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]() 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.

import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

from __future__ import print_function

# This is a bit of magic to make matplotlib figures appear inline in the no
tebook
# 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-i
python
%load_ext autoreload
%autoreload 2
```

In [2]:

```
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

# Cleaning up variables to prevent loading data multiple times (which may c
ause 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', 's
hip', '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))
```

```
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 cs231n.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 `cs231n/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.

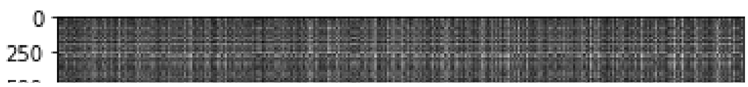
In [7]:

```
# Open cs231n/classifiers/k_nearest_neighbor.py and implement
# compute_distances_two_loops.

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

```
(500, 5000)
```

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:

The rows are caused by the test example associated with that row being a high distance away from a great number of training examples.

The columns are caused by the training example associated with that column being a high distance away from a great number of 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, accur
acy))
```

```
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, accur
acy))
```

```
Got 136 / 500 correct => accuracy: 0.272000
```

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. None of the above.

Your explanation: 1 is not true because preprocessing the data by subtracting the mean changes the relative distance between data. 2 is not true for the same reason as 1. 3 is not true because L1 distance (AKA Manhattan distance) depends on the rotation of the coordinate system, so rotating the coordinate axes would change the relative distance between data. By elimination, our answer is 4.

In [11]:

```
# Now lets speed up distance matrix computation by using partial vectorizat
ion
# 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 tha
t it
# agrees with the naive implementation. There are many ways to decide wheth
er
# 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, r
eshape
# 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_tes
t)
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 26.996551 seconds
One loop version took 51.321399 seconds
No loop version took 0.332472 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)
y_train_folds = np.array_split(y_train, num_folds)
###########################################################################
##
#                                END OF YOUR CODE
```

```
#
################################################################################
##

# A dictionary holding the accuracies for different values of k that we fin
d
# when running cross-validation. After running cross-validation,
# k_to_accuracies[k] should be a list of length num_folds giving the differ
ent
# 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:
    k_to_accuracies[k] = []
    for i in range(num_folds):
        xtr = np.concatenate([X_train_folds[j] for j in range(num_folds) if
i != j])
        ytr = np.concatenate([y_train_folds[j] for j in range(num_folds) if
i != j])
        classifier = KNearestNeighbor()
        classifier.train(xtr, ytr)
        pred = classifier.predict_labels(X_train_folds[i], k)
        acc = float(sum(pred == y_train_folds[i]) / len(pred))
        k_to_accuracies[k] += [acc]
################################################################################
##
#                                 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))
```

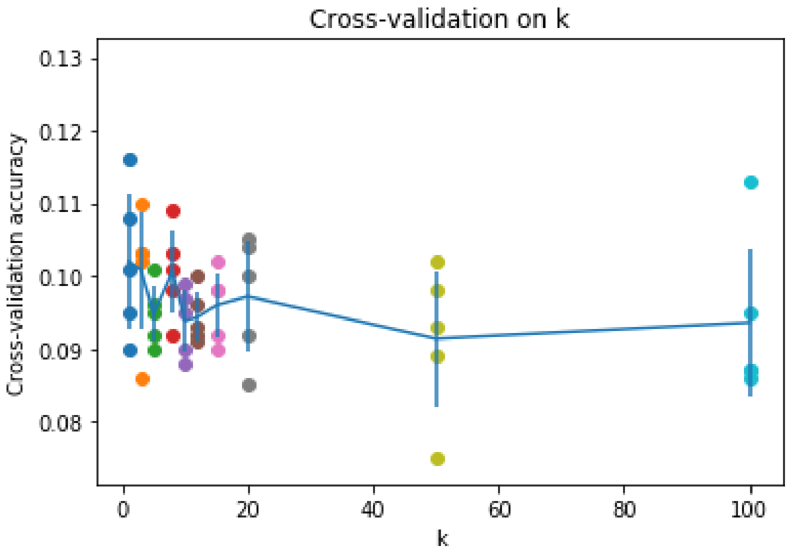
```
k = 1, accuracy = 0.108000
k = 1, accuracy = 0.090000
k = 1, accuracy = 0.095000
k = 1, accuracy = 0.101000
k = 1, accuracy = 0.116000
k = 3, accuracy = 0.103000
k = 3, accuracy = 0.086000
```

```
k = 3, accuracy = 0.103000
k = 3, accuracy = 0.110000
k = 3, accuracy = 0.102000
k = 5, accuracy = 0.101000
k = 5, accuracy = 0.090000
k = 5, accuracy = 0.092000
k = 5, accuracy = 0.096000
k = 5, accuracy = 0.095000
k = 8, accuracy = 0.109000
k = 8, accuracy = 0.092000
k = 8, accuracy = 0.098000
k = 8, accuracy = 0.101000
k = 8, accuracy = 0.103000
k = 10, accuracy = 0.095000
k = 10, accuracy = 0.097000
k = 10, accuracy = 0.090000
k = 10, accuracy = 0.099000
k = 10, accuracy = 0.088000
k = 12, accuracy = 0.093000
k = 12, accuracy = 0.091000
k = 12, accuracy = 0.092000
k = 12, accuracy = 0.100000
k = 12, accuracy = 0.096000
k = 15, accuracy = 0.098000
k = 15, accuracy = 0.092000
k = 15, accuracy = 0.102000
k = 15, accuracy = 0.098000
k = 15, accuracy = 0.090000
k = 20, accuracy = 0.092000
k = 20, accuracy = 0.085000
k = 20, accuracy = 0.104000
k = 20, accuracy = 0.105000
k = 20, accuracy = 0.100000
k = 50, accuracy = 0.075000
k = 50, accuracy = 0.102000
k = 50, accuracy = 0.093000
k = 50, accuracy = 0.089000
k = 50, accuracy = 0.098000
k = 100, accuracy = 0.086000
k = 100, accuracy = 0.095000
k = 100, accuracy = 0.087000
k = 100, accuracy = 0.113000
k = 100, accuracy = 0.087000
```

In [16]:

```
# 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.it
ems())])
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 [17]:

```
# 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 te
st
# data. You should be able to get above 28% accuracy on the test data.
best_k = max(k_to_accuracies, key=k_to_accuracies.get)
print('Best k = {}'.format(best_k))

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, accur
acy))
```

```
Best k = 8
Got 144 / 500 correct => accuracy: 0.288000
```

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: 1 is not true because there are counterexamples where 5-NN performs better than 1-NN (e.g. where the the boundaries between two classes are very close to each other).

2 is not true for the same reason as 1.

3 is not true because the density of regions of data points can produce decision boundaries that are curved (i.e. non-linear).

4 is true because the number of other data points we must compare a test data point to (and therefore computational complexity) grows with 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](#) 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 [65]:

```
# Run some setup code for this notebook.

import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

from __future__ import print_function

# 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-i
python
%load_ext autoreload
%autoreload 2
```

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

CIFAR-10 Data Loading and Preprocessing

In [66]:

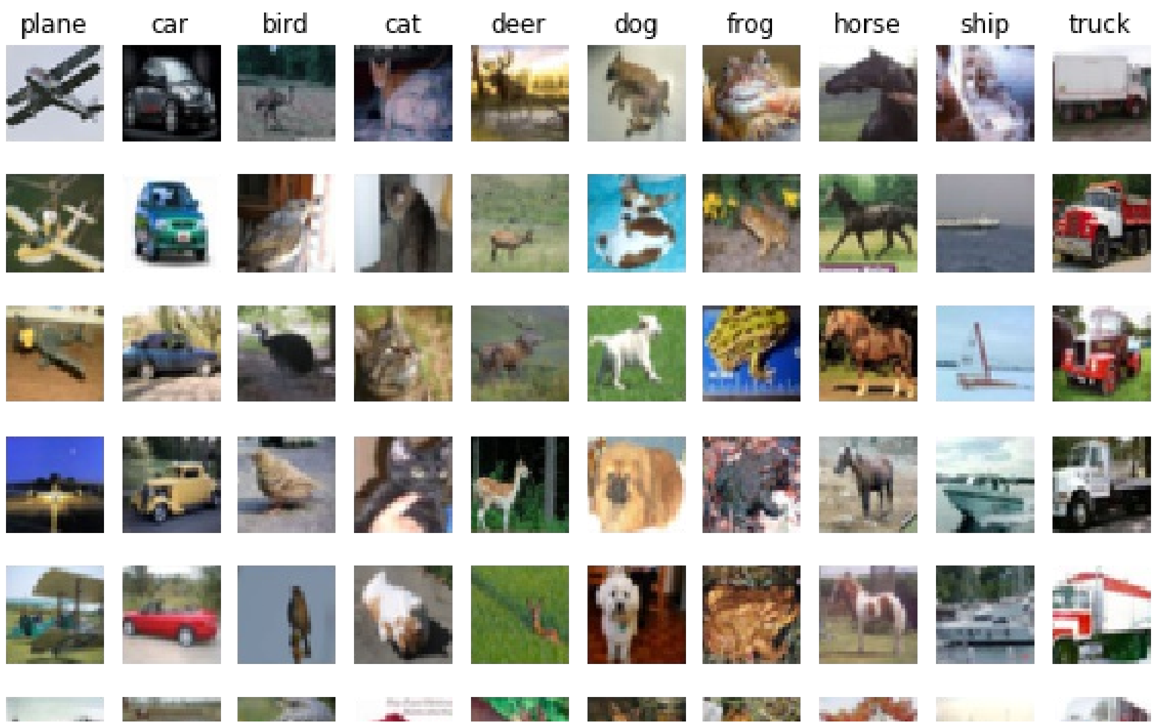
```
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

