val_feats -= mean_feat
X_test_feats -= mean_feat

# Preprocessing: Divide by standard deviation. This ensures that each feature
# has roughly the same scale.
std_feat = np.std(X_train_feats, axis=0, keepdims=True)
X_train_feats /= std_feat
X_val_feats /= std_feat
X_test_feats /= std_feat

# Preprocessing: Add a bias dimension
X_train_feats = np.hstack([X_train_feats, np.ones((X_train_feats.shape[0], 1))])
X_val_feats = np.hstack([X_val_feats, np.ones((X_val_feats.shape[0], 1))])
X_test_feats = np.hstack([X_test_feats, np.ones((X_test_feats.shape[0], 1))])
```

```
Done extracting features for 1000 / 49000 images
Done extracting features for 2000 / 49000 images
Done extracting features for 3000 / 49000 images
Done extracting features for 4000 / 49000 images
Done extracting features for 5000 / 49000 images
Done extracting features for 6000 / 49000 images
Done extracting features for 7000 / 49000 images
Done extracting features for 8000 / 49000 images
Done extracting features for 9000 / 49000 images
Done extracting features for 10000 / 49000 images
Done extracting features for 11000 / 49000 images
Done extracting features for 12000 / 49000 images
Done extracting features for 13000 / 49000 images
Done extracting features for 14000 / 49000 images
Done extracting features for 15000 / 49000 images
Done extracting features for 16000 / 49000 images
Done extracting features for 17000 / 49000 images
Done extracting features for 18000 / 49000 images
Done extracting features for 19000 / 49000 images
Done extracting features for 20000 / 49000 images
Done extracting features for 21000 / 49000 images
Done extracting features for 22000 / 49000 images
Done extracting features for 23000 / 49000 images
Done extracting features for 24000 / 49000 images
Done extracting features for 25000 / 49000 images
Done extracting features for 26000 / 49000 images
Done extracting features for 27000 / 49000 images
Done extracting features for 28000 / 49000 images
Done extracting features for 29000 / 49000 images
Done extracting features for 30000 / 49000 images
Done extracting features for 31000 / 49000 images
Done extracting features for 32000 / 49000 images
Done extracting features for 33000 / 49000 images
Done extracting features for 34000 / 49000 images
Done extracting features for 35000 / 49000 images
Done extracting features for 36000 / 49000 images
Done extracting features for 37000 / 49000 images
Done extracting features for 38000 / 49000 images
Done extracting features for 39000 / 49000 images
Done extracting features for 40000 / 49000 images
Done extracting features for 41000 / 49000 images
Done extracting features for 42000 / 49000 images
Done extracting features for 43000 / 49000 images
Done extracting features for 44000 / 49000 images
Done extracting features for 45000 / 49000 images
Done extracting features for 46000 / 49000 images
Done extracting features for 47000 / 49000 images
Done extracting features for 48000 / 49000 images
```

Train SVM on features

Using the multiclass SVM code developed earlier in the assignment, train SVMs on top of the features extracted above; this should achieve better results than training SVMs directly on top of raw pixels.

In [84]:

```
# Use the validation set to tune the learning rate and regularization strength

from cs682.classifiers.linear_classifier import LinearSVM

# learning_rates = [1e-9, 1e-8, 1e-7, 1e-6, 1e-5]
# regularization_strengths = [1e3, 1e4, 5e4, 5e5, 5e6]
learning_rates = [5e-7, 1e-6, 2.5e-6, 5e-6, 7.5e-6, 1e-5, 5e-5, 1e-4, 5e-4]
regularization_strengths = [1e2, 5e2, 1e3, 5e3, 1e4, 5e4]

results = {}
best_val = -1
best_svm = None

################################################################################
# TODO:                                                                        #
# Use the validation set to set the learning rate and regularization strength. #
# This should be identical to the validation that you did for the SVM; save    #
# the best trained classifer in best_svm. You might also want to play          #
# with different numbers of bins in the color histogram. If you are careful    #
# you should be able to get accuracy of near 0.44 on the validation set.       #
################################################################################
np.random.seed(123)
num_trials = 200
for t in range(num_trials):
#   print ('learning rate: %f, regularization strength: %f' %(lr, rs))
    lr = 10 ** np.random.uniform (-7.5, -4)
    rs = 10 ** np.random.uniform (1,5)
    svm = LinearSVM()
    loss_hist = svm.train(X_train_feats, y_train, learning_rate=lr, reg=rs,
                  num_iters=1000, batch_size=400, verbose=False)
    y_train_pred = svm.predict(X_train_feats)
    train_acc = np.mean(y_train == y_train_pred)
#   print('training accuracy: %f' % (train_acc))
    y_val_pred = svm.predict(X_val_feats)
    val_acc = np.mean(y_val == y_val_pred)
#   print('validation accuracy: %f' % (val_acc))
    results [(lr, rs)] = (train_acc, val_acc)
    if (val_acc > best_val):
        best_val = val_acc
        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)
```

```
lr 3.215477e-08 reg 5.523815e+02 train accuracy: 0.097286 val accuracy: 0.099000
lr 3.399504e-08 reg 1.592791e+04 train accuracy: 0.128286 val accuracy: 0.130000
lr 3.552800e-08 reg 2.347738e+01 train accuracy: 0.114633 val accuracy: 0.097000
lr 3.708583e-08 reg 7.259182e+01 train accuracy: 0.104286 val accuracy: 0.128000
lr 3.731825e-08 reg 1.216398e+01 train accuracy: 0.106388 val accuracy: 0.101000
lr 3.783558e-08 reg 1.684695e+03 train accuracy: 0.107429 val accuracy: 0.095000
lr 4.687757e-08 reg 4.343593e+01 train accuracy: 0.104122 val accuracy: 0.100000
lr 4.959981e-08 reg 1.646587e+02 train accuracy: 0.122776 val accuracy: 0.140000
lr 5.361862e-08 reg 2.310790e+03 train accuracy: 0.115020 val accuracy: 0.126000
lr 5.424113e-08 reg 1.926131e+02 train accuracy: 0.107918 val accuracy: 0.099000
lr 5.579342e-08 reg 6.632885e+03 train accuracy: 0.104265 val accuracy: 0.104000
lr 5.931982e-08 reg 1.682655e+03 train accuracy: 0.096510 val accuracy: 0.093000
lr 6.057849e-08 reg 1.225719e+02 train accuracy: 0.110592 val accuracy: 0.107000
lr 6.099074e-08 reg 1.313430e+04 train accuracy: 0.112918 val accuracy: 0.108000
lr 6.138025e-08 reg 6.599516e+01 train accuracy: 0.099776 val accuracy: 0.103000
lr 6.241379e-08 reg 3.684925e+03 train accuracy: 0.097796 val accuracy: 0.085000
lr 6.679378e-08 reg 9.758148e+03 train accuracy: 0.100490 val accuracy: 0.092000
lr 7.336283e-08 reg 1.650692e+01 train accuracy: 0.104469 val accuracy: 0.108000
lr 7.753666e-08 reg 3.373391e+03 train accuracy: 0.127143 val accuracy: 0.125000
lr 8.371576e-08 reg 7.639510e+02 train accuracy: 0.125000 val accuracy: 0.126000
lr 9.673969e-08 reg 4.534152e+01 train accuracy: 0.101184 val accuracy: 0.112000
lr 1.019249e-07 reg 7.395322e+03 train accuracy: 0.111327 val accuracy: 0.104000
lr 1.047759e-07 reg 1.489352e+01 train accuracy: 0.096633 val accuracy: 0.105000
lr 1.062432e-07 reg 2.380446e+03 train accuracy: 0.111204 val accuracy: 0.118000
lr 1.085271e-07 reg 8.482845e+01 train accuracy: 0.113367 val accuracy: 0.121000
lr 1.120190e-07 reg 2.836335e+02 train accuracy: 0.126347 val accuracy: 0.129000
lr 1.121666e-07 reg 2.854463e+04 train accuracy: 0.414490 val accuracy: 0.407000
lr 1.160666e-07 reg 1.330320e+03 train accuracy: 0.118429 val accuracy: 0.115000
lr 1.206724e-07 reg 1.382925e+03 train accuracy: 0.103082 val accuracy: 0.109000
lr 1.210385e-07 reg 1.684383e+04 train accuracy: 0.281041 val accuracy: 0.271000
lr 1.215481e-07 reg 4.409044e+02 train accuracy: 0.127367 val accuracy: 0.125000
lr 1.417455e-07 reg 1.992695e+02 train accuracy: 0.114224 val accuracy: 0.102000
lr 1.561872e-07 reg 5.839544e+02 train accuracy: 0.114571 val accuracy: 0.121000
lr 1.563581e-07 reg 1.216483e+04 train accuracy: 0.282837 val accuracy: 0.296000
lr 1.610030e-07 reg 4.254853e+03 train accuracy: 0.130347 val accuracy: 0.121000
lr 1.668774e-07 reg 2.391839e+04 train accuracy: 0.422510 val accuracy: 0.427000
lr 1.867786e-07 reg 2.206809e+04 train accuracy: 0.422102 val accuracy: 0.418000
lr 1.884455e-07 reg 6.781663e+04 train accuracy: 0.420714 val accuracy: 0.427000
lr 1.997262e-07 reg 6.179427e+04 train accuracy: 0.422694 val accuracy: 0.431000
lr 2.060173e-07 reg 2.313650e+01 train accuracy: 0.151531 val accuracy: 0.144000
lr 2.139225e-07 reg 2.623846e+01 train accuracy: 0.138959 val accuracy: 0.151000
lr 2.152326e-07 reg 6.915241e+03 train accuracy: 0.256061 val accuracy: 0.287000
lr 2.185422e-07 reg 3.195334e+03 train accuracy: 0.161388 val accuracy: 0.154000
lr 2.274074e-07 reg 1.604818e+04 train accuracy: 0.419102 val accuracy: 0.416000
lr 2.361381e-07 reg 6.145687e+02 train accuracy: 0.122286 val accuracy: 0.117000
lr 2.400752e-07 reg 2.328278e+03 train accuracy: 0.173898 val accuracy: 0.152000
lr 2.495423e-07 reg 3.957439e+03 train accuracy: 0.198776 val accuracy: 0.183000
lr 2.609098e-07 reg 1.992900e+03 train accuracy: 0.140816 val accuracy: 0.140000
lr 2.646037e-07 reg 3.780309e+03 train accuracy: 0.216980 val accuracy: 0.236000
lr 2.746492e-07 reg 1.043268e+03 train accuracy: 0.156673 val accuracy: 0.173000
lr 2.764092e-07 reg 1.453555e+01 train accuracy: 0.132939 val accuracy: 0.114000
lr 3.050717e-07 reg 2.202507e+02 train accuracy: 0.130653 val accuracy: 0.137000
lr 3.063412e-07 reg 5.771449e+02 train accuracy: 0.169000 val accuracy: 0.163000
lr 3.127928e-07 reg 1.419143e+02 train accuracy: 0.127510 val accuracy: 0.126000
lr 3.165317e-07 reg 7.538234e+01 train accuracy: 0.128796 val accuracy: 0.125000
lr 3.347130e-07 reg 4.111003e+02 train accuracy: 0.162041 val accuracy: 0.181000
lr 3.452989e-07 reg 4.577037e+02 train accuracy: 0.144776 val accuracy: 0.132000
lr 3.572813e-07 reg 4.482377e+03 train accuracy: 0.324714 val accuracy: 0.321000
lr 3.663415e-07 reg 6.621154e+02 train accuracy: 0.146143 val accuracy: 0.143000
lr 3.738338e-07 reg 1.877676e+01 train accuracy: 0.169327 val accuracy: 0.162000
lr 3.828175e-07 reg 4.523546e+04 train accuracy: 0.421306 val accuracy: 0.413000
lr 3.892380e-07 reg 1.149872e+01 train accuracy: 0.147918 val accuracy: 0.141000
lr 3.924042e-07 reg 7.626411e+01 train accuracy: 0.144184 val accuracy: 0.123000
lr 3.939285e-07 reg 1.341938e+01 train accuracy: 0.152184 val accuracy: 0.149000
lr 3.957572e-07 reg 3.208169e+02 train accuracy: 0.133204 val accuracy: 0.140000
lr 4.262258e-07 reg 1.895436e+01 train accuracy: 0.162020 val accuracy: 0.191000
lr 4.584292e-07 reg 1.690614e+02 train accuracy: 0.143592 val accuracy: 0.150000
lr 4.662920e-07 reg 1.537846e+02 train accuracy: 0.144653 val accuracy: 0.145000
lr 4.701563e-07 reg 2.988316e+04 train accuracy: 0.422163 val accuracy: 0.417000
lr 5.187767e-07 reg 2.934000e+03 train accuracy: 0.353388 val accuracy: 0.345000
lr 5.361621e-07 reg 2.697898e+02 train accuracy: 0.174490 val accuracy: 0.186000
lr 5.457122e-07 reg 1.346060e+03 train accuracy: 0.234735 val accuracy: 0.238000
lr 5.606072e-07 reg 2.864756e+02 train accuracy: 0.189592 val accuracy: 0.174000
lr 5.621451e-07 reg 1.372526e+02 train accuracy: 0.153204 val accuracy: 0.171000
lr 5.929073e-07 reg 3.050488e+04 train accuracy: 0.421408 val accuracy: 0.421000
lr 6.069741e-07 reg 1.744836e+03 train accuracy: 0.282184 val accuracy: 0.289000
```

```
lr 6.128538e-07 reg 7.147501e+02 train accuracy: 0.213490 val accuracy: 0.217000
lr 6.210454e-07 reg 4.739310e+01 train accuracy: 0.184347 val accuracy: 0.209000
lr 6.634091e-07 reg 1.322160e+03 train accuracy: 0.265020 val accuracy: 0.259000
lr 6.773313e-07 reg 2.084070e+02 train accuracy: 0.203429 val accuracy: 0.180000
lr 6.891545e-07 reg 1.479571e+03 train accuracy: 0.305122 val accuracy: 0.307000
lr 6.925374e-07 reg 2.097842e+01 train accuracy: 0.159061 val accuracy: 0.163000
lr 7.160274e-07 reg 1.915747e+01 train accuracy: 0.185939 val accuracy: 0.185000
lr 7.168276e-07 reg 1.134728e+01 train accuracy: 0.164265 val accuracy: 0.166000
lr 7.211071e-07 reg 1.591265e+04 train accuracy: 0.419184 val accuracy: 0.405000
lr 7.279735e-07 reg 2.221416e+01 train accuracy: 0.163551 val accuracy: 0.176000
lr 7.544392e-07 reg 3.078819e+01 train accuracy: 0.162367 val accuracy: 0.133000
lr 7.740266e-07 reg 1.083192e+02 train accuracy: 0.197857 val accuracy: 0.182000
lr 7.803893e-07 reg 1.218351e+03 train accuracy: 0.299163 val accuracy: 0.281000
lr 8.228810e-07 reg 5.096185e+03 train accuracy: 0.424082 val accuracy: 0.432000
lr 8.801491e-07 reg 2.235453e+03 train accuracy: 0.414673 val accuracy: 0.420000
lr 9.131907e-07 reg 4.348301e+02 train accuracy: 0.242061 val accuracy: 0.257000
lr 9.359920e-07 reg 9.036733e+04 train accuracy: 0.410490 val accuracy: 0.408000
lr 9.868756e-07 reg 1.370181e+02 train accuracy: 0.217776 val accuracy: 0.203000
lr 1.008992e-06 reg 4.130031e+03 train accuracy: 0.421204 val accuracy: 0.420000
lr 1.063085e-06 reg 4.924258e+02 train accuracy: 0.288408 val accuracy: 0.317000
lr 1.093926e-06 reg 9.191740e+03 train accuracy: 0.423878 val accuracy: 0.425000
lr 1.162361e-06 reg 4.070752e+04 train accuracy: 0.414102 val accuracy: 0.405000
lr 1.206688e-06 reg 2.715384e+01 train accuracy: 0.212286 val accuracy: 0.233000
lr 1.248321e-06 reg 7.498278e+01 train accuracy: 0.254673 val accuracy: 0.277000
lr 1.256727e-06 reg 9.226247e+01 train accuracy: 0.215020 val accuracy: 0.231000
lr 1.327664e-06 reg 1.127984e+04 train accuracy: 0.422571 val accuracy: 0.432000
lr 1.328575e-06 reg 2.544168e+01 train accuracy: 0.263898 val accuracy: 0.267000
lr 1.386882e-06 reg 1.554533e+04 train accuracy: 0.419694 val accuracy: 0.416000
lr 1.472344e-06 reg 3.961437e+02 train accuracy: 0.327306 val accuracy: 0.338000
lr 1.505270e-06 reg 2.625025e+04 train accuracy: 0.420327 val accuracy: 0.415000
lr 1.560074e-06 reg 2.622690e+02 train accuracy: 0.310245 val accuracy: 0.303000
lr 1.592095e-06 reg 1.277238e+01 train accuracy: 0.257245 val accuracy: 0.260000
lr 1.599685e-06 reg 6.811864e+03 train accuracy: 0.419673 val accuracy: 0.424000
lr 1.680692e-06 reg 7.949338e+01 train accuracy: 0.276571 val accuracy: 0.291000
lr 1.807304e-06 reg 3.930772e+04 train accuracy: 0.418204 val accuracy: 0.426000
lr 1.834229e-06 reg 3.080823e+04 train accuracy: 0.415163 val accuracy: 0.422000
lr 1.857352e-06 reg 1.080742e+03 train accuracy: 0.419592 val accuracy: 0.412000
lr 1.898195e-06 reg 1.946133e+03 train accuracy: 0.421796 val accuracy: 0.432000
lr 1.905938e-06 reg 3.883260e+02 train accuracy: 0.363490 val accuracy: 0.366000
lr 1.947440e-06 reg 3.262071e+04 train accuracy: 0.417143 val accuracy: 0.403000
lr 1.980101e-06 reg 5.758813e+04 train accuracy: 0.407857 val accuracy: 0.405000
lr 2.222751e-06 reg 6.287541e+04 train accuracy: 0.403959 val accuracy: 0.391000
lr 2.248450e-06 reg 6.719434e+02 train accuracy: 0.417571 val accuracy: 0.420000
lr 2.293663e-06 reg 8.004866e+03 train accuracy: 0.423122 val accuracy: 0.412000
lr 2.327069e-06 reg 2.635884e+02 train accuracy: 0.378204 val accuracy: 0.371000
lr 2.350620e-06 reg 8.490437e+02 train accuracy: 0.422286 val accuracy: 0.412000
lr 2.617494e-06 reg 2.497538e+04 train accuracy: 0.417592 val accuracy: 0.426000
lr 2.759967e-06 reg 1.061740e+02 train accuracy: 0.357327 val accuracy: 0.364000
lr 2.780461e-06 reg 2.744577e+04 train accuracy: 0.410796 val accuracy: 0.399000
lr 2.970717e-06 reg 9.735884e+03 train accuracy: 0.424531 val accuracy: 0.429000
lr 3.245099e-06 reg 1.136722e+02 train accuracy: 0.384735 val accuracy: 0.384000
lr 3.384527e-06 reg 1.008812e+02 train accuracy: 0.372122 val accuracy: 0.345000
lr 3.390337e-06 reg 2.508739e+03 train accuracy: 0.421551 val accuracy: 0.417000
lr 3.466104e-06 reg 1.371255e+01 train accuracy: 0.353551 val accuracy: 0.378000
lr 3.481548e-06 reg 6.327564e+04 train accuracy: 0.404204 val accuracy: 0.394000
lr 3.728607e-06 reg 3.004157e+01 train accuracy: 0.370755 val accuracy: 0.348000
lr 3.949796e-06 reg 2.504298e+01 train accuracy: 0.367286 val accuracy: 0.384000
lr 3.959333e-06 reg 4.250235e+04 train accuracy: 0.405673 val accuracy: 0.391000
lr 4.105120e-06 reg 1.011434e+04 train accuracy: 0.421327 val accuracy: 0.436000
lr 4.291552e-06 reg 4.179273e+03 train accuracy: 0.421143 val accuracy: 0.410000
lr 4.361936e-06 reg 5.261546e+04 train accuracy: 0.387143 val accuracy: 0.375000
lr 4.391907e-06 reg 7.379842e+03 train accuracy: 0.421592 val accuracy: 0.421000
lr 4.441526e-06 reg 1.337092e+02 train accuracy: 0.402531 val accuracy: 0.396000
lr 4.646572e-06 reg 1.958151e+02 train accuracy: 0.416020 val accuracy: 0.430000
lr 4.893116e-06 reg 1.912094e+03 train accuracy: 0.421429 val accuracy: 0.425000
lr 5.349085e-06 reg 5.113398e+01 train accuracy: 0.403041 val accuracy: 0.411000
lr 5.485054e-06 reg 7.309816e+03 train accuracy: 0.415082 val accuracy: 0.412000
lr 6.218083e-06 reg 2.940384e+03 train accuracy: 0.423633 val accuracy: 0.423000
lr 6.320323e-06 reg 2.420311e+02 train accuracy: 0.422878 val accuracy: 0.424000
lr 6.558777e-06 reg 1.401850e+03 train accuracy: 0.421898 val accuracy: 0.414000
lr 6.784752e-06 reg 1.371298e+02 train accuracy: 0.418510 val accuracy: 0.411000
lr 6.908389e-06 reg 8.631326e+04 train accuracy: 0.355041 val accuracy: 0.356000
lr 8.557884e-06 reg 7.591051e+04 train accuracy: 0.302224 val accuracy: 0.327000
lr 8.662478e-06 reg 1.394946e+02 train accuracy: 0.421143 val accuracy: 0.415000
lr 8.820757e-06 reg 1.146374e+03 train accuracy: 0.422918 val accuracy: 0.426000
lr 8.953919e-06 reg 3.683136e+04 train accuracy: 0.376918 val accuracy: 0.378000
```

```
lr 8.990801e-06 reg 5.629061e+03 train accuracy: 0.412061 val accuracy: 0.415000
lr 1.021579e-05 reg 3.057906e+02 train accuracy: 0.422959 val accuracy: 0.427000
lr 1.174242e-05 reg 2.264260e+01 train accuracy: 0.421571 val accuracy: 0.424000
lr 1.231112e-05 reg 2.248526e+03 train accuracy: 0.424000 val accuracy: 0.434000
lr 1.235419e-05 reg 8.029376e+02 train accuracy: 0.420857 val accuracy: 0.422000
lr 1.291702e-05 reg 2.821390e+04 train accuracy: 0.377694 val accuracy: 0.383000
lr 1.363308e-05 reg 3.004173e+02 train accuracy: 0.422061 val accuracy: 0.423000
lr 1.369823e-05 reg 2.418206e+04 train accuracy: 0.393061 val accuracy: 0.398000
lr 1.458569e-05 reg 1.652118e+04 train accuracy: 0.391531 val accuracy: 0.407000
lr 1.474567e-05 reg 9.787518e+03 train accuracy: 0.406551 val accuracy: 0.384000
lr 1.507723e-05 reg 2.558308e+04 train accuracy: 0.394429 val accuracy: 0.406000
lr 1.581974e-05 reg 3.298719e+02 train accuracy: 0.421469 val accuracy: 0.410000
lr 1.693449e-05 reg 1.074951e+02 train accuracy: 0.423551 val accuracy: 0.424000
lr 1.700689e-05 reg 9.911403e+02 train accuracy: 0.422673 val accuracy: 0.421000
lr 1.724904e-05 reg 7.054054e+02 train accuracy: 0.426898 val accuracy: 0.440000
lr 1.735279e-05 reg 5.692641e+04 train accuracy: 0.185347 val accuracy: 0.174000
lr 1.796524e-05 reg 1.057199e+04 train accuracy: 0.403429 val accuracy: 0.397000
lr 1.866064e-05 reg 9.198029e+03 train accuracy: 0.416061 val accuracy: 0.411000
lr 1.998988e-05 reg 1.377366e+01 train accuracy: 0.420041 val accuracy: 0.426000
lr 2.035542e-05 reg 5.266937e+04 train accuracy: 0.096469 val accuracy: 0.099000
lr 2.280849e-05 reg 1.604221e+03 train accuracy: 0.417551 val accuracy: 0.426000
lr 2.380328e-05 reg 1.195698e+04 train accuracy: 0.396673 val accuracy: 0.403000
lr 2.390705e-05 reg 2.225763e+02 train accuracy: 0.423735 val accuracy: 0.416000
lr 2.579595e-05 reg 6.507990e+03 train accuracy: 0.389918 val accuracy: 0.384000
lr 2.623867e-05 reg 1.367072e+04 train accuracy: 0.385755 val accuracy: 0.382000
lr 2.677629e-05 reg 3.523630e+03 train accuracy: 0.411245 val accuracy: 0.409000
lr 2.711507e-05 reg 1.064365e+04 train accuracy: 0.390102 val accuracy: 0.378000
lr 2.931307e-05 reg 3.833057e+03 train accuracy: 0.415286 val accuracy: 0.417000
lr 3.278565e-05 reg 5.403398e+02 train accuracy: 0.420122 val accuracy: 0.420000
lr 3.489113e-05 reg 8.753801e+03 train accuracy: 0.393000 val accuracy: 0.400000
lr 3.504692e-05 reg 7.462352e+01 train accuracy: 0.422265 val accuracy: 0.422000
lr 3.648821e-05 reg 7.379983e+02 train accuracy: 0.419531 val accuracy: 0.419000
lr 4.065921e-05 reg 2.804168e+01 train accuracy: 0.422490 val accuracy: 0.424000
lr 4.195540e-05 reg 3.163226e+01 train accuracy: 0.421673 val accuracy: 0.424000
lr 4.359904e-05 reg 1.767997e+02 train accuracy: 0.421490 val accuracy: 0.430000
lr 4.500832e-05 reg 2.420866e+03 train accuracy: 0.406061 val accuracy: 0.407000
lr 4.720947e-05 reg 6.716030e+02 train accuracy: 0.419633 val accuracy: 0.418000
lr 4.772076e-05 reg 1.114902e+01 train accuracy: 0.424204 val accuracy: 0.434000
lr 5.231552e-05 reg 1.079649e+02 train accuracy: 0.422245 val accuracy: 0.424000
lr 5.246390e-05 reg 2.575231e+01 train accuracy: 0.421735 val accuracy: 0.426000
lr 5.833050e-05 reg 5.683436e+01 train accuracy: 0.422571 val accuracy: 0.420000
lr 7.307664e-05 reg 2.048875e+02 train accuracy: 0.416551 val accuracy: 0.411000
lr 7.399916e-05 reg 1.289937e+02 train accuracy: 0.421592 val accuracy: 0.425000
lr 7.424579e-05 reg 1.605049e+01 train accuracy: 0.424367 val accuracy: 0.426000
lr 7.638568e-05 reg 6.064908e+02 train accuracy: 0.415245 val accuracy: 0.407000
lr 7.923079e-05 reg 3.758559e+02 train accuracy: 0.426224 val accuracy: 0.434000
lr 8.297855e-05 reg 7.731149e+03 train accuracy: 0.317633 val accuracy: 0.320000
lr 8.490819e-05 reg 2.933328e+04 train accuracy: 0.100265 val accuracy: 0.087000
best validation accuracy achieved during cross-validation: 0.440000
```

In [85]:
```
# Evaluate your trained SVM on the test set
y_test_pred = best_svm.predict(X_test_feats)
test_accuracy = np.mean(y_test == y_test_pred)
print(test_accuracy)
```

```
0.436
```

In [86]:
```
# An important way to gain intuition about how an algorithm works is to
# visualize the mistakes that it makes. In this visualization, we show examples
# of images that are misclassified by our current system. The first column
# shows images that our system labeled as "plane" but whose true label is
# something other than "plane".

examples_per_class = 8
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
for cls, cls_name in enumerate(classes):
    idxs = np.where((y_test != cls) & (y_test_pred == cls))[0]
    idxs = np.random.choice(idxs, examples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt.subplot(examples_per_class, len(classes), i * len(classes) + cls + 1)
        plt.imshow(X_test[idx].astype('uint8'))
        plt.axis('off')
        if i == 0:
            plt.title(cls_name)
plt.show()
```

Inline question 1:

Describe the misclassification results that you see. Do they make sense?

Answer: The misclassification reflects the low capacity of the SVM model. The classifier is unable to distinguish between images that have similar hues, color ranges and texture patterns but belong to different classes.

Neural Network on image features

Earlier in this assigment we saw that training a two-layer neural network on raw pixels achieved better classification performance than linear classifiers on raw pixels. In this notebook we have seen that linear classifiers on image features outperform linear classifiers on raw pixels.

For completeness, we should also try training a neural network on image features. This approach should outperform all previous approaches: you should easily be able to achieve over 55% classification accuracy on the test set; our best model achieves about 60% classification accuracy.

In [52]:
```
# Preprocessing: Remove the bias dimension
# Make sure to run this cell only ONCE
print(X_train_feats.shape)
X_train_feats = X_train_feats[:, :-1]
X_val_feats = X_val_feats[:, :-1]
X_test_feats = X_test_feats[:, :-1]

print(X_train_feats.shape)
```

```
(49000, 170)
(49000, 169)
```

In [63]:
```
from cs682.classifiers.neural_net import TwoLayerNet

input_dim = X_train_feats.shape[1]
hidden_dim = 500
num_classes = 10

net = TwoLayerNet(input_dim, hidden_dim, num_classes)
best_net = None

################################################################################
# TODO: Train a two-layer neural network on image features. You may want to    #
# cross-validate various parameters as in previous sections. Store your best   #
# model in the best_net variable.                                              #
################################################################################
num_classes = 10

hidden_sizes = [100, 200, 400, 800, 1600]
learning_rates = [1, 1e-1]
regularizations = [1e-4, 1e-3, 1e-2]
batch_sizes = [100, 200, 400, 800]
best_val_acc = 0
best_hyperparams = None
for hs in hidden_sizes:
    for bs in batch_sizes:
        num_trials = 5
        for t in range(num_trials):
            # Train the network
            lr = 10 ** np.random.uniform (-1,0)
            rs = 10 ** np.random.uniform (-4,-2)
            # Train the network
            net = TwoLayerNet(input_dim, hs, num_classes)
            stats = net.train(X_train_feats, y_train, X_val_feats, y_val,
                        num_iters=2000, batch_size=bs,
                        learning_rate=lr, learning_rate_decay=0.95,
                        reg=reg, verbose=False)

            # Predict on the train set
            train_acc = (net.predict(X_train_feats) == y_train).mean()

            # Predict on the validation set
            val_acc = (net.predict(X_val_feats) == y_val).mean()
            print (val_acc, train_acc, hs, bs, lr, reg)
            if best_val_acc < val_acc:
                best_val_acc = val_acc
                best_net = net
                best_hyper_params = (val_acc, train_acc, hs, bs, lr, reg)
print('Best validation accuracy: ', best_val_acc)
print('Best hyper params: ', best_hyper_params)
################################################################################
#                              END OF YOUR CODE                                #
################################################################################
```

0.55 0.6047142857142858 100 100 1 0.0001
0.529 0.5761632653061225 100 100 1 0.001
0.465 0.4729795918367347 100 100 1 0.01
0.543 0.5570816326530612 100 100 0.1 0.0001
0.534 0.5554897959183673 100 100 0.1 0.001
0.521 0.5203061224489796 100 100 0.1 0.01
0.56 0.6576122448979592 100 200 1 0.0001
0.59 0.6366122448979592 100 200 1 0.001
0.493 0.5062040816326531 100 200 1 0.01
0.54 0.5576122448979591 100 200 0.1 0.0001
0.529 0.5529795918367347 100 200 0.1 0.001
0.521 0.526469387755102 100 200 0.1 0.01
0.56 0.6863673469387755 100 400 1 0.0001
0.578 0.6570816326530612 100 400 1 0.001
0.496 0.5086734693877552 100 400 1 0.01
0.537 0.5588775510204081 100 400 0.1 0.0001
0.54 0.5551836734693878 100 400 0.1 0.001
0.516 0.5275714285714286 100 400 0.1 0.01
0.573 0.7108571428571429 100 800 1 0.0001
0.576 0.6875918367346939 100 800 1 0.001
0.513 0.5312244897959184 100 800 1 0.01
0.55 0.5608775510204081 100 800 0.1 0.0001
0.539 0.556 100 800 0.1 0.001
0.515 0.5290408163265307 100 800 0.1 0.01
0.558 0.6370816326530612 200 100 1 0.0001
0.546 0.5980408163265306 200 100 1 0.001
0.471 0.4766530612244898 200 100 1 0.01
0.534 0.5606326530612245 200 100 0.1 0.0001
0.539 0.5564285714285714 200 100 0.1 0.001
0.525 0.5199183673469387 200 100 0.1 0.01
0.567 0.6975918367346938 200 200 1 0.0001
0.57 0.6599591836734694 200 200 1 0.001
0.497 0.5138571428571429 200 200 1 0.01
0.557 0.5609795918367347 200 200 0.1 0.0001
0.534 0.5525714285714286 200 200 0.1 0.001
0.528 0.5267755102040816 200 200 0.1 0.01
0.583 0.7654897959183673 200 400 1 0.0001
0.59 0.711469387755102 200 400 1 0.001
0.491 0.5113877551020408 200 400 1 0.01
0.547 0.5623673469387755 200 400 0.1 0.0001
0.544 0.557795918367347 200 400 0.1 0.001
0.514 0.5262040816326531 200 400 0.1 0.01
0.561 0.790530612244898 200 800 1 0.0001
0.595 0.7328775510204082 200 800 1 0.001
0.542 0.5281020408163265 200 800 1 0.01
0.543 0.5629387755102041 200 800 0.1 0.0001
0.535 0.5585102040816327 200 800 0.1 0.001
0.515 0.5276530612244898 200 800 0.1 0.01
0.564 0.6476326530612245 400 100 1 0.0001
0.512 0.5950204081632653 400 100 1 0.001
0.469 0.4692857142857143 400 100 1 0.01
0.555 0.566204081632653 400 100 0.1 0.0001
0.544 0.559795918367347 400 100 0.1 0.001
0.508 0.5196530612244898 400 100 0.1 0.01
0.57 0.7428979591836735 400 200 1 0.0001
0.588 0.6970816326530612 400 200 1 0.001
0.487 0.4926938775510204 400 200 1 0.01
0.55 0.5624489795918367 400 200 0.1 0.0001
0.538 0.5561836734693878 400 200 0.1 0.001
0.52 0.5270204081632653 400 200 0.1 0.01
0.58 0.8164081632653061 400 400 1 0.0001
0.614 0.7345306122448979 400 400 1 0.001
0.527 0.5236530612244898 400 400 1 0.01
0.547 0.5638775510204082 400 400 0.1 0.0001
0.543 0.5593469387755102 400 400 0.1 0.001
0.513 0.5279183673469388 400 400 0.1 0.01
0.557 0.8463469387755101 400 800 1 0.0001
0.588 0.7664081632653061 400 800 1 0.001
0.522 0.5323673469387755 400 800 1 0.01
0.548 0.566 400 800 0.1 0.0001
0.544 0.5595102040816327 400 800 0.1 0.001
0.515 0.5293877551020408 400 800 0.1 0.01
0.554 0.6726326530612244 800 100 1 0.0001
0.562 0.6278775510204082 800 100 1 0.001
0.47 0.4774285714285714 800 100 1 0.01
0.542 0.570734693877551 800 100 0.1 0.0001

```
0.537 0.5623877551020409 800 100 0.1 0.001
0.511 0.5193469387755102 800 100 0.1 0.01
0.557 0.7961632653061225 800 200 1 0.0001
0.584 0.695734693877551 800 200 1 0.001
0.493 0.49389795918367346 800 200 1 0.01
0.544 0.5645714285714286 800 200 0.1 0.0001
0.546 0.5611224489795918 800 200 0.1 0.001
0.517 0.5264081632653062 800 200 0.1 0.01
0.57 0.8538163265306122 800 400 1 0.0001
0.602 0.7631836734693878 800 400 1 0.001
0.521 0.525 800 400 1 0.01
0.542 0.5671836734693878 800 400 0.1 0.0001
0.545 0.561469387755102 800 400 0.1 0.001
0.519 0.5283061224489796 800 400 0.1 0.01
0.593 0.933 800 800 1 0.0001
0.604 0.7998571428571428 800 800 1 0.001
0.521 0.5389795918367347 800 800 1 0.01
0.549 0.5683469387755102 800 800 0.1 0.0001
0.548 0.5622448979591836 800 800 0.1 0.001
0.519 0.5290204081632653 800 800 0.1 0.01
0.513 0.672530612244898 1600 100 1 0.0001
0.556 0.6290408163265306 1600 100 1 0.001
0.489 0.4944081632653061 1600 100 1 0.01
0.548 0.5693877551020409 1600 100 0.1 0.0001
0.545 0.560734693877551 1600 100 0.1 0.001
0.516 0.5200204081632653 1600 100 0.1 0.01
0.585 0.8117142857142857 1600 200 1 0.0001
0.588 0.7153061224489796 1600 200 1 0.001
0.471 0.4927142857142857 1600 200 1 0.01
0.558 0.5676530612244898 1600 200 0.1 0.0001
0.545 0.562 1600 200 0.1 0.001
0.528 0.5276938775510204 1600 200 0.1 0.01
0.607 0.8999183673469388 1600 400 1 0.0001
0.619 0.7750612244897959 1600 400 1 0.001
0.547 0.5244081632653061 1600 400 1 0.01
0.556 0.569469387755102 1600 400 0.1 0.0001
0.543 0.5635510204081633 1600 400 0.1 0.001
0.525 0.5288775510204081 1600 400 0.1 0.01
0.591 0.9576530612244898 1600 800 1 0.0001
0.616 0.8164081632653061 1600 800 1 0.001
0.518 0.5301632653061225 1600 800 1 0.01
0.555 0.5710408163265306 1600 800 0.1 0.0001
0.551 0.5639795918367347 1600 800 0.1 0.001
0.529 0.5310408163265307 1600 800 0.1 0.01
Best validation accuracy:  0.619
Best hyper params:  (0.619, 0.7750612244897959, 1600, 400, 1, 0.001)
```

In [64]:
```
# Run your best neural net classifier on the test set. You should be able
# to get more than 55% accuracy.

test_acc = (best_net.predict(X_test_feats) == y_test).mean()
print(test_acc)
```

```
0.599
```

In [ ]: