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 (http://vision.stanford.edu/teaching/cs231n/assignments.html)](http://vision.stanford.edu/teaching/cs231n/assignments.html) 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

# This is a bit of magic to make matplotlib figures appear inline in t
he notebook
# rather than in a new window.
%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plo
ts
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

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

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

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

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

# As a sanity check, we print out the size of the training and test da
ta.
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', 'hors
e', 'ship', 'truck']
num_classes = len(classes)
samples_per_class = 7
for y, cls in enumerate(classes):
    idxs = np.flatnonzero(y_train == y)
    idxs = np.random.choice(idxs, samples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt_idx = i * num_classes + y + 1
        plt.subplot(samples_per_class, num_classes, plt_idx)
        plt.imshow(X_train[idx].astype('uint8'))
        plt.axis('off')
        if i == 0:
            plt.title(cls)
plt.show()
```

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

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

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

```
(5000, 3072) (500, 3072)
```

In [5]:
```
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 process
ing
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.

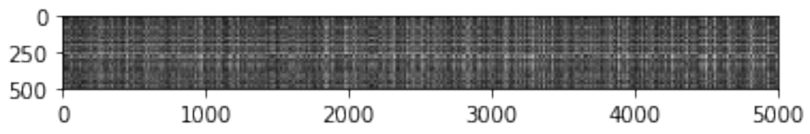
Note: For the three distance computations that we require you to implement in this notebook, you may not use the np.linalg.norm() function that numpy provides.

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 [6]:
```
# 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 [7]:
```
# We can visualize the distance matrix: each row is a single test exam
ple 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 white indicates high distances meaning that the image in question is far from the rest of the examples in that set. When we have more similarity we will have insight into the similarity features of the image rather than the general outline of the features of the image. The above is a good indication to visually understand how our dataset is comparing to each other.*

In [8]:
```
# Now implement the function predict_labels and run the code below:
# We use k = 1 (which is Nearest Neighbor).
y_test_pred = classifier.predict_labels(dists, k=1)

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

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

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

In [9]:
```
y_test_pred = classifier.predict_labels(dists, k=5)
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test,
accuracy))
```

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

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

Inline Question 2

We can also use other distance metrics such as L1 distance. For pixel values $p_{ij}^{(k)}$ at location (i, j) of some image I_k,

the mean μ across all pixels over all images is

$$\mu = \frac{1}{nhw} \sum_{k=1}^{n} \sum_{i=1}^{h} \sum_{j=1}^{w} p_{ij}^{(k)}$$

And the pixel-wise mean μ_{ij} across all images is

$$\mu_{ij} = \frac{1}{n} \sum_{k=1}^{n} p_{ij}^{(k)}.$$

The general standard deviation σ and pixel-wise standard deviation σ_{ij} is defined similarly.

Which of the following preprocessing steps will not change the performance of a Nearest Neighbor classifier that uses L1 distance? Select all that apply.

1. Subtracting the mean μ ($\tilde{p}_{ij}^{(k)} = p_{ij}^{(k)} - \mu$.)
2. Subtracting the per pixel mean μ_{ij} ($\tilde{p}_{ij}^{(k)} = p_{ij}^{(k)} - \mu_{ij}$.)
3. Subtracting the mean μ and dividing by the standard deviation σ.
4. Subtracting the pixel-wise mean μ_{ij} and dividing by the pixel-wise standard deviation σ_{ij}.
5. Rotating the coordinate axes of the data.

$\color{blue}{\textit{Your Answer :}}$ 1, 2 and 3.

$\color{blue}{\textit{Your Explanation :}}$ For the first two, we are only scaling it a bit, but the deduction does not significantly alter the calculations. With the third option, similarly, we are scaling by sigma, but the overall distandce will be the same, scaled, and therefore the classifier will perform the same.

In [10]:
```
# Now lets speed up distance matrix computation by using partial vecto
rization
# 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 sur
e that it
# agrees with the naive implementation. There are many ways to decide
whether
# two matrices are similar; one of the simplest is the Frobenius norm.
In case
# you haven't seen it before, the Frobenius norm of two matrices is th
e square
# root of the squared sum of differences of all elements; in other wor
ds, reshape
# the matrices into vectors and compute the Euclidean distance between
them.
difference = np.linalg.norm(dists - dists_one, ord='fro')
print('One loop 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')
```

```
One loop difference was: 0.000000
Good! The distance matrices are the same
```

In [11]:
```
# 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 befor
e:
difference = np.linalg.norm(dists - dists_two, ord='fro')
print('No loop 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')
```

```
No loop difference was: 0.000000
Good! The distance matrices are the same
```

In [12]:
```
# Let's compare how fast the implementations are
def time_function(f, *args):
    """
    Call a function f with args and return the time (in seconds) that
it took to execute.
    """
    import time
    tic = time.time()
    f(*args)
    toc = time.time()
    return toc - tic

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

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

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

# You should see significantly faster performance with the fully vecto
rized implementation!

# NOTE: depending on what machine you're using,
# you might not see a speedup when you go from two loops to one loop,
# and might even see a slow-down.
```

```
Two loop version took 26.573888 seconds
One loop version took 35.068864 seconds
No loop version took 0.140703 seconds
```

Cross-validation

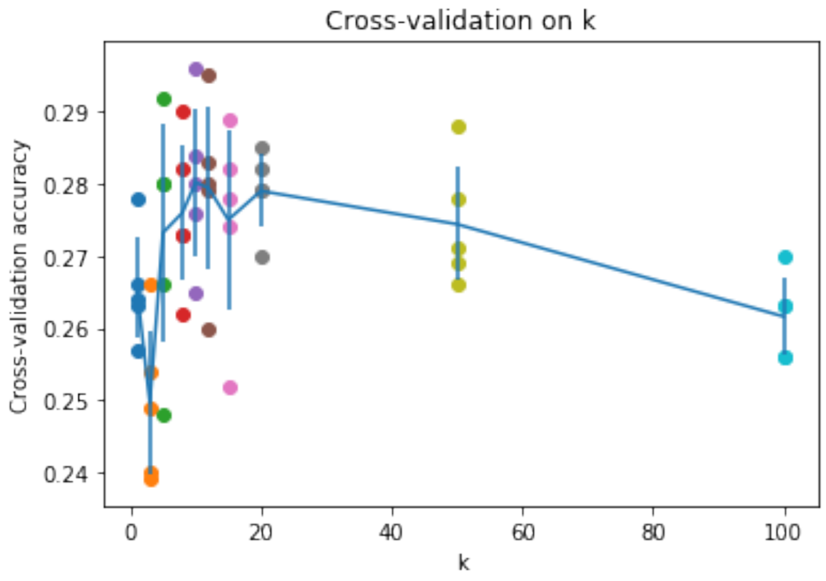
We have implemented the k-Nearest Neighbor classifier but we set the value k = 5 arbitrarily. We will now determine the best value of this hyperparameter with cross-validation.

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_fold
```

```
s 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.
#
#######################################################################
##########
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

X_train_folds = np.array_split(X_train, num_folds)
Y_train_folds = np.array_split(y_train, num_folds)

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

# A dictionary holding the accuracies for different values of k that w
e find
# when running cross-validation. After running cross-validation,
# k_to_accuracies[k] should be a list of length num_folds giving the d
ifferent
# 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 eac
h        #
# 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.
#
#######################################################################
##########
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

for k in k_choices:
    accuracies = []
    for i in range(num_folds):
        ##our dataset##
        training_data = np.concatenate(X_train_folds[0:i] + X_train_fo
lds[i + 1:])
        training_labels = np.concatenate(Y_train_folds[0:i] + Y_train_
folds[i + 1:])
        data_for_validation = X_train_folds[i]
        labels_for_validation = Y_train_folds[i]
        ##our model evaluation##
        classifier.train(training_data, training_labels) #model listed
above, already specified earlier in script
        prediction = classifier.predict(data_for_validation, k = k)
        accuracy = float(np.sum(prediction == labels_for_validation))
/ prediction.shape[0]
        accuracies.append(accuracy)

    #adding accuracies back to accuracy
    k_to_accuracies[k] = accuracies

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

# 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 = 0.263000
k = 1, accuracy = 0.257000
k = 1, accuracy = 0.264000
k = 1, accuracy = 0.278000
k = 1, accuracy = 0.266000
k = 3, accuracy = 0.239000
k = 3, accuracy = 0.249000
k = 3, accuracy = 0.240000
k = 3, accuracy = 0.266000
k = 3, accuracy = 0.254000
k = 5, accuracy = 0.248000
k = 5, accuracy = 0.266000
k = 5, accuracy = 0.280000
k = 5, accuracy = 0.292000
k = 5, accuracy = 0.280000
k = 8, accuracy = 0.262000
k = 8, accuracy = 0.282000
k = 8, accuracy = 0.273000
k = 8, accuracy = 0.290000
k = 8, accuracy = 0.273000
k = 10, accuracy = 0.265000
k = 10, accuracy = 0.296000
k = 10, accuracy = 0.276000
k = 10, accuracy = 0.284000
k = 10, accuracy = 0.280000
k = 12, accuracy = 0.260000
k = 12, accuracy = 0.295000
k = 12, accuracy = 0.279000
k = 12, accuracy = 0.283000
k = 12, accuracy = 0.280000
k = 15, accuracy = 0.252000
k = 15, accuracy = 0.289000
k = 15, accuracy = 0.278000
k = 15, accuracy = 0.282000
k = 15, accuracy = 0.274000
k = 20, accuracy = 0.270000
k = 20, accuracy = 0.279000
k = 20, accuracy = 0.279000
k = 20, accuracy = 0.282000
k = 20, accuracy = 0.285000
k = 50, accuracy = 0.271000
k = 50, accuracy = 0.288000
k = 50, accuracy = 0.278000
k = 50, accuracy = 0.269000
k = 50, accuracy = 0.266000
k = 100, accuracy = 0.256000
k = 100, accuracy = 0.270000
k = 100, accuracy = 0.263000
k = 100, accuracy = 0.256000
k = 100, accuracy = 0.263000
```

In [15]:
```
# plot the raw observations
for k in k_choices:
    accuracies = k_to_accuracies[k]
    plt.scatter([k] * len(accuracies), accuracies)

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



In [23]:
```
# Based on the cross-validation results above, choose the best value f
or k,
# retrain the classifier using all the training data, and test it on t
he test
# data. You should be able to get above 28% accuracy on the test data.
best_k = 10 #best accuracy with 10, numbers around it did not do so we
ll.

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

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

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

Inline Question 3

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

1. The decision boundary of the k-NN classifier is linear.
2. The training error of a 1-NN will always be lower than that of 5-NN.
3. The test error of a 1-NN will always be lower than that of a 5-NN.
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.

$Your Answer:$ 2 and 4

$Your Explanation:$ 1 KK will have a lower training value than 5 because with 1, it trains on itself. The last one is true because as we increase the training set, we increase the time needed to run the calculations, as we need to calculate more distances between points.

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 (http://vision.stanford.edu/teaching/cs231n/assignments.html)](http://vision.stanford.edu/teaching/cs231n/assignments.html) 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 [134]:
```
# 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

# This is a bit of magic to make matplotlib figures appear inline in t
he
# notebook rather than in a new window.
%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plo
ts
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

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

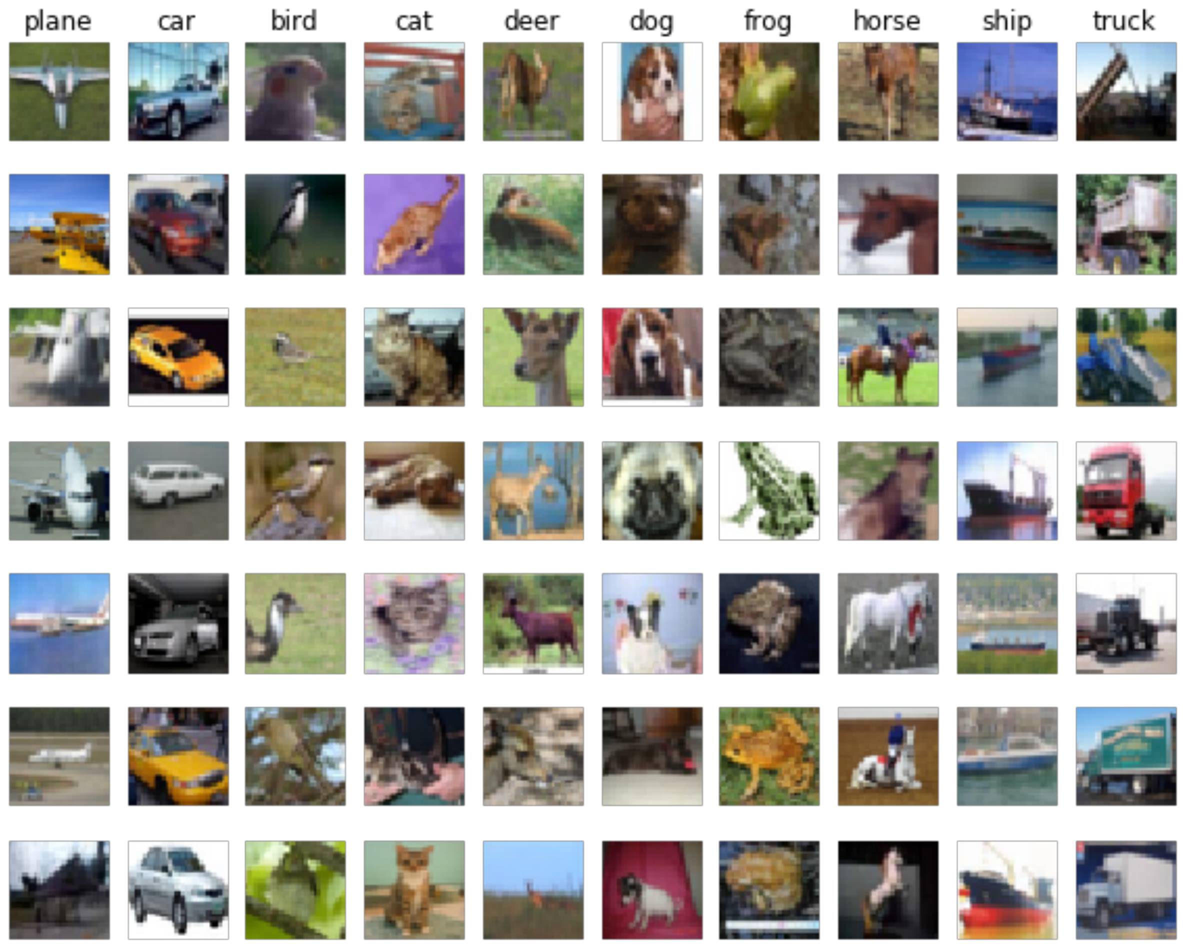
```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

CIFAR-10 Data Loading and Preprocessing

In [135]:
```
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

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

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

# As a sanity check, we print out the size of the training and test da
ta.
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 [136]:
```
# 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', 'hors
e', 'ship', 'truck']
num_classes = len(classes)
samples_per_class = 7
for y, cls in enumerate(classes):
    idxs = np.flatnonzero(y_train == y)
    idxs = np.random.choice(idxs, samples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt_idx = i * num_classes + y + 1
        plt.subplot(samples_per_class, num_classes, plt_idx)
        plt.imshow(X_train[idx].astype('uint8'))
        plt.axis('off')
        if i == 0:
            plt.title(cls)
plt.show()
```


plane
car
bird
cat
deer
dog
frog
horse
ship
truck

In [137]:
```
# 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 origina
l
# 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)
```

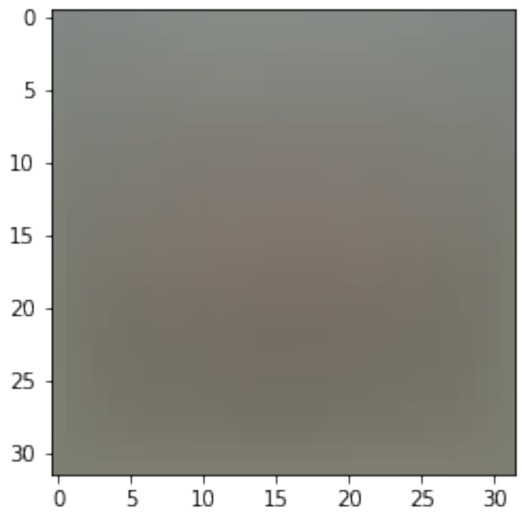
```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

In [138]:
```
# 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 [139]:
```
# Preprocessing: subtract the mean image
# first: compute the image mean based on the training data
mean_image = np.mean(X_train, axis=0)
print(mean_image[:10]) # print a few of the elements
plt.figure(figsize=(4,4))
plt.imshow(mean_image.reshape((32,32,3)).astype('uint8')) # visualize
the mean image
plt.show()

# 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

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

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

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



```
(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 `svm_loss_naive` which uses for loops to evaluate the multiclass SVM loss function.

In [140]:
```python
# 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: 9.556129
```

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 [141]:
```
# Once you've implemented the gradient, recompute it with the code bel
ow
# 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 dimen
sions, and
# compare them with your analytically computed gradient. The numbers s
hould 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: 1.398617 analytic: 0.000000, relative error: 1.000000e+00
numerical: 6.894196 analytic: 0.000000, relative error: 1.000000e+00
numerical: -36.241177 analytic: 0.000000, relative error: 1.000000e+
00
numerical: 12.263645 analytic: 0.000000, relative error: 1.000000e+0
0
numerical: -8.174303 analytic: 0.000000, relative error: 1.000000e+0
0
numerical: 27.384101 analytic: 0.000000, relative error: 1.000000e+0
0
numerical: -29.206113 analytic: 0.000000, relative error: 1.000000e+
00
numerical: -0.072241 analytic: 0.000000, relative error: 1.000000e+0
0
numerical: 8.244012 analytic: 0.000000, relative error: 1.000000e+00
numerical: 10.005147 analytic: 0.000000, relative error: 1.000000e+0
0
numerical: 0.506922 analytic: 0.005874, relative error: 9.770913e-01
numerical: -1.811488 analytic: 0.005121, relative error: 1.000000e+0
0
numerical: 42.397012 analytic: 0.001566, relative error: 9.999262e-0
1
numerical: 32.493101 analytic: -0.018471, relative error: 1.000000e+
00
numerical: -3.961386 analytic: 0.006077, relative error: 1.000000e+0
0
numerical: -2.258468 analytic: 0.002197, relative error: 1.000000e+0
0
numerical: 5.272086 analytic: -0.022157, relative error: 1.000000e+0
0
numerical: -0.514169 analytic: 0.005980, relative error: 1.000000e+0
0
numerical: -9.784324 analytic: 0.006605, relative error: 1.000000e+0
0
numerical: -11.236171 analytic: 0.002423, relative error: 1.000000e+
00
```

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*

$Your Answer:$ Not entirely sure what the question is asking. If we are referring to when there are discrepencies with the loss function, then yes, it could be that the loss function does not fully differentiate which causes the gradcheck to not match exactly.

In [142]:
```
# Next implement the function svm_loss_vectorized; for now only comput
e 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: 9.556129e+00 computed in 0.014977s
Vectorized loss: 9.556129e+00 computed in 0.002795s
difference: 0.000000
```

In [148]:
```
# Complete the implementation of svm_loss_vectorized, and compute the
gradient
# of the loss function in a vectorized way.

# The naive implementation and the vectorized implementation should ma
tch, 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 com
puted
# by the two implementations. The gradient on the other hand is a matr
ix, 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 0.013766s
Vectorized loss and gradient: computed in 0.000287s
difference: 0.000000
```

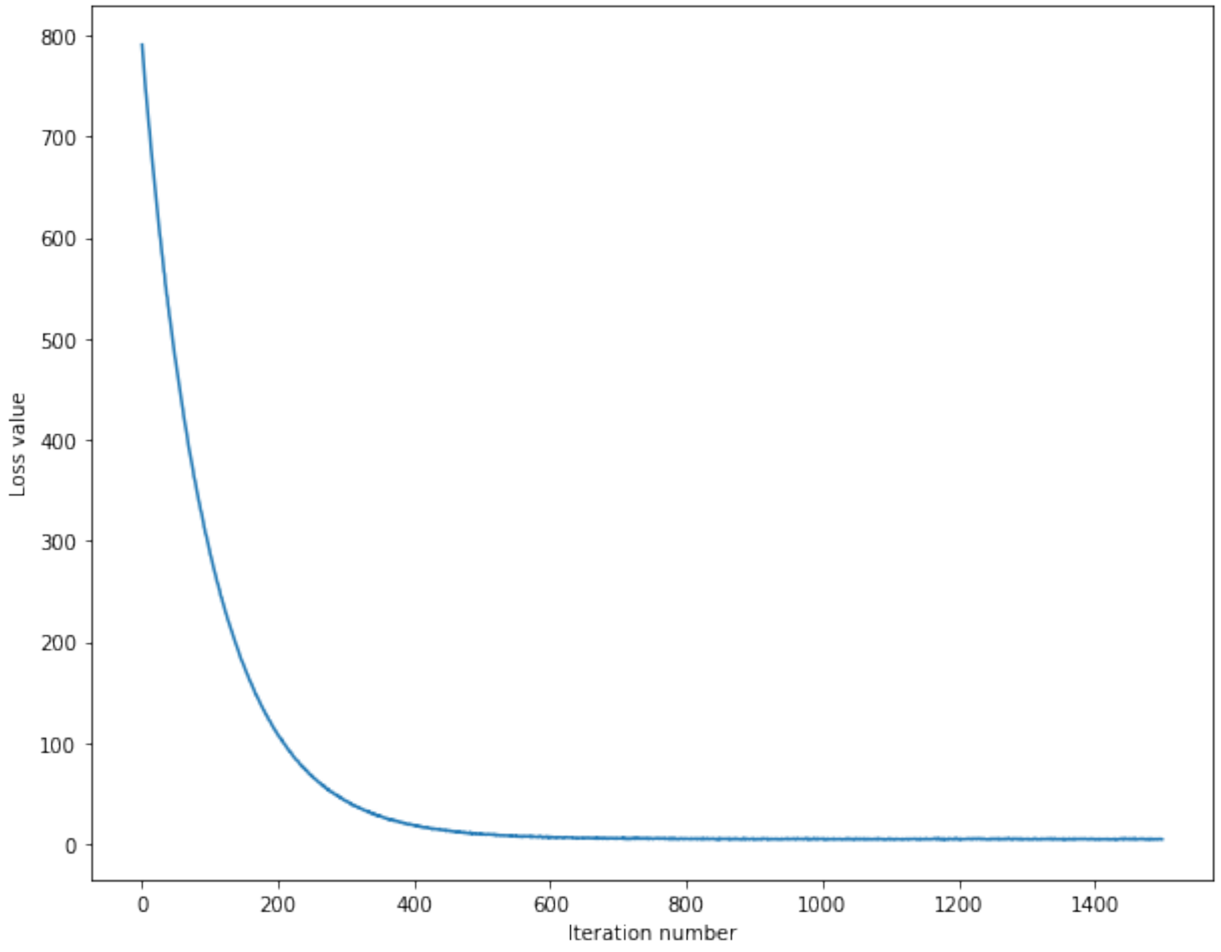
Stochastic Gradient Descent

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. Your code for this part will be written inside `cs231n/classifiers/linear_classifier.py` .

In [144]:
```
# 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 790.718095
iteration 100 / 1500: loss 286.693907
iteration 200 / 1500: loss 107.788907
iteration 300 / 1500: loss 43.050307
iteration 400 / 1500: loss 18.666922
iteration 500 / 1500: loss 9.911225
iteration 600 / 1500: loss 6.825496
iteration 700 / 1500: loss 6.095525
iteration 800 / 1500: loss 5.745249
iteration 900 / 1500: loss 4.891450
iteration 1000 / 1500: loss 5.554712
iteration 1100 / 1500: loss 5.305421
iteration 1200 / 1500: loss 5.030613
iteration 1300 / 1500: loss 5.282118
iteration 1400 / 1500: loss 5.450685
That took 3.111909s
```

In [145]:
```
# 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 [146]:
```
# 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: 0.371510
validation accuracy: 0.377000
```

In [130]:
```
# Use the validation set to tune hyperparameters (regularization stren
gth and
# learning rate). You should experiment with different ranges for the
learning
# rates and regularization strengths; if you are careful you should be
able to
# get a classification accuracy of about 0.39 on the validation set.
# Note: you may see runtime/overflow warnings during hyper-parameter s
earch.
# This may be caused by extreme values, and is not a bug.

# 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 valid
ation rate.

#######################################################################
##########
# TODO:
#
# Write code that chooses the best hyperparameters by tuning on the va
lidation #
# set. For each combination of hyperparameters, train a linear SVM on
the      #
# training set, compute its accuracy on the training and validation se
ts, and  #
# store these numbers in the results dictionary. In addition, store th
e best   #
# validation accuracy in best_val and the LinearSVM object that achiev
es 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 vali
dation   #
# code with a larger value for num_iters.
#
#######################################################################
##########

# Provided as a reference. You may or may not want to change these hyp
erparameters
learning_rates = [1e-7, 5e-5]
regularization_strengths = [2.5e4, 5e4]

# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

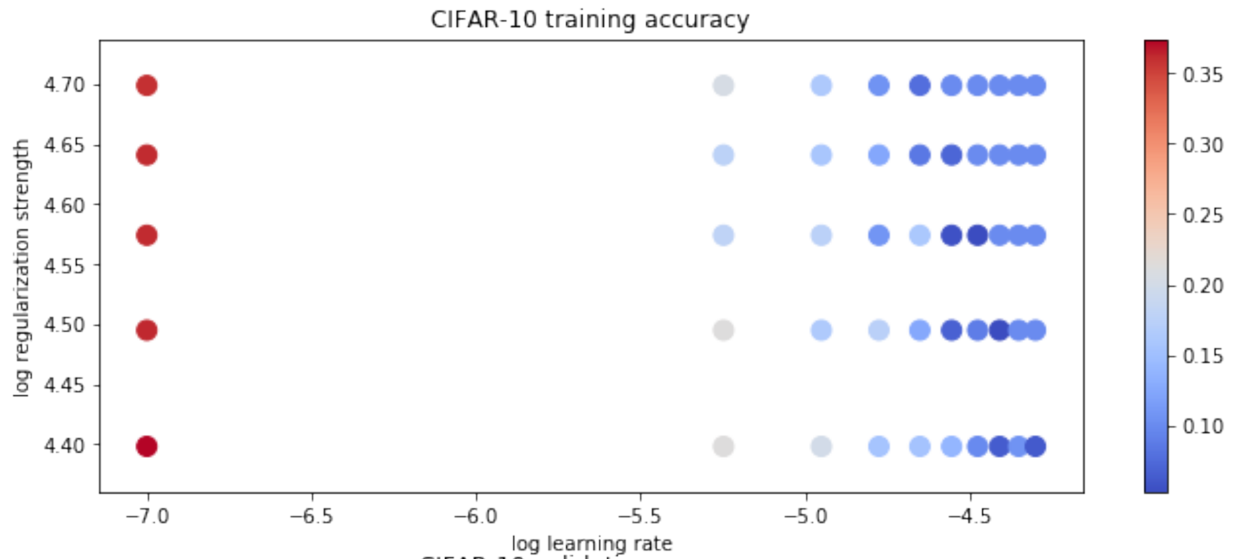
```
for lr in learning_rates:
    for reg in regularization_strengths:
        #linear SVM - updated above already
        loss_hist = svm.train(X_train, y_train, learning_rate=lr, reg=
reg, num_iters=10000,
                    verbose=True)
        #pull from above - y_train_pred = svm.predict(X_train)
        train_accuracy = np.mean(svm.predict(X_train) == y_train)
        #pull from above - y_val_pred = svm.predict(X_val)
        val_accuracy = np.mean(svm.predict(X_val) == y_val)

        if val_accuracy > best_val:
            best_val = val_accuracy
            best_svm = svm




# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

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

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

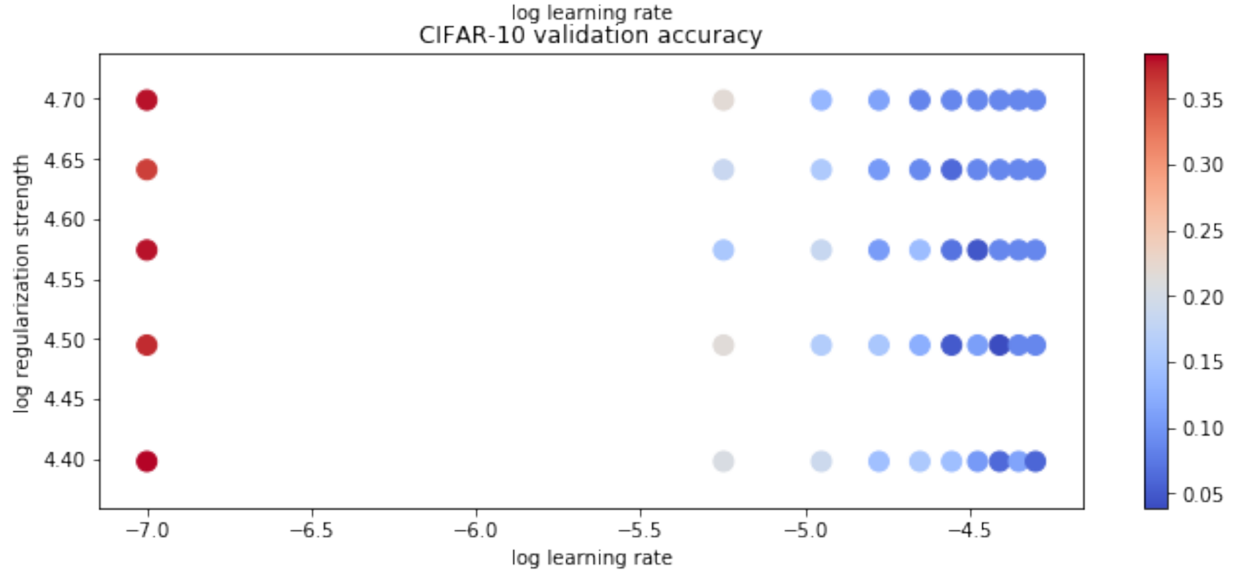
```
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.374000 val accura
cy: 0.384000
lr 1.000000e-07 reg 3.125000e+04 train accuracy: 0.363857 val accura
cy: 0.371000
lr 1.000000e-07 reg 3.750000e+04 train accuracy: 0.362408 val accura
cy: 0.379000
lr 1.000000e-07 reg 4.375000e+04 train accuracy: 0.362000 val accura
cy: 0.361000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.358959 val accura
cy: 0.379000
lr 5.644444e-06 reg 2.500000e+04 train accuracy: 0.213612 val accura
cy: 0.205000
lr 5.644444e-06 reg 3.125000e+04 train accuracy: 0.214143 val accura
cy: 0.216000
lr 5.644444e-06 reg 3.750000e+04 train accuracy: 0.179245 val accura
cy: 0.156000
lr 5.644444e-06 reg 4.375000e+04 train accuracy: 0.177041 val accura
cy: 0.186000
lr 5.644444e-06 reg 5.000000e+04 train accuracy: 0.204653 val accura
cy: 0.219000
lr 1.118889e-05 reg 2.500000e+04 train accuracy: 0.200265 val accura
cy: 0.190000
lr 1.118889e-05 reg 3.125000e+04 train accuracy: 0.165102 val accura
cy: 0.163000
lr 1.118889e-05 reg 3.750000e+04 train accuracy: 0.176143 val accura
cy: 0.185000
```

```
lr 1.118889e-05 reg 4.375000e+04 train accuracy: 0.158082 val accura
cy: 0.159000
lr 1.118889e-05 reg 5.000000e+04 train accuracy: 0.163878 val accura
cy: 0.134000
lr 1.673333e-05 reg 2.500000e+04 train accuracy: 0.155796 val accura
cy: 0.145000
lr 1.673333e-05 reg 3.125000e+04 train accuracy: 0.174714 val accura
cy: 0.154000
lr 1.673333e-05 reg 3.750000e+04 train accuracy: 0.108673 val accura
cy: 0.106000
lr 1.673333e-05 reg 4.375000e+04 train accuracy: 0.124633 val accura
cy: 0.104000
lr 1.673333e-05 reg 5.000000e+04 train accuracy: 0.106776 val accura
cy: 0.112000
lr 2.227778e-05 reg 2.500000e+04 train accuracy: 0.155184 val accura
cy: 0.157000
lr 2.227778e-05 reg 3.125000e+04 train accuracy: 0.126204 val accura
cy: 0.124000
lr 2.227778e-05 reg 3.750000e+04 train accuracy: 0.161061 val accura
cy: 0.141000
lr 2.227778e-05 reg 4.375000e+04 train accuracy: 0.085898 val accura
cy: 0.089000
lr 2.227778e-05 reg 5.000000e+04 train accuracy: 0.077653 val accura
cy: 0.085000
lr 2.782222e-05 reg 2.500000e+04 train accuracy: 0.140184 val accura
cy: 0.144000
lr 2.782222e-05 reg 3.125000e+04 train accuracy: 0.067490 val accura
cy: 0.051000
lr 2.782222e-05 reg 3.750000e+04 train accuracy: 0.056857 val accura
cy: 0.070000
lr 2.782222e-05 reg 4.375000e+04 train accuracy: 0.071082 val accura
cy: 0.062000
lr 2.782222e-05 reg 5.000000e+04 train accuracy: 0.100265 val accura
cy: 0.087000
lr 3.336667e-05 reg 2.500000e+04 train accuracy: 0.098959 val accura
cy: 0.105000
lr 3.336667e-05 reg 3.125000e+04 train accuracy: 0.088898 val accura
cy: 0.108000
lr 3.336667e-05 reg 3.750000e+04 train accuracy: 0.052367 val accura
cy: 0.048000
lr 3.336667e-05 reg 4.375000e+04 train accuracy: 0.100265 val accura
cy: 0.087000
lr 3.336667e-05 reg 5.000000e+04 train accuracy: 0.100265 val accura
cy: 0.087000
lr 3.891111e-05 reg 2.500000e+04 train accuracy: 0.064714 val accura
cy: 0.063000
lr 3.891111e-05 reg 3.125000e+04 train accuracy: 0.052224 val accura
cy: 0.039000
lr 3.891111e-05 reg 3.750000e+04 train accuracy: 0.100265 val accura
cy: 0.087000
lr 3.891111e-05 reg 4.375000e+04 train accuracy: 0.100265 val accura
cy: 0.087000
lr 3.891111e-05 reg 5.000000e+04 train accuracy: 0.100265 val accura
```

```
cy: 0.087000
lr 4.445556e-05 reg 2.500000e+04 train accuracy: 0.107408 val accura
cy: 0.114000
lr 4.445556e-05 reg 3.125000e+04 train accuracy: 0.100265 val accura
cy: 0.087000
lr 4.445556e-05 reg 3.750000e+04 train accuracy: 0.100265 val accura
cy: 0.087000
lr 4.445556e-05 reg 4.375000e+04 train accuracy: 0.100265 val accura
cy: 0.087000
lr 4.445556e-05 reg 5.000000e+04 train accuracy: 0.100265 val accura
cy: 0.087000
lr 5.000000e-05 reg 2.500000e+04 train accuracy: 0.063388 val accura
cy: 0.060000
lr 5.000000e-05 reg 3.125000e+04 train accuracy: 0.100265 val accura
cy: 0.087000
lr 5.000000e-05 reg 3.750000e+04 train accuracy: 0.100265 val accura
cy: 0.087000
lr 5.000000e-05 reg 4.375000e+04 train accuracy: 0.100265 val accura
cy: 0.087000
lr 5.000000e-05 reg 5.000000e+04 train accuracy: 0.100265 val accura
cy: 0.087000
best validation accuracy achieved during cross-validation: 0.384000
```

In [131]:
```python
# Visualize the cross-validation results
import math
import pdb

# pdb.set_trace()

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.tight_layout(pad=3)
plt.scatter(x_scatter, y_scatter, marker_size, c=colors, cmap=plt.cm.c
oolwarm)
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, cmap=plt.cm.c
oolwarm)
plt.colorbar()
plt.xlabel('log learning rate')
plt.ylabel('log regularization strength')
plt.title('CIFAR-10 validation accuracy')
plt.show()
```

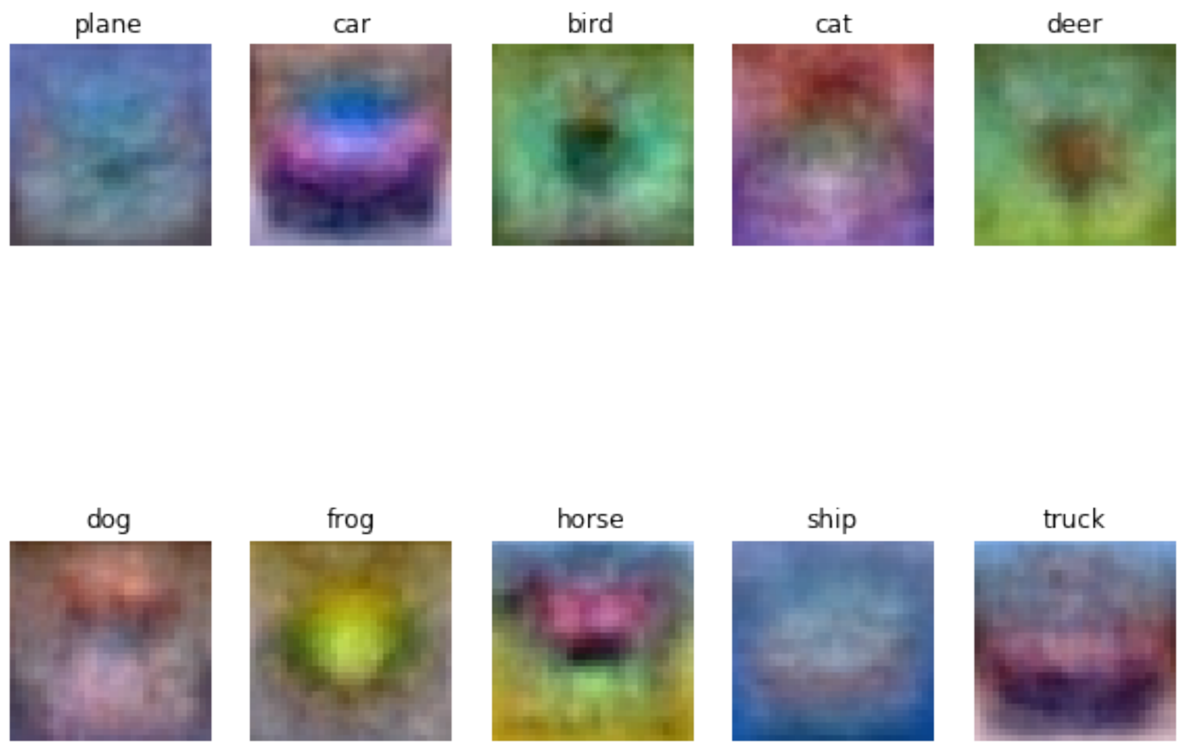




In [124]:
```
# Evaluate the best svm on test set
y_test_pred = best_svm.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('linear SVM on raw pixels final test set accuracy: %f' % test_ac
curacy)
```

```
linear SVM on raw pixels final test set accuracy: 0.371000
```

In [125]:
```
# Visualize the learned weights for each class.
# Depending on your choice of learning rate and regularization strengt
h, these may
# or may not be nice to look at.
w = best_svm.W[:-1,:] # strip out the bias
w = w.reshape(32, 32, 3, 10)
w_min, w_max = np.min(w), np.max(w)
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'hors
e', 'ship', 'truck']
for i in range(10):
    plt.subplot(2, 5, i + 1)

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



Inline question 2

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

$Your Answer$: they look like initial outlines for what the different image classes are. It could look this way because the robustness of the network analyzing the dataset is not as strong/robust as if we were to use a more complex CNN.

```
In [ ]:
```

Softmax exercise

Complete and hand in this completed worksheet (including its outputs and any supporting code outside of the worksheet) with your assignment submission. For more details see the [assignments page (http://vision.stanford.edu/teaching/cs231n/assignments.html)](http://vision.stanford.edu/teaching/cs231n/assignments.html) 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 [1]:
```
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plo
ts
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

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

In [2]:
```
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 p
repare
    it for the linear classifier. These are the same steps as we used
for the
    SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

    # Cleaning up variables to prevent loading data multiple times (wh
ich may cause memory issue)
```

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

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

    # 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_de
v


# 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_CIF
AR10_data()
print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
print('dev data shape: ', X_dev.shape)
print('dev labels shape: ', y_dev.shape)
```

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

Softmax Classifier

Your code for this section will all be written inside `cs231n/classifiers/softmax.py` .

In [8]:

```
# 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 lo
ss.
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: 2.380704
sanity check: 2.302585
```

Inline Question 1

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

$Your Answer:$ Our initialization number of classes is low, which means that the probability predictions will all huddle around the same distribution at initialization, close to 0.1.

In [9]:
```
# Complete the implementation of softmax_loss_naive and implement a (n
aive)
# 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: -0.714321 analytic: -0.714321, relative error: 2.023323e-
08
numerical: -3.132088 analytic: -3.132088, relative error: 2.194385e-
08
numerical: -0.416735 analytic: -0.416735, relative error: 3.867904e-
08
numerical: 1.327177 analytic: 1.327177, relative error: 2.587317e-08
numerical: 1.399740 analytic: 1.399740, relative error: 4.753285e-08
numerical: 1.812898 analytic: 1.812898, relative error: 2.535014e-08
numerical: -0.888163 analytic: -0.888163, relative error: 2.589727e-
08
numerical: -1.257603 analytic: -1.257603, relative error: 3.617679e-
08
numerical: -1.854077 analytic: -1.854077, relative error: 1.288557e-
08
numerical: -1.224152 analytic: -1.224153, relative error: 2.582007e-
08
numerical: 1.321453 analytic: 1.321453, relative error: 4.461755e-09
numerical: -2.744751 analytic: -2.744751, relative error: 6.710593e-
09
numerical: 0.364409 analytic: 0.364409, relative error: 1.493146e-08
numerical: -3.907450 analytic: -3.907450, relative error: 4.122210e-
09
numerical: 2.346587 analytic: 2.346587, relative error: 9.298321e-09
numerical: -0.118372 analytic: -0.118372, relative error: 6.468934e-
08
numerical: -0.007598 analytic: -0.007598, relative error: 4.914527e-
06
numerical: 0.591932 analytic: 0.591932, relative error: 5.035477e-08
numerical: 0.348856 analytic: 0.348856, relative error: 1.091617e-07
numerical: -1.033043 analytic: -1.033043, relative error: 2.639177e-
08
```

In [16]:
```
# 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='fr
o')
print('Loss difference: %f' % np.abs(loss_naive - loss_vectorized))
print('Gradient difference: %f' % grad_difference)
```

```
naive loss: 2.380704e+00 computed in 0.106275s
vectorized loss: 0.000000e+00 computed in 0.002561s
Loss difference: 2.380704
Gradient difference: 0.000000
```

In [25]:
```
# Use the validation set to tune hyperparameters (regularization stren
gth 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

######################################################################
##########
# TODO:
#
# Use the validation set to set the learning rate and regularization s
trength. #
# This should be identical to the validation that you did for the SVM;
```

```
save    #
# the best trained softmax classifer in best_softmax.
#
######################################################################
##########

# Provided as a reference. You may or may not want to change these hyp
erparameters
learning_rates = [1e-7, 5e-7]
regularization_strengths = [2.5e3, 5e3]

# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

for lr in learning_rates:
    for reg in regularization_strengths:
        softmax = Softmax() #specifying function
        softmax.train(X_train, y_train, learning_rate = lr, reg = reg,
num_iters = 1500)

        #train
        y_train_pred = softmax.predict(X_train)
        train_accuracy = np.mean(y_train == y_train_pred)

        #validation
        y_val_pred = softmax.predict(X_val)
        val_accuracy = np.mean(y_val == y_val_pred)

        if val_accuracy > best_val:
            best_val = val_accuracy
            best_softmax = softmax

        results[(lr, reg)] = train_accuracy, val_accuracy


        ##updating reg stregnth down by one factor (4-3) gave a 3 poin
t jump in cross valid accuracy.

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

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

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

```
lr 1.000000e-07 reg 2.500000e+03 train accuracy: 0.274163 val accura
cy: 0.276000
lr 1.000000e-07 reg 5.000000e+03 train accuracy: 0.298265 val accura
cy: 0.294000
lr 5.000000e-07 reg 2.500000e+03 train accuracy: 0.389429 val accura
cy: 0.384000
lr 5.000000e-07 reg 5.000000e+03 train accuracy: 0.385673 val accura
cy: 0.393000
best validation accuracy achieved during cross-validation: 0.393000
```

In [26]:
```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_accu
racy, ))
```

```
softmax on raw pixels final test set accuracy: 0.373000
```

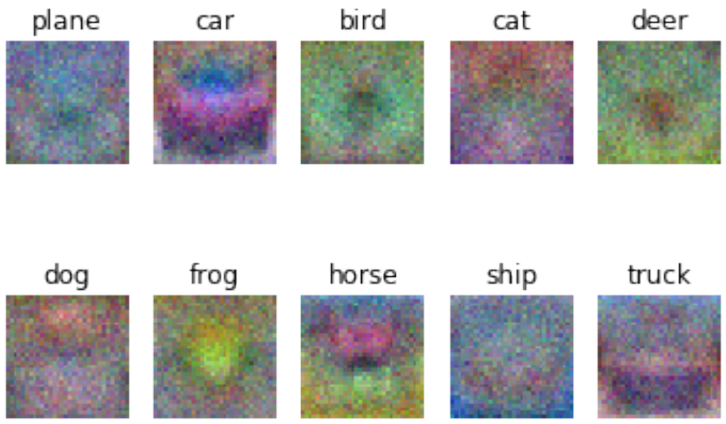
Inline Question 2 - *True or False*

Suppose the overall training loss is defined as the sum of the per-datapoint loss over all training examples. It is 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.

$Your Answer :$ yes, it is possible - True.

$Your Explanation :$ with softmax, because we are in log scale, the overall value added to our classifier will always be greater than 0.

In [27]:
```
# 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', 'hors
e', 'ship', 'truck']
for i in range(10):
    plt.subplot(2, 5, i + 1)

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



In [ ]:

Implementing a Neural Network

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

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

import numpy as np
import matplotlib.pyplot as plt

from cs231n.classifiers.neural_net import TwoLayerNet

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plo
ts
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

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

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

We will use the class `TwoLayerNet` in the file `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 [2]:
```
# Create a small net and some toy data to check your implementations.
# Note that we set the random seed for repeatable experiments.

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

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

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

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

Forward pass: compute scores

Open the file `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 [5]:
```python
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:
3.6802720745909845e-08
```

Forward pass: compute loss

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

In [15]:
```python
loss, _ = net.loss(X, y, reg=0.05)
correct_loss = 1.30378789133

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

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

Backward pass

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

In [16]:
```
from cs231n.gradient_check import eval_numerical_gradient

# Use numeric gradient checking to check your implementation of the ba
ckward pass.
# If your implementation is correct, the difference between the numeri
c 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_g
rad_num, grads[param_name])))
```

```
W2 max relative error: 3.440708e-09
b2 max relative error: 4.447625e-11
W1 max relative error: 3.561318e-09
b1 max relative error: 2.738421e-09
```

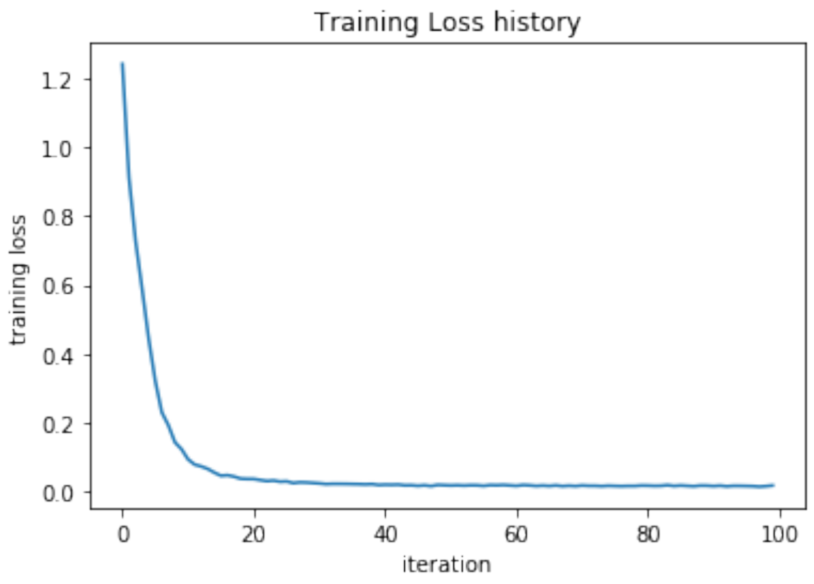
Train the network

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 0.02.

In [19]:
```
net = init_toy_model()
stats = net.train(X, y, X, y,
            learning_rate=1e-1, reg=5e-6,
            num_iters=100, verbose=False)

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

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

```
Final training loss:  0.017149607938732093
```



Load the data

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

In [20]:
```
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 p
repare
    it for the two-layer neural net classifier. These are the same ste
ps 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'

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

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

    # 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


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

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

Train a network

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

In [21]:
```
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
```

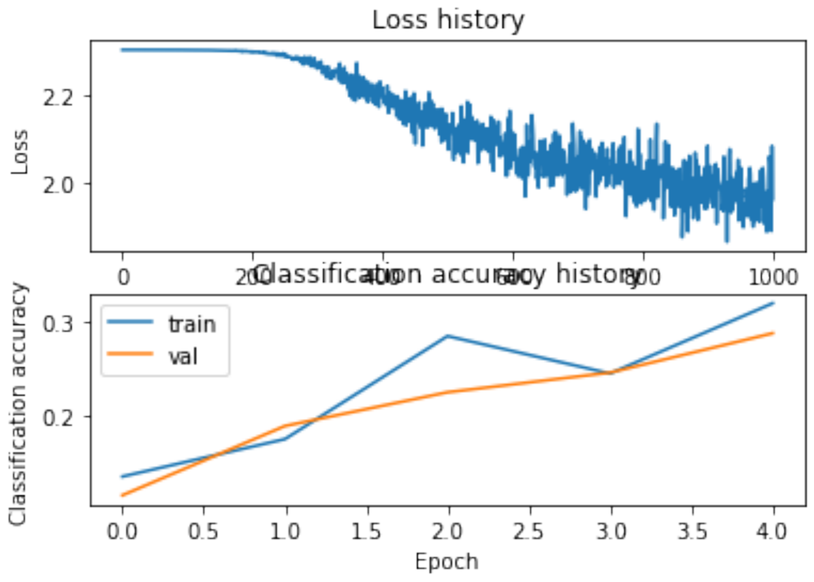
Debug the training

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.

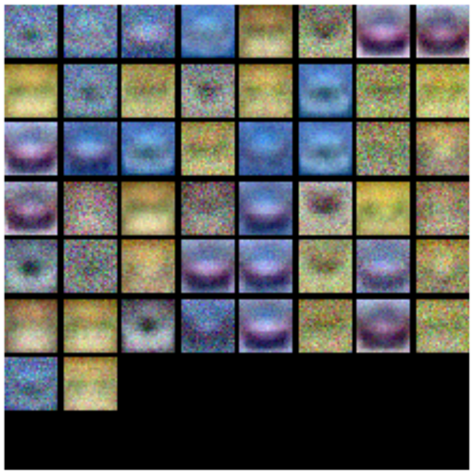
One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

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 [22]:
```
# 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('Classification accuracy')
plt.legend()
plt.show()
```



In [23]:
```
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 (52% could serve as a reference), 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.).

Explain your hyperparameter tuning process below.

$Your Answer$: we would want to adjust things like the learning rate, regularization, and hidden size in the beginning. This will allow for us to tune the network a bit better.

```
In [39]: best_net = None # store the best model into this

         ######################################################################
         ###########
         # TODO: Tune hyperparameters using the validation set. Store your best
         trained  #
         # model in best_net.
         #
         #
         #
         # To help debug your network, it may help to use visualizations simila
         r to the  #
         # ones we used above; these visualizations will have significant quali
         tative    #
         # differences from the ones we saw above for the poorly tuned network.
         #
         #
         #
         # Tweaking hyperparameters by hand can be fun, but you might find it u
         seful to  #
         # write code to sweep through possible combinations of hyperparameters
```

```
#
# automatically like we did on the previous exercises.
#
##########################################################################
###########
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

best_val = -1
learning_rates = [1e-1, 1e-3, 1e-5]
regularization_strengths = [1e2, 1, 1e-1,]
hidden_sizes = [50, 100, 150]


for lr in learning_rates:
    for reg in regularization_strengths:
        for hidden in hidden_sizes:
            net = TwoLayerNet(input_size, hidden, num_classes)
            data = net.train(X_train, y_train, X_val, y_val, learning_
rate = lr, reg = reg, num_iters=1500, batch_size = 200)


            # training set evaluation
            train_accuracy = (net.predict(X_train) == y_train).mean()

            # validation set evaluation
            val_accuracy = (net.predict(X_val) == y_val).mean()

            print('lr = %f, reg = %f, hidden = %d,' % (lr, reg, hidden
))
            print('val_acc = %f' % (val_accuracy))
            print('------------')

            if val_accuracy > best_val:
                best_val = val_accuracy
                best_net = net

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

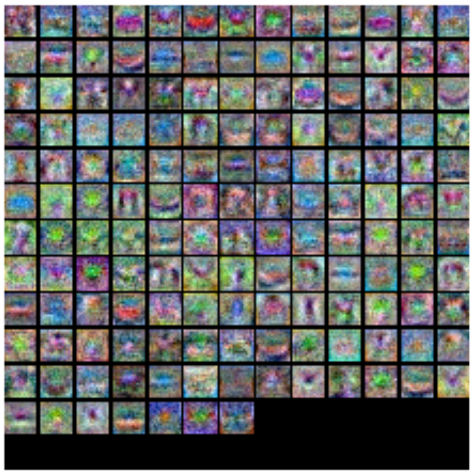
```
lr = 0.100000, reg = 100.000000, hidden = 50,
val_acc = 0.087000
------------
lr = 0.100000, reg = 100.000000, hidden = 100,
val_acc = 0.087000
------------
lr = 0.100000, reg = 100.000000, hidden = 150,
val_acc = 0.087000
------------
lr = 0.100000, reg = 1.000000, hidden = 50,
val_acc = 0.087000
------------
lr = 0.100000, reg = 1.000000, hidden = 100,
val_acc = 0.087000
------------
lr = 0.100000, reg = 1.000000, hidden = 150,
val_acc = 0.087000
-----------
lr = 0.100000, reg = 0.100000, hidden = 50,
val_acc = 0.087000
-----------
lr = 0.100000, reg = 0.100000, hidden = 100,
val_acc = 0.087000
-----------
lr = 0.100000, reg = 0.100000, hidden = 150,
val_acc = 0.087000
-----------
lr = 0.001000, reg = 100.000000, hidden = 50,
val_acc = 0.087000
-----------
lr = 0.001000, reg = 100.000000, hidden = 100,
val_acc = 0.107000
-----------
lr = 0.001000, reg = 100.000000, hidden = 150,
val_acc = 0.079000
-----------
lr = 0.001000, reg = 1.000000, hidden = 50,
val_acc = 0.469000
-----------
lr = 0.001000, reg = 1.000000, hidden = 100,
val_acc = 0.486000
-----------
lr = 0.001000, reg = 1.000000, hidden = 150,
val_acc = 0.479000
-----------
lr = 0.001000, reg = 0.100000, hidden = 50,
val_acc = 0.478000
-----------
lr = 0.001000, reg = 0.100000, hidden = 100,
val_acc = 0.499000
-----------
lr = 0.001000, reg = 0.100000, hidden = 150,
val_acc = 0.503000
-----------
lr = 0.000010, reg = 100.000000, hidden = 50,
val_acc = 0.204000
-----------
lr = 0.000010, reg = 100.000000, hidden = 100,
val_acc = 0.146000
-----------
lr = 0.000010, reg = 100.000000, hidden = 150,
val_acc = 0.212000
-----------
lr = 0.000010, reg = 1.000000, hidden = 50,
val_acc = 0.186000
-----------
lr = 0.000010, reg = 1.000000, hidden = 100,
val_acc = 0.214000
-----------
```

```
lr = 0.000010, reg = 1.000000, hidden = 150,
val_acc = 0.234000
-----------
lr = 0.000010, reg = 0.100000, hidden = 50,
val_acc = 0.191000
-----------
lr = 0.000010, reg = 0.100000, hidden = 100,
val_acc = 0.198000
-----------
lr = 0.000010, reg = 0.100000, hidden = 150,
val_acc = 0.231000
-----------
```

In [40]:
```
# Print your validation accuracy: this should be above 48%
val_acc = (best_net.predict(X_val) == y_val).mean()
print('Validation accuracy: ', val_acc)
```

```
Validation accuracy:  0.503
```

In [41]:
```
# 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 [45]:
```
# Print your test accuracy: this should be above 48%
test_acc = (best_net.predict(X_test) == y_test).mean()
print('Test accuracy: ', test_acc)
```

```
Test accuracy:  0.495
```

Inline Question

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

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

$Your Answer:$ 1 and 3.

$Your Explanation:$ more data and regularization can improve network performance. If we add a lot of hidden units we can loose the ability to generalize.

```
In [ ]:
```

Image features exercise

Complete and hand in this completed worksheet (including its outputs and any supporting code outside of the worksheet) with your assignment submission. For more details see the [assignments page (http://vision.stanford.edu/teaching/cs231n/assignments.html)](http://vision.stanford.edu/teaching/cs231n/assignments.html) on the course website.

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

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

In [1]:
```
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt


%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plo
ts
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

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

Load data

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

```
In [2]: from 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'

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

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

    # 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

X_train, y_train, X_val, y_val, X_test, y_test = get_CIFAR10_data()
```

Extract Features

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

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

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 [3]:
```
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
feature
# has roughly the same scale.
std_feat = np.std(X_train_feats, axis=0, keepdims=True)
X_train_feats /= std_feat
X_val_feats /= std_feat
X_test_feats /= std_feat

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

```
Done extracting features for 1000 / 49000 images
Done extracting features for 2000 / 49000 images
Done extracting features for 3000 / 49000 images
Done extracting features for 4000 / 49000 images
Done extracting features for 5000 / 49000 images
Done extracting features for 6000 / 49000 images
Done extracting features for 7000 / 49000 images
Done extracting features for 8000 / 49000 images
Done extracting features for 9000 / 49000 images
Done extracting features for 10000 / 49000 images
Done extracting features for 11000 / 49000 images
Done extracting features for 12000 / 49000 images
Done extracting features for 13000 / 49000 images
Done extracting features for 14000 / 49000 images
Done extracting features for 15000 / 49000 images
Done extracting features for 16000 / 49000 images
Done extracting features for 17000 / 49000 images
Done extracting features for 18000 / 49000 images
Done extracting features for 19000 / 49000 images
Done extracting features for 20000 / 49000 images
Done extracting features for 21000 / 49000 images
Done extracting features for 22000 / 49000 images
Done extracting features for 23000 / 49000 images
Done extracting features for 24000 / 49000 images
Done extracting features for 25000 / 49000 images
Done extracting features for 26000 / 49000 images
Done extracting features for 27000 / 49000 images
Done extracting features for 28000 / 49000 images
Done extracting features for 29000 / 49000 images
Done extracting features for 30000 / 49000 images
Done extracting features for 31000 / 49000 images
Done extracting features for 32000 / 49000 images
Done extracting features for 33000 / 49000 images
Done extracting features for 34000 / 49000 images
Done extracting features for 35000 / 49000 images
Done extracting features for 36000 / 49000 images
Done extracting features for 37000 / 49000 images
Done extracting features for 38000 / 49000 images
Done extracting features for 39000 / 49000 images
Done extracting features for 40000 / 49000 images
Done extracting features for 41000 / 49000 images
Done extracting features for 42000 / 49000 images
Done extracting features for 43000 / 49000 images
Done extracting features for 44000 / 49000 images
Done extracting features for 45000 / 49000 images
Done extracting features for 46000 / 49000 images
Done extracting features for 47000 / 49000 images
Done extracting features for 48000 / 49000 images
Done extracting features for 49000 / 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 [ ]:
```
# 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]
regularization_strengths = [5e4, 5e5, 5e6]

results = {}
best_val = -1
best_svm = None

######################################################################
##########
# TODO:
#
# Use the validation set to set the learning rate