# Cleaning up variables to prevent loading data multiple times (which may c
ause memory issue)
try:
```

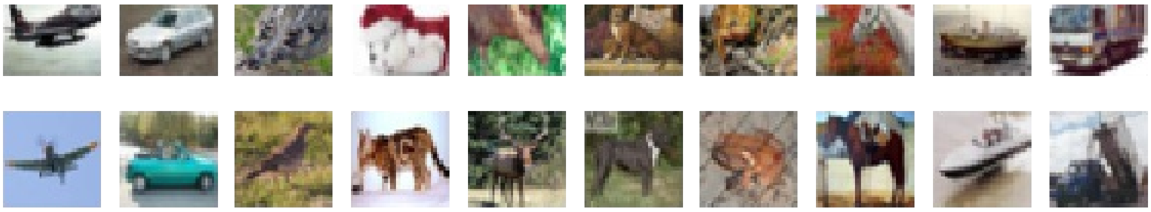
```
    del X_train, y_train
    del X_test, y_test
    print('Clear previously loaded data.')
except:
    pass

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)
```

```
Clear previously loaded data.
Training data shape:  (50000, 32, 32, 3)
Training labels shape:  (50000,)
Test data shape:  (10000, 32, 32, 3)
Test labels shape:  (10000,)
```

In [67]:

```
# 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', 's
hip', '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 [68]:

```
# 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 [69]:

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

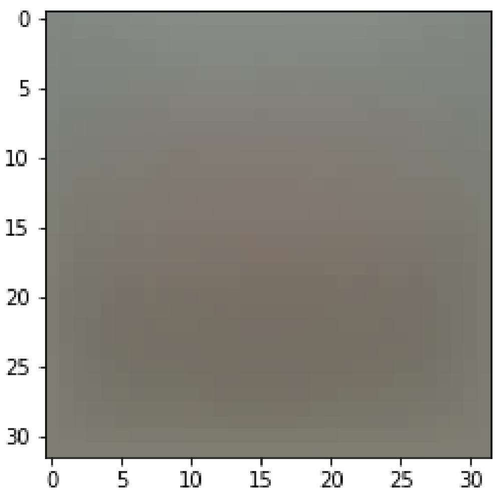
```
X_val = np.reshape(X_val, (X_val.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

# 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 [70]:

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

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



In [71]:

```
# 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 [72]:

```
# third: append the bias dimension of ones (i.e. bias trick) so that our SV
M
# 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))])
```

```
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 **cs231n/classifiers/linear_svm.py**.

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

In [73]:

```
# Evaluate the naive implementation of the loss we provided for you:
from cs231n.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: 9.181935
```

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 [74]:

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

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

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

# do the gradient check once again with regularization turned on
# you didn't forget the regularization gradient did you?
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: -8.345527 analytic: -8.345527, relative error: 3.357752e-11
numerical: -14.186275 analytic: -14.186275, relative error: 2.145899e-12
numerical: 13.665230 analytic: 13.665230, relative error: 3.059449e-11
numerical: -3.798422 analytic: -3.798422, relative error: 6.967311e-11
numerical: 10.181495 analytic: 10.181495, relative error: 2.019917e-11
numerical: -15.241853 analytic: -15.241853, relative error: 3.893933e-12
numerical: -24.744685 analytic: -24.744685, relative error: 1.540335e-11
numerical: 14.790684 analytic: 14.790684, relative error: 1.802108e-11
numerical: -3.797349 analytic: -3.797349, relative error: 4.916500e-11
numerical: -4.460438 analytic: -4.460438, relative error: 2.903690e-11
numerical: 13.455027 analytic: 13.455027, relative error: 1.685483e-11
numerical: -25.887387 analytic: -25.887387, relative error: 2.015178e-11
numerical: 2.810013 analytic: 2.810013, relative error: 4.635519e-11
numerical: -0.461431 analytic: -0.461431, relative error: 4.017301e-10
numerical: -0.449056 analytic: -0.449056, relative error: 2.081376e-10
numerical: -12.731837 analytic: -12.731837, relative error: 1.801152e-11
numerical: -3.158156 analytic: -3.158156, relative error: 3.199042e-11
numerical: 15.799358 analytic: 15.799358, relative error: 4.696614e-11
numerical: -2.002246 analytic: -2.002246, relative error: 4.427072e-11
numerical: 1.519818 analytic: 1.519818, relative error: 7.768867e-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:

In [75]:

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

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

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

```
Naive loss: 9.181935e+00 computed in 0.096224s
Vectorized loss: 9.181935e+00 computed in 0.475134s
difference: 0.000000
```

In [76]:

```
# Complete the implementation of svm_loss_vectorized, and compute the gradi
ent
# 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 0.149554s
Vectorized loss and gradient: computed in 0.004971s
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.

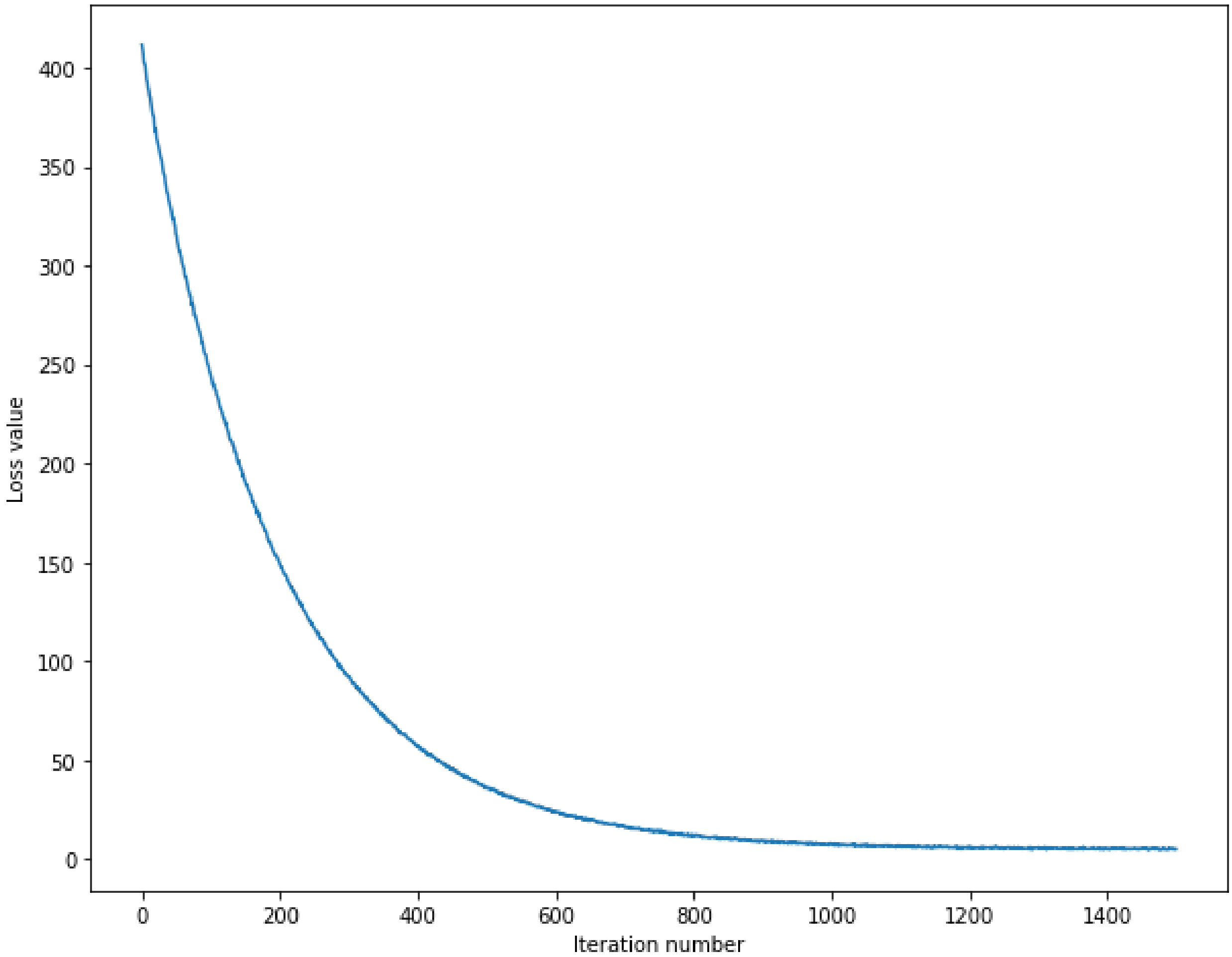
In [77]:

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

```
iteration 0 / 1500: loss 411.595666
iteration 100 / 1500: loss 244.753993
iteration 200 / 1500: loss 149.318287
iteration 300 / 1500: loss 92.124630
iteration 400 / 1500: loss 56.932667
iteration 500 / 1500: loss 36.144869
iteration 600 / 1500: loss 23.874836
iteration 700 / 1500: loss 16.263667
iteration 800 / 1500: loss 11.711825
iteration 900 / 1500: loss 9.321136
iteration 1000 / 1500: loss 7.691581
iteration 1100 / 1500: loss 7.019264
iteration 1200 / 1500: loss 5.946215
iteration 1300 / 1500: loss 6.139099
iteration 1400 / 1500: loss 5.421402
That took 5.957175s
```

In [78]:

```
# 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 [79]:

```
# 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: 0.382327
validation accuracy: 0.390000
```

In [80]:

```
# Use the validation set to tune hyperparameters (regularization strength a
nd
# learning rate). You should experiment with different ranges for the
learning
# rates and regularization strengths; if you are careful you should be able
to
# get a classification accuracy of about 0.4 on the validation set.
learning_rates = [1e-7, 5e-7, 1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-8]
```

```
learning_rates = [1e-7, 5e-7, 1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-0]
regularization_strengths = [2.5e4, 5e4, 1e5, 1e6, 1e3, 1e2]

# 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 validat
ion #
# 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 bes
t   #
# 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 validatio
n   #
# code with a larger value for num_iters.
#
###########################################################################
##
num_iters = 500

for lr in learning_rates:
    for reg in regularization_strengths:
        svm = LinearSVM()
        svm.train(X_train, y_train, learning_rate=lr, reg=reg, num_iters=num
_iters)
        y_train_pred = svm.predict(X_train)
        acc_train = np.mean(y_train_pred == y_train)
        y_val_pred = svm.predict(X_val)
        acc_val = np.mean(y_val_pred == y_val)
        results[(lr, reg)] = (acc_train, acc_val)

        if acc_val > best_val:
            best_val = acc_val
            best_svm = svm
###########################################################################
##
#                              END OF YOUR CODE
#
```

```
#
################################################################################
##

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

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

```
/Users/ianscottknight/Google Drive/current
school/cs231n/assignment1/cs231n/classifiers/linear_svm.py:83:
RuntimeWarning: overflow encountered in double_scalars
  loss += 0.5 * reg * np.sum(W * W)
/Users/ianscottknight/anaconda/envs/cs231n/lib/python3.6/site-
packages/numpy/core/_methods.py:32: RuntimeWarning: overflow encountered in
reduce
  return umr_sum(a, axis, dtype, out, keepdims)
/Users/ianscottknight/Google Drive/current
school/cs231n/assignment1/cs231n/classifiers/linear_svm.py:83:
RuntimeWarning: overflow encountered in multiply
  loss += 0.5 * reg * np.sum(W * W)
/Users/ianscottknight/Google Drive/current
school/cs231n/assignment1/cs231n/classifiers/linear_svm.py:105: RuntimeWarn
ing: overflow encountered in multiply
  dW += reg*W
/Users/ianscottknight/Google Drive/current
school/cs231n/assignment1/cs231n/classifiers/linear_svm.py:79:
RuntimeWarning: invalid value encountered in maximum
  margin = np.maximum(0, scores - correct_class_score[:, np.newaxis] + delt
a)
/Users/ianscottknight/Google Drive/current
school/cs231n/assignment1/cs231n/classifiers/linear_svm.py:99:
RuntimeWarning: invalid value encountered in greater
  X_mask[margin > 0] = 1
/Users/ianscottknight/Google Drive/current
school/cs231n/assignment1/cs231n/classifiers/linear_classifier.py:70: Runti
meWarning: invalid value encountered in add
  self.W += -learning_rate*grad
/Users/ianscottknight/Google Drive/current
school/cs231n/assignment1/cs231n/classifiers/linear_svm.py:79:
RuntimeWarning: overflow encountered in subtract
  margin = np.maximum(0, scores - correct_class_score[:, np.newaxis] + delt
a)
/Users/ianscottknight/Google Drive/current
school/cs231n/assignment1/cs231n/classifiers/linear_svm.py:79:
RuntimeWarning: invalid value encountered in subtract
  margin = np.maximum(0, scores - correct_class_score[:, np.newaxis] + delt
a)
```

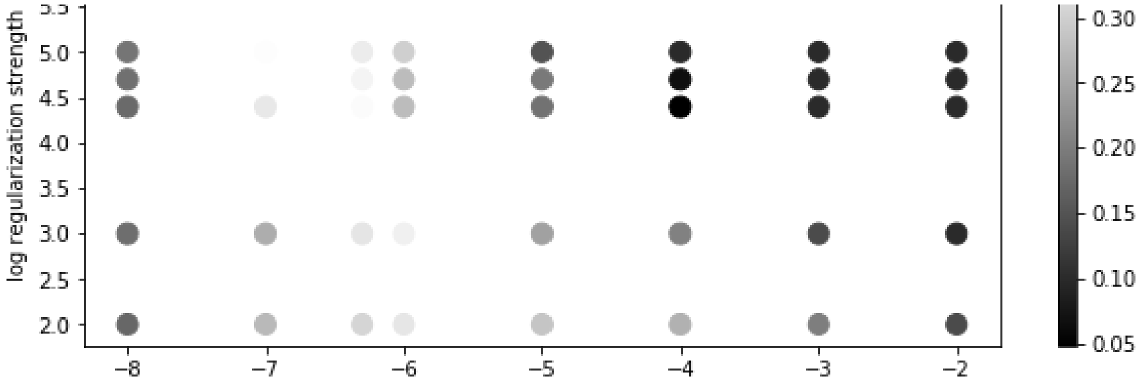
```
lr 1.000000e-08 reg 1.000000e+02 train accuracy: 0.173612 val accuracy: 0.1
80000
lr 1.000000e-08 reg 1.000000e+03 train accuracy: 0.182816 val accuracy: 0.1
87000
lr 1.000000e-08 reg 2.500000e+04 train accuracy: 0.177959 val accuracy: 0.1
86000
lr 1.000000e-08 reg 5.000000e+04 train accuracy: 0.185653 val accuracy: 0.1
91000
```

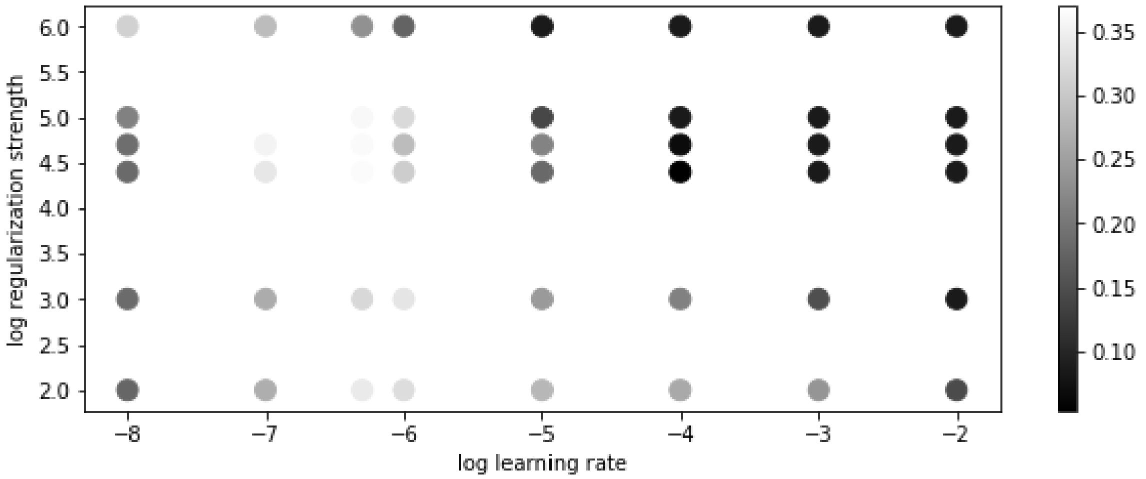
```
lr 1.000000e-08 reg 1.000000e+05 train accuracy: 0.190612 val accuracy: 0.2
14000
lr 1.000000e-08 reg 1.000000e+06 train accuracy: 0.297980 val accuracy: 0.3
12000
lr 1.000000e-07 reg 1.000000e+02 train accuracy: 0.273163 val accuracy: 0.2
69000
lr 1.000000e-07 reg 1.000000e+03 train accuracy: 0.259429 val accuracy: 0.2
67000
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.330816 val accuracy: 0.3
40000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.359143 val accuracy: 0.3
54000
lr 1.000000e-07 reg 1.000000e+05 train accuracy: 0.356449 val accuracy: 0.3
70000
lr 1.000000e-07 reg 1.000000e+06 train accuracy: 0.282082 val accuracy: 0.2
87000
lr 5.000000e-07 reg 1.000000e+02 train accuracy: 0.307857 val accuracy: 0.3
43000
lr 5.000000e-07 reg 1.000000e+03 train accuracy: 0.326918 val accuracy: 0.3
21000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.353653 val accuracy: 0.3
64000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.344735 val accuracy: 0.3
63000
lr 5.000000e-07 reg 1.000000e+05 train accuracy: 0.335592 val accuracy: 0.3
61000
lr 5.000000e-07 reg 1.000000e+06 train accuracy: 0.232796 val accuracy: 0.2
33000
lr 1.000000e-06 reg 1.000000e+02 train accuracy: 0.328755 val accuracy: 0.3
27000
lr 1.000000e-06 reg 1.000000e+03 train accuracy: 0.340531 val accuracy: 0.3
37000
lr 1.000000e-06 reg 2.500000e+04 train accuracy: 0.277490 val accuracy: 0.3
08000
lr 1.000000e-06 reg 5.000000e+04 train accuracy: 0.277816 val accuracy: 0.2
87000
lr 1.000000e-06 reg 1.000000e+05 train accuracy: 0.299959 val accuracy: 0.3
23000
lr 1.000000e-06 reg 1.000000e+06 train accuracy: 0.177143 val accuracy: 0.1
77000
lr 1.000000e-05 reg 1.000000e+02 train accuracy: 0.288245 val accuracy: 0.2
80000
lr 1.000000e-05 reg 1.000000e+03 train accuracy: 0.243939 val accuracy: 0.2
45000
lr 1.000000e-05 reg 2.500000e+04 train accuracy: 0.188061 val accuracy: 0.1
84000
lr 1.000000e-05 reg 5.000000e+04 train accuracy: 0.195939 val accuracy: 0.2
16000
lr 1.000000e-05 reg 1.000000e+05 train accuracy: 0.150102 val accuracy: 0.1
42000
lr 1.000000e-05 reg 1.000000e+06 train accuracy: 0.100265 val accuracy: 0.0
87000
lr 1.000000e-04 reg 1.000000e+02 train accuracy: 0.263816 val accuracy: 0.2
63000
lr 1.000000e-04 reg 1.000000e+03 train accuracy: 0.205633 val accuracy: 0.2
14000
lr 1.000000e-04 reg 2.500000e+04 train accuracy: 0.048592 val accuracy: 0.0
54000
lr 1.000000e-04 reg 5.000000e+04 train accuracy: 0.065898 val accuracy: 0.0
68000
```

```
lr 1.000000e-04 reg 1.000000e+05 train accuracy: 0.100265 val accuracy: 0.0
87000
lr 1.000000e-04 reg 1.000000e+06 train accuracy: 0.100265 val accuracy: 0.0
87000
lr 1.000000e-03 reg 1.000000e+02 train accuracy: 0.202061 val accuracy: 0.2
39000
lr 1.000000e-03 reg 1.000000e+03 train accuracy: 0.139898 val accuracy: 0.1
52000
lr 1.000000e-03 reg 2.500000e+04 train accuracy: 0.100265 val accuracy: 0.0
87000
lr 1.000000e-03 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.0
87000
lr 1.000000e-03 reg 1.000000e+05 train accuracy: 0.100265 val accuracy: 0.0
87000
lr 1.000000e-03 reg 1.000000e+06 train accuracy: 0.100265 val accuracy: 0.0
87000
lr 1.000000e-02 reg 1.000000e+02 train accuracy: 0.139796 val accuracy: 0.1
47000
lr 1.000000e-02 reg 1.000000e+03 train accuracy: 0.100265 val accuracy: 0.0
87000
lr 1.000000e-02 reg 2.500000e+04 train accuracy: 0.100265 val accuracy: 0.0
87000
lr 1.000000e-02 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.0
87000
lr 1.000000e-02 reg 1.000000e+05 train accuracy: 0.100265 val accuracy: 0.0
87000
lr 1.000000e-02 reg 1.000000e+06 train accuracy: 0.100265 val accuracy: 0.0
87000
best validation accuracy achieved during cross-validation: 0.370000
```

In [81]:

```python
# 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 [82]:

```
# 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_accurac
y)
```

```
linear SVM on raw pixels final test set accuracy: 0.363000
```

In [83]:

```
# Visualize the learned weights for each class.
# Depending on your choice of learning rate and regularization strength, th
ese 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', 's
hip', '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 visualized SVM weights look like a very low-resolution image of the subject they purport to represent. For example, the visualized weights for the "ship" class have a distinct border of blue pixels (i.e. water/ocean) with a center object of different color (i.e. the ship). For the "frog" class, the main object is green, which makes sense since frogs are green. In this way, the visualized weights appear as "prototypical" images of their associated classes.

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] 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 [85]:

```
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

from __future__ import print_function

%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-i
python
%load_ext autoreload
%autoreload 2
```

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

In [86]:

```
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 prepar
e
    it for the linear classifier. These are the same steps as we used for t
he
    SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

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

    # subsample the data
```

```
    # 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 c
ause 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)
```

```
Clear previously loaded data.
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 **cs231n/classifiers/softmax.py**.

In [87]:

```
# First implement the naive softmax loss function with nested loops.
# Open the file cs231n/classifiers/softmax.py and implement the
# softmax_loss_naive function.

from cs231n.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: 2.323632
sanity check: 2.302585
```

Inline Question 1:

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

Your answer: Because this correlates with the equal probability of choosing any class (1 / 10 = 0.1).

In [88]:

```
# 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 cs231n.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: 1.861162 analytic: 1.861162, relative error: 6.344120e-09
numerical: 0.864603 analytic: 0.864603, relative error: 4.105155e-08
numerical: 0.861921 analytic: 0.861921, relative error: 1.063835e-08
numerical: 0.346938 analytic: 0.346938, relative error: 7.682057e-09
```

```
numerical: 0.574901 analytic: 0.574901, relative error: 1.020259e-07
numerical: -0.296371 analytic: -0.296371, relative error: 2.638844e-08
numerical: -0.126350 analytic: -0.126350, relative error: 4.461955e-07
numerical: -0.359912 analytic: -0.359912, relative error: 6.022621e-08
numerical: -0.378025 analytic: -0.378025, relative error: 5.612359e-08
numerical: -1.141886 analytic: -1.141886, relative error: 6.757137e-08
numerical: -2.434841 analytic: -2.434841, relative error: 2.471309e-08
numerical: -0.051010 analytic: -0.051010, relative error: 7.065835e-07
numerical: -1.602608 analytic: -1.602608, relative error: 1.150269e-08
numerical: 2.679628 analytic: 2.679628, relative error: 1.993523e-08
numerical: -2.600769 analytic: -2.600769, relative error: 1.938812e-08
numerical: -1.510904 analytic: -1.510904, relative error: 9.770688e-09
numerical: -0.105186 analytic: -0.105186, relative error: 1.366734e-07
numerical: 0.049258 analytic: 0.049258, relative error: 5.227336e-07
numerical: -2.860923 analytic: -2.860923, relative error: 1.137793e-08
numerical: -0.512653 analytic: -0.512653, relative error: 4.109639e-08
```

In [89]:

```
# 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 cs231n.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: 2.323632e+00 computed in 0.215279s
vectorized loss: 2.323632e+00 computed in 0.168685s
Loss difference: 0.000000
Gradient difference: 0.000000
```

In [90]:

```
# 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 cs231n.classifiers import Softmax
results = {}
```

```
best_val = -1
best_softmax = None
learning_rates = [1e-7, 5e-7, 1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-8]
regularization_strengths = [2.5e4, 5e4, 1e5, 1e6, 1e3, 1e2]

###########################################################################
##
# 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.
#
###########################################################################
##

num_iters = 500

for lr in learning_rates:
    for reg in regularization_strengths:
        softmax = Softmax()
        softmax.train(X_train, y_train, learning_rate=lr, reg=reg,
num_iters=num_iters)
        y_train_pred = softmax.predict(X_train)
        acc_train = np.mean(y_train_pred == y_train)
        y_val_pred = softmax.predict(X_val)
        acc_val = np.mean(y_val_pred == y_val)
        results[(lr, reg)] = (acc_train, acc_val)

        if acc_val > best_val:
            best_val = acc_val
            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' % bes
t_val)
```

```
/Users/ianscottknight/Google Drive/current
school/cs231n/assignment1/cs231n/classifiers/softmax.py:83: RuntimeWarning:
divide by zero encountered in log
  loss = np.sum(-np.log(p[np.arange(n_train), y]))
/Users/ianscottknight/Google Drive/current
school/cs231n/assignment1/cs231n/classifiers/softmax.py:90: RuntimeWarning:
overflow encountered in double_scalars
  loss += 0.5 * reg * np.sum(W * W)
/Users/ianscottknight/anaconda/envs/cs231n/lib/python3.6/site-
```

```
packages/numpy/core/_methods.py:32: RuntimeWarning: overflow encountered in reduce
  return umr_sum(a, axis, dtype, out, keepdims)
/Users/ianscottknight/Google Drive/current school/cs231n/assignment1/cs231n/classifiers/softmax.py:90: RuntimeWarning: overflow encountered in multiply
  loss += 0.5 * reg * np.sum(W * W)
/Users/ianscottknight/Google Drive/current school/cs231n/assignment1/cs231n/classifiers/softmax.py:92: RuntimeWarning: overflow encountered in multiply
  dW += reg*W
/Users/ianscottknight/anaconda/envs/cs231n/lib/python3.6/site-packages/numpy/core/_methods.py:26: RuntimeWarning: invalid value encountered in reduce
  return umr_maximum(a, axis, None, out, keepdims)
/Users/ianscottknight/Google Drive/current school/cs231n/assignment1/cs231n/classifiers/softmax.py:80: RuntimeWarning: overflow encountered in subtract
  f -= np.max(f, axis=1, keepdims=True)
/Users/ianscottknight/Google Drive/current school/cs231n/assignment1/cs231n/classifiers/softmax.py:80: RuntimeWarning: invalid value encountered in subtract
  f -= np.max(f, axis=1, keepdims=True)
```

```
lr 1.000000e-08 reg 1.000000e+02 train accuracy: 0.099571 val accuracy: 0.092000
lr 1.000000e-08 reg 1.000000e+03 train accuracy: 0.114245 val accuracy: 0.125000
lr 1.000000e-08 reg 2.500000e+04 train accuracy: 0.139878 val accuracy: 0.159000
lr 1.000000e-08 reg 5.000000e+04 train accuracy: 0.116327 val accuracy: 0.101000
lr 1.000000e-08 reg 1.000000e+05 train accuracy: 0.121041 val accuracy: 0.110000
lr 1.000000e-08 reg 1.000000e+06 train accuracy: 0.253592 val accuracy: 0.265000
lr 1.000000e-07 reg 1.000000e+02 train accuracy: 0.205673 val accuracy: 0.222000
lr 1.000000e-07 reg 1.000000e+03 train accuracy: 0.200592 val accuracy: 0.201000
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.266286 val accuracy: 0.287000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.310408 val accuracy: 0.325000
lr 1.000000e-07 reg 1.000000e+05 train accuracy: 0.297388 val accuracy: 0.320000
lr 1.000000e-07 reg 1.000000e+06 train accuracy: 0.255510 val accuracy: 0.270000
lr 5.000000e-07 reg 1.000000e+02 train accuracy: 0.270061 val accuracy: 0.240000
lr 5.000000e-07 reg 1.000000e+03 train accuracy: 0.285898 val accuracy: 0.296000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.347714 val accuracy: 0.359000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.334918 val accuracy: 0.346000
lr 5.000000e-07 reg 1.000000e+05 train accuracy: 0.305776 val accuracy: 0.319000
lr 5.000000e-07 reg 1.000000e+06 train accuracy: 0.254755 val accuracy: 0.267000
```

```
lr 1.000000e-06 reg 1.000000e+02 train accuracy: 0.299082 val accuracy: 0.3
17000
lr 1.000000e-06 reg 1.000000e+03 train accuracy: 0.331327 val accuracy: 0.3
39000
lr 1.000000e-06 reg 2.500000e+04 train accuracy: 0.342286 val accuracy: 0.3
45000
lr 1.000000e-06 reg 5.000000e+04 train accuracy: 0.307694 val accuracy: 0.3
15000
lr 1.000000e-06 reg 1.000000e+05 train accuracy: 0.307796 val accuracy: 0.3
15000
lr 1.000000e-06 reg 1.000000e+06 train accuracy: 0.228408 val accuracy: 0.2
39000
lr 1.000000e-05 reg 1.000000e+02 train accuracy: 0.346959 val accuracy: 0.3
45000
lr 1.000000e-05 reg 1.000000e+03 train accuracy: 0.218000 val accuracy: 0.2
16000
lr 1.000000e-05 reg 2.500000e+04 train accuracy: 0.201857 val accuracy: 0.2
18000
lr 1.000000e-05 reg 5.000000e+04 train accuracy: 0.171245 val accuracy: 0.1
78000
lr 1.000000e-05 reg 1.000000e+05 train accuracy: 0.071245 val accuracy: 0.0
67000
lr 1.000000e-05 reg 1.000000e+06 train accuracy: 0.100265 val accuracy: 0.0
87000
lr 1.000000e-04 reg 1.000000e+02 train accuracy: 0.275592 val accuracy: 0.2
65000
lr 1.000000e-04 reg 1.000000e+03 train accuracy: 0.200184 val accuracy: 0.2
07000
lr 1.000000e-04 reg 2.500000e+04 train accuracy: 0.058082 val accuracy: 0.0
59000
lr 1.000000e-04 reg 5.000000e+04 train accuracy: 0.084204 val accuracy: 0.0
86000
lr 1.000000e-04 reg 1.000000e+05 train accuracy: 0.100265 val accuracy: 0.0
87000
lr 1.000000e-04 reg 1.000000e+06 train accuracy: 0.100265 val accuracy: 0.0
87000
lr 1.000000e-03 reg 1.000000e+02 train accuracy: 0.210388 val accuracy: 0.2
02000
lr 1.000000e-03 reg 1.000000e+03 train accuracy: 0.144776 val accuracy: 0.1
47000
lr 1.000000e-03 reg 2.500000e+04 train accuracy: 0.100265 val accuracy: 0.0
87000
lr 1.000000e-03 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.0
87000
lr 1.000000e-03 reg 1.000000e+05 train accuracy: 0.100265 val accuracy: 0.0
87000
lr 1.000000e-03 reg 1.000000e+06 train accuracy: 0.100265 val accuracy: 0.0
87000
lr 1.000000e-02 reg 1.000000e+02 train accuracy: 0.085980 val accuracy: 0.0
84000
lr 1.000000e-02 reg 1.000000e+03 train accuracy: 0.100265 val accuracy: 0.0
87000
lr 1.000000e-02 reg 2.500000e+04 train accuracy: 0.100265 val accuracy: 0.0
87000
lr 1.000000e-02 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.0
87000
lr 1.000000e-02 reg 1.000000e+05 train accuracy: 0.100265 val accuracy: 0.0
87000
lr 1.000000e-02 reg 1.000000e+06 train accuracy: 0.100265 val accuracy: 0.0
87000
```

```
best validation accuracy achieved during cross-validation: 0.359000
```

In [91]:

```python
# 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: 0.362000
```

Inline Question - *True or False*

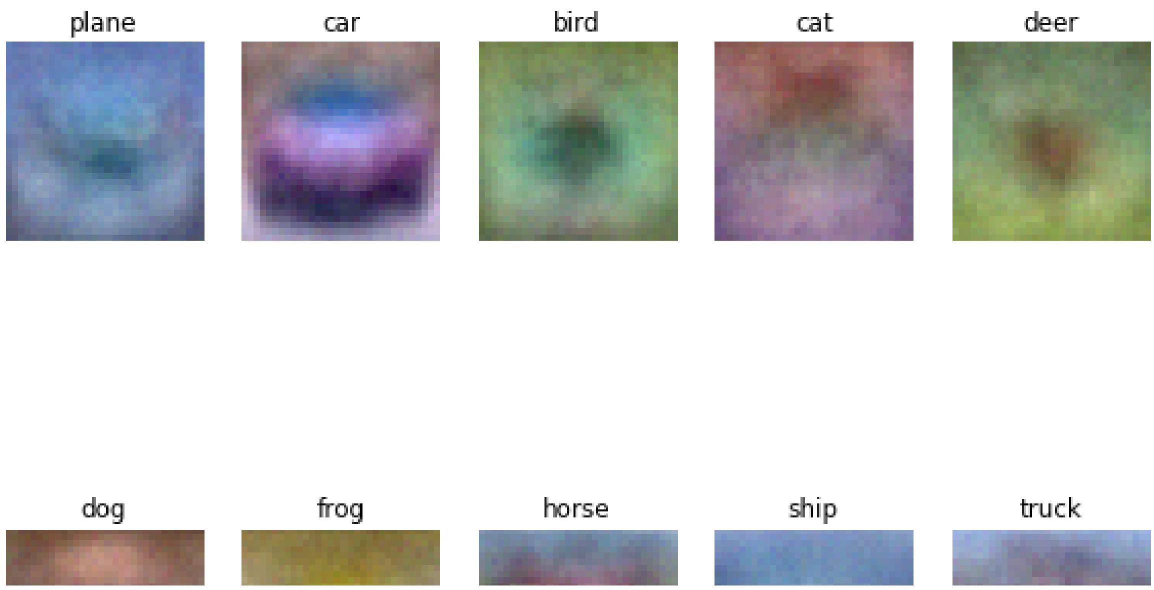
It's possible to add a new datapoint to a training set that would leave the SVM loss unchanged, but this is not the case with the Softmax classifier loss.

Your answer: True

Your explanation: In the case of SVM, it is possible for a new data point to produce a score that is outside of the margin range from the correct class score, such that the loss remains unchanged. However, there is no such scenario in the case of Softmax, since it does make use of a margin.

In [92]:

```python
# 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', 's
hip', '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])
```



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 [113]:

```
# A bit of setup

import numpy as np
import matplotlib.pyplot as plt

from cs231n.classifiers.neural_net import TwoLayerNet

from __future__ import print_function

%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-i
python
%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))))
```

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

We will use the class `TwoLayerNet` in the file `cs231n/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 [114]:

```
# 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 `cs231n/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 [115]:

```
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 [116]:

```
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:
1.794120407794253e-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 [117]:

```
from cs231n.gradient_check import eval_numerical_gradient

# Use numeric gradient checking to check your implementation of the backwar
d 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('%s max relative error: %e' % (param_name,
rel_error(param_grad_num, grads[param_name])))
```

```
W2 max relative error: 3.440708e-09
b2 max relative error: 3.865091e-11
W1 max relative error: 3.561318e-09
b1 max relative error: 2.738421e-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.

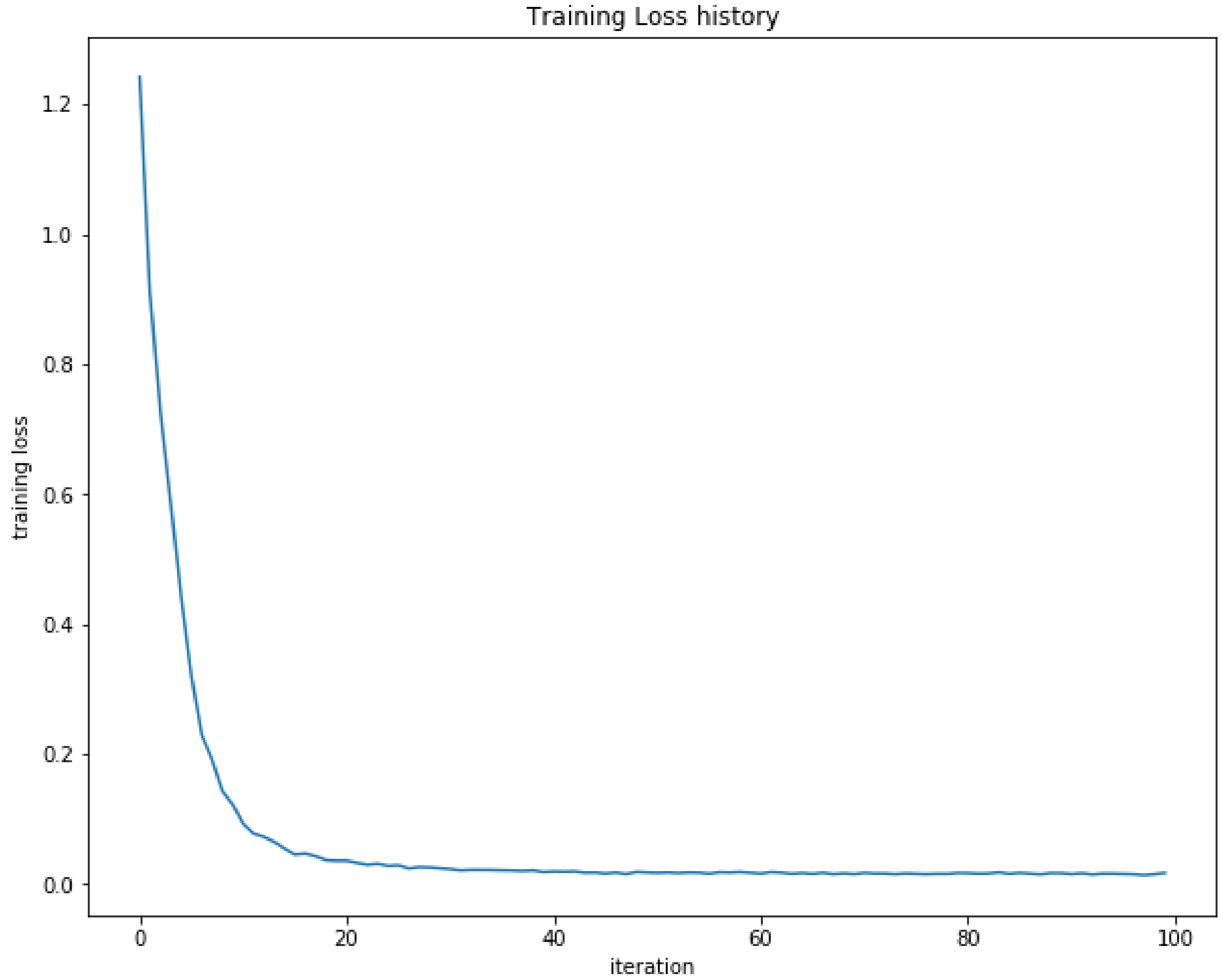
Once you have implemented the method, run the code below to train a two-layer network on toy data. You should achieve a training loss less than 0.2.

You should achieve a training loss less than 0.2.

In [118]:

```
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:  0.017149607938732037
```



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 [119]:

```
from cs231n.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 prepar
e
    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 = 'cs231n/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 c
ause 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)
```

```
Clear previously loaded data.
Train data shape:  (49000, 3072)
Train labels shape:  (49000,)
Validation data shape:  (1000, 3072)
```

```
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 [120]:

```
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.

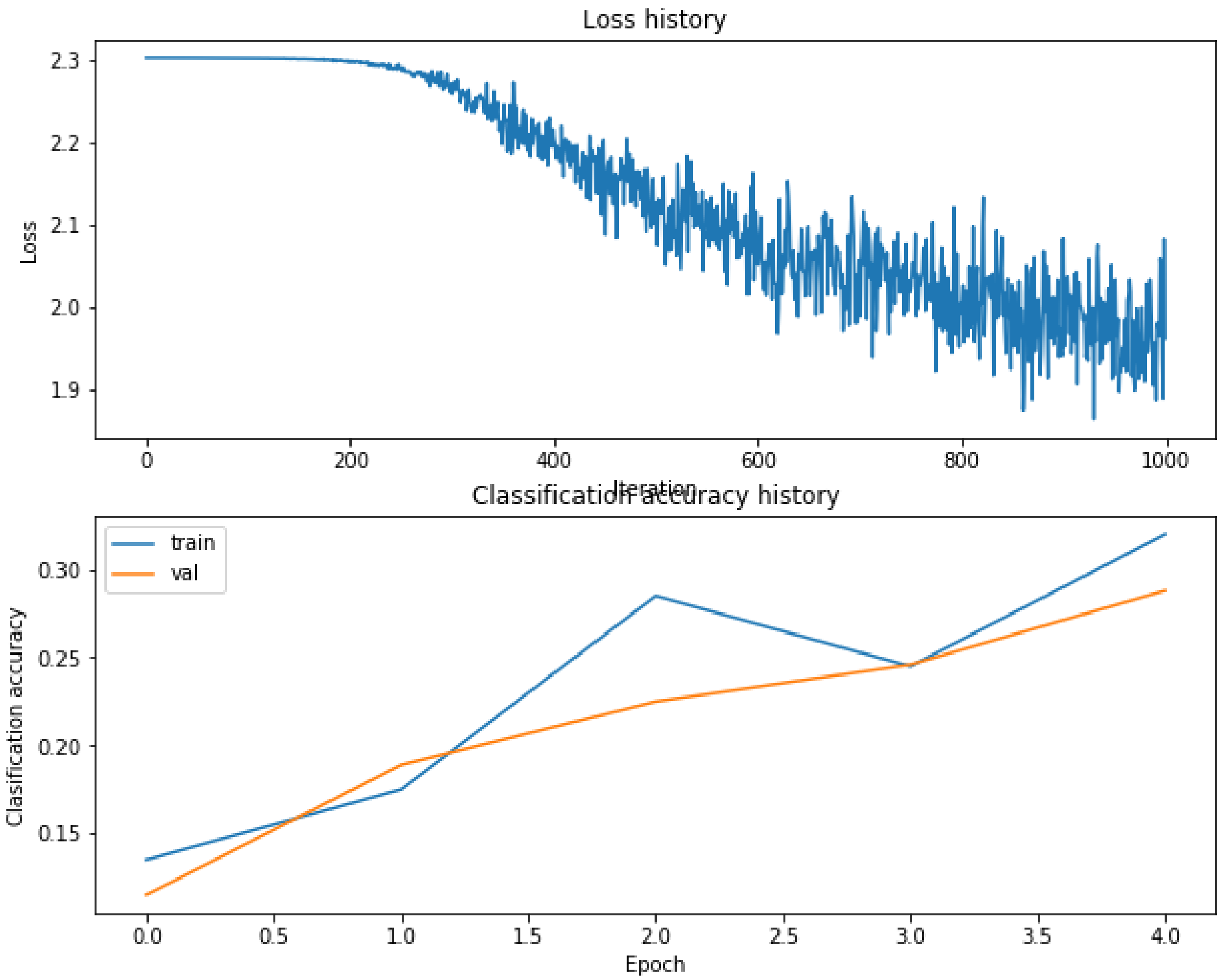
One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

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 [121]:

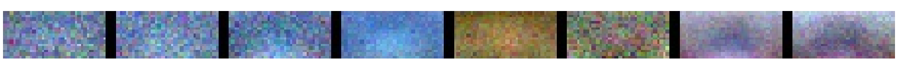
```
# Plot the loss function and train / validation accuracies
plt.subplot(2, 1, 1)
```

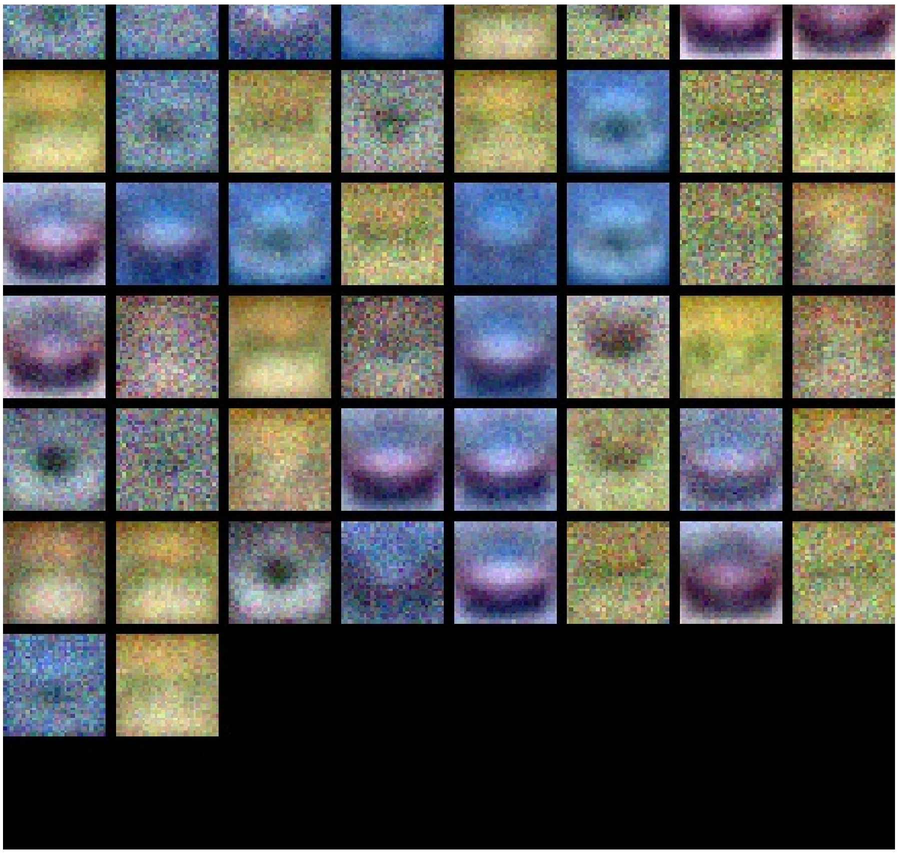
```
plt.plot(stats['loss_history'])
plt.title('Loss history')
plt.xlabel('Iteration')
plt.ylabel('Loss')

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 [122]:

```
from cs231n.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 [133]:

```
best_net = None # store the best model into this

#################################################################################
###
```

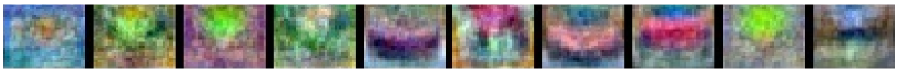
```
# TODO: Tune hyperparameters using the validation set. Store your best trai
ned  #
# model in best_net.
#
#
#
# To help debug your network, it may help to use visualizations similar to
the  #
# ones we used above; these visualizations will have significant qualitativ
e    #
# differences from the 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

learning_rates = [7e-4, 1e-3, 3e-3]

regularization_strengths = [5e-1, 3e-1, 1e-1]

learning_rate_decays = [0.95, 0.9, 0.8]

batch_sizes_and_epochs = [(50, 2000), (200, 1000), (500, 500)]

hidden_sizes = [60, 100, 140]

results = {}
best_val = -1

for lr in learning_rates:
    for reg in regularization_strengths:
        for decay in learning_rate_decays:
            for batch_size, num_iters in batch_sizes_and_epochs:
                for hidden_size in hidden_sizes:
                    net = TwoLayerNet(input_size, hidden_size, num_classes)
                    net.train(X_train, y_train, X_val, y_val, learning_rate=
lr, reg=reg, num_iters=num_iters, \
                              learning_rate_decay=decay,
batch_size=batch_size)
                    y_train_pred = net.predict(X_train)
                    acc_train = np.mean(y_train_pred == y_train)
                    y_val_pred = net.predict(X_val)
                    acc_val = np.mean(y_val_pred == y_val)
                    results[(lr, reg, decay, batch_size, num_iters, hidden_s
ize)] = (acc_train, acc_val)

                    if acc_val > best_val:
                        best_val = acc_val
                        best_net = net

for lr, reg, decay, batch_size, num_iters, hidden_size in sorted(results):
```

```
    train_accuracy, val_accuracy = results[(lr, reg, decay, batch_size,
num_iters, hidden_size)]
    print('lr %e reg %e decay %e batch size %e num_iters %e hidden size %e
train accuracy: %f val accuracy: %f' % (
                lr, reg, decay, batch_size, num_iters, hidden_size, train_ac
curacy, val_accuracy))

print('best validation accuracy achieved during cross-validation: %f' % bes
t_val)

#############################################################################
###
#                              END OF YOUR CODE
#
#############################################################################
###
```

```
0.473
0.485
0.391
lr 7.000000e-04 reg 3.000000e-01 decay 9.500000e-01 batch size 2.000000e+02
num_iters 1.000000e+03 hidden size 1.000000e+02 train accuracy: 0.479163 va
l accuracy: 0.473000
lr 1.000000e-03 reg 3.000000e-01 decay 9.500000e-01 batch size 2.000000e+02
num_iters 1.000000e+03 hidden size 1.000000e+02 train accuracy: 0.499041 va
l accuracy: 0.485000
lr 3.000000e-03 reg 3.000000e-01 decay 9.500000e-01 batch size 2.000000e+02
num_iters 1.000000e+03 hidden size 1.000000e+02 train accuracy: 0.426959 va
l accuracy: 0.391000
best validation accuracy achieved during cross-validation: 0.485000
```

In [134]:

```
# 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 [135]:

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

```
Test accuracy:  0.484
```

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 + 2 + 3

Your explanation:

1. Train on a larger dataset, because larger datasets contain more datapoints, and the increased likelihood of variety of datapoints may allow better generalizability.
2. Add more hidden units, because it will give the model more complexity which may be necessary to generalize to the test set.
3. Increase the regularization strength, because it will reduce overfitting, which is likely occuring since the training accuracy is relaitively high and test accuracy is relatively low.

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]() 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 [2]:

```
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

from __future__ import print_function

%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-i
python
%load_ext autoreload
%autoreload 2
```

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

Load data

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

In [14]:

```
from cs231n.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 = 'cs231n/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 c
ause 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()
```

```
Clear previously loaded 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 [15]:

```
from cs231n.features import *

num_color_bins = 10 # 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 featu
```

```
# Preprocessing: Divide by standard deviation. This ensures that each featu
re
# 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 [31]:

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

from cs231n.classifiers.linear_classifier import LinearSVM

learning_rates = [1e-9, 1e-8, 1e-7, 1e-6, 1e-5, 1e-4, 1e-3, 1e-2]
regularization_strengths = [5e4, 5e5, 5e6, 1e4, 1e5, 1e3]

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.
#
###########################################################################
##
num_iters = 1000

for lr in learning_rates:
    for reg in regularization_strengths:
        svm = LinearSVM()
        svm.train(X_train_feats, y_train, learning_rate=lr, reg=reg, num_ite
rs=num_iters)
        y_train_pred = svm.predict(X_train_feats)
        acc_train = np.mean(y_train_pred == y_train)
        y_val_pred = svm.predict(X_val_feats)
        acc_val = np.mean(y_val_pred == y_val)
        results[(lr, reg)] = (acc_train, acc_val)

        if acc_val > best_val:
            best_val = acc_val
            best_svm = svm

###########################################################################
##
#                               END OF YOUR CODE
#
```

```
#
################################################################################
##

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

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

```
/Users/ianscottknight/Google Drive/current
school/cs231n/assignment1/cs231n/classifiers/linear_svm.py:83:
RuntimeWarning: overflow encountered in double_scalars
  loss += 0.5 * reg * np.sum(W * W)
/Users/ianscottknight/anaconda/envs/cs231n/lib/python3.6/site-
packages/numpy/core/_methods.py:32: RuntimeWarning: overflow encountered in
reduce
  return umr_sum(a, axis, dtype, out, keepdims)
/Users/ianscottknight/Google Drive/current
school/cs231n/assignment1/cs231n/classifiers/linear_svm.py:83:
RuntimeWarning: overflow encountered in multiply
  loss += 0.5 * reg * np.sum(W * W)
/Users/ianscottknight/Google Drive/current
school/cs231n/assignment1/cs231n/classifiers/linear_svm.py:105: RuntimeWarn
ing: overflow encountered in multiply
  dW += reg*W
/Users/ianscottknight/Google Drive/current
school/cs231n/assignment1/cs231n/classifiers/linear_svm.py:79:
RuntimeWarning: invalid value encountered in subtract
  margin = np.maximum(0, scores - correct_class_score[:, np.newaxis] + delt
a)
/Users/ianscottknight/Google Drive/current
school/cs231n/assignment1/cs231n/classifiers/linear_svm.py:79:
RuntimeWarning: invalid value encountered in maximum
  margin = np.maximum(0, scores - correct_class_score[:, np.newaxis] + delt
a)
/Users/ianscottknight/Google Drive/current
school/cs231n/assignment1/cs231n/classifiers/linear_classifier.py:70: Runti
meWarning: invalid value encountered in add
  self.W += -learning_rate*grad
/Users/ianscottknight/Google Drive/current
school/cs231n/assignment1/cs231n/classifiers/linear_svm.py:99:
RuntimeWarning: invalid value encountered in greater
  X_mask[margin > 0] = 1
```

```
lr 1.000000e-09 reg 1.000000e+03 train accuracy: 0.091224 val accuracy: 0.0
83000
lr 1.000000e-09 reg 1.000000e+04 train accuracy: 0.081551 val accuracy: 0.0
83000
lr 1.000000e-09 reg 5.000000e+04 train accuracy: 0.081122 val accuracy: 0.0
77000
lr 1.000000e-09 reg 1.000000e+05 train accuracy: 0.073204 val accuracy: 0.0
74000
lr 1.000000e-09 reg 5.000000e+05 train accuracy: 0.110429 val accuracy: 0.1
14000
lr 1.000000e-09 reg 5.000000e+06 train accuracy: 0.099143 val accuracy: 0.0
97000
```

```
lr 1.000000e-08 reg 1.000000e+03 train accuracy: 0.089673 val accuracy: 0.1
01000
lr 1.000000e-08 reg 1.000000e+04 train accuracy: 0.103408 val accuracy: 0.1
11000
lr 1.000000e-08 reg 5.000000e+04 train accuracy: 0.108102 val accuracy: 0.1
00000
lr 1.000000e-08 reg 1.000000e+05 train accuracy: 0.095959 val accuracy: 0.0
99000
lr 1.000000e-08 reg 5.000000e+05 train accuracy: 0.146082 val accuracy: 0.1
51000
lr 1.000000e-08 reg 5.000000e+06 train accuracy: 0.402714 val accuracy: 0.3
95000
lr 1.000000e-07 reg 1.000000e+03 train accuracy: 0.129122 val accuracy: 0.1
29000
lr 1.000000e-07 reg 1.000000e+04 train accuracy: 0.126755 val accuracy: 0.1
20000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.321571 val accuracy: 0.3
43000
lr 1.000000e-07 reg 1.000000e+05 train accuracy: 0.412367 val accuracy: 0.4
10000
lr 1.000000e-07 reg 5.000000e+05 train accuracy: 0.410327 val accuracy: 0.4
21000
lr 1.000000e-07 reg 5.000000e+06 train accuracy: 0.356122 val accuracy: 0.3
69000
lr 1.000000e-06 reg 1.000000e+03 train accuracy: 0.271694 val accuracy: 0.2
62000
lr 1.000000e-06 reg 1.000000e+04 train accuracy: 0.414367 val accuracy: 0.4
26000
lr 1.000000e-06 reg 5.000000e+04 train accuracy: 0.410531 val accuracy: 0.4
05000
lr 1.000000e-06 reg 1.000000e+05 train accuracy: 0.395918 val accuracy: 0.3
83000
lr 1.000000e-06 reg 5.000000e+05 train accuracy: 0.373102 val accuracy: 0.3
71000
lr 1.000000e-06 reg 5.000000e+06 train accuracy: 0.100265 val accuracy: 0.0
87000
lr 1.000000e-05 reg 1.000000e+03 train accuracy: 0.416061 val accuracy: 0.4
11000
lr 1.000000e-05 reg 1.000000e+04 train accuracy: 0.405959 val accuracy: 0.4
11000
lr 1.000000e-05 reg 5.000000e+04 train accuracy: 0.373612 val accuracy: 0.3
67000
lr 1.000000e-05 reg 1.000000e+05 train accuracy: 0.320776 val accuracy: 0.3
28000
lr 1.000000e-05 reg 5.000000e+05 train accuracy: 0.100265 val accuracy: 0.0
87000
lr 1.000000e-05 reg 5.000000e+06 train accuracy: 0.100265 val accuracy: 0.0
87000
lr 1.000000e-04 reg 1.000000e+03 train accuracy: 0.411837 val accuracy: 0.4
42000
lr 1.000000e-04 reg 1.000000e+04 train accuracy: 0.345000 val accuracy: 0.3
53000
lr 1.000000e-04 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.0
87000
lr 1.000000e-04 reg 1.000000e+05 train accuracy: 0.100265 val accuracy: 0.0
87000
lr 1.000000e-04 reg 5.000000e+05 train accuracy: 0.100265 val accuracy: 0.0
87000
lr 1.000000e-04 reg 5.000000e+06 train accuracy: 0.100265 val accuracy: 0.0
87000
lr 1.000000e-03 reg 1.000000e+03 train accuracy: 0.323286 val accuracy: 0.3
```

```
23000
lr 1.000000e-03 reg 1.000000e+04 train accuracy: 0.100265 val accuracy: 0.0
87000
lr 1.000000e-03 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.0
87000
lr 1.000000e-03 reg 1.000000e+05 train accuracy: 0.100265 val accuracy: 0.0
87000
lr 1.000000e-03 reg 5.000000e+05 train accuracy: 0.100265 val accuracy: 0.0
87000
lr 1.000000e-03 reg 5.000000e+06 train accuracy: 0.100265 val accuracy: 0.0
87000
lr 1.000000e-02 reg 1.000000e+03 train accuracy: 0.100265 val accuracy: 0.0
87000
lr 1.000000e-02 reg 1.000000e+04 train accuracy: 0.100265 val accuracy: 0.0
87000
lr 1.000000e-02 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.0
87000
lr 1.000000e-02 reg 1.000000e+05 train accuracy: 0.100265 val accuracy: 0.0
87000
lr 1.000000e-02 reg 5.000000e+05 train accuracy: 0.100265 val accuracy: 0.0
87000
lr 1.000000e-02 reg 5.000000e+06 train accuracy: 0.100265 val accuracy: 0.0
87000
best validation accuracy achieved during cross-validation: 0.442000
```

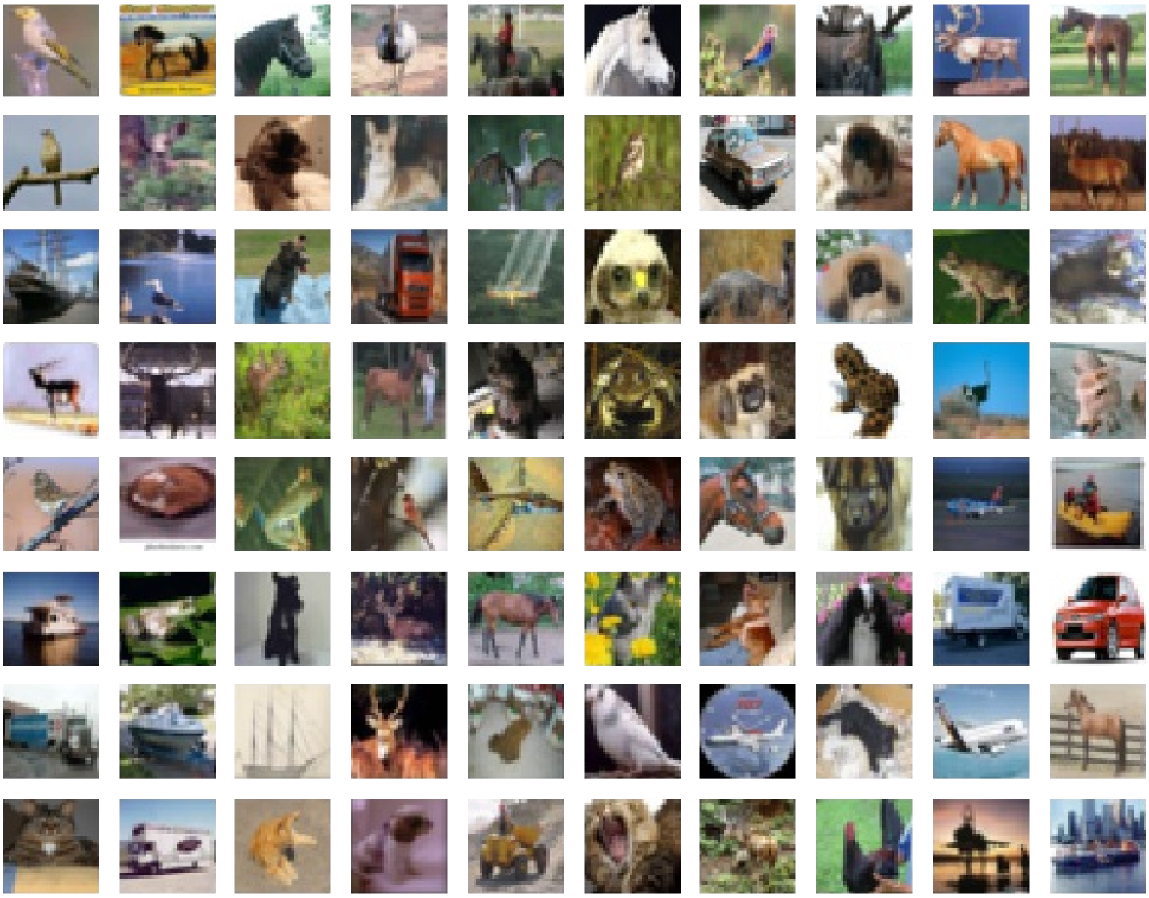
In [33]:

```python
# 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)
```

```
0.419
```

In [34]:

```python
# An important way to gain intuition about how an algorithm works is to
# visualize the mistakes that it makes. In this visualization, we show exam
ples
# 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', 's
hip', '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?

Some of the misclassification results make sense. For example, when the predicted class is "ship" and the true class is not "ship", it is often the case that the test image has blue surroundings with a central subject, as we would expect to find when the image is of a ship. Next, for cases where the predicted class is "plane" and the true class is not "plane", the sky is nearly always prominent with some elongated object as the subject (which is similar to the image of a plane in the sky).

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 [35]:

```
# 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, 155)
(49000, 154)
```

In [53]:

```
from cs231n.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 bes
t   #
# model in the best_net variable.
#
##############################################################################
##
input_size = 32 * 32 * 3
num_classes = 10

learning_rates = [1e0]
regularization_strengths = [1e-6]
batch_sizes_and_epochs = [(50, 2000), (200, 1000), (500, 500)]
hidden_sizes = [100, 200, 500, 800]

results = {}
best_val = -1

for lr in learning_rates:
    for reg in regularization_strengths:
        for batch_size, num_iters in batch_sizes_and_epochs:
            for hidden_size in hidden_sizes:
                net = TwoLayerNet(input_dim, hidden_size, num_classes)
                net.train(X_train_feats, y_train, X_val_feats, y_val, learni
ng_rate=lr, reg=reg, num_iters=num_iters, \
                          learning_rate_decay=decay, batch_size=batch_size)
                y_train_pred = net.predict(X_train_feats)
                acc_train = np.mean(y_train_pred == y_train)
                y_val_pred = net.predict(X_val_feats)
                acc_val = np.mean(y_val_pred == y_val)
                results[(lr, reg, decay, batch_size, num_iters, hidden_size)
] = (acc_train, acc_val)

                if acc_val > best_val:
                    best_val = acc_val
                    best_net = net

for lr, reg, decay, batch_size, num_iters, hidden_size in sorted(results):
    train_accuracy, val_accuracy = results[(lr, reg, decay, batch_size,
num_iters, hidden_size)]
```

```
    print('lr %e reg %e decay %e batch size %e num_iters %e hidden size %e
train accuracy: %f val accuracy: %f' % (
                lr, reg, decay, batch_size, num_iters, hidden_size, train_ac
curacy, val_accuracy))

print('best validation accuracy achieved during cross-validation: %f' % bes
t_val)
###########################################################################
##
#                              END OF YOUR CODE
#
###########################################################################
##
```

```
0.575
0.573
0.591
0.599
lr 1.000000e+00 reg 1.000000e-06 decay 8.000000e-01 batch size 2.000000e+02
num_iters 1.000000e+03 hidden size 1.000000e+02 train accuracy: 0.631265 va
l accuracy: 0.575000
lr 1.000000e+00 reg 1.000000e-06 decay 8.000000e-01 batch size 2.000000e+02
num_iters 1.000000e+03 hidden size 2.000000e+02 train accuracy: 0.653776 va
l accuracy: 0.573000
lr 1.000000e+00 reg 1.000000e-06 decay 8.000000e-01 batch size 2.000000e+02
num_iters 1.000000e+03 hidden size 5.000000e+02 train accuracy: 0.680408 va
l accuracy: 0.591000
lr 1.000000e+00 reg 1.000000e-06 decay 8.000000e-01 batch size 2.000000e+02
num_iters 1.000000e+03 hidden size 8.000000e+02 train accuracy: 0.686551 va
l accuracy: 0.599000
best validation accuracy achieved during cross-validation: 0.599000
```

In [54]:

```
# 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)
```

```
0.58
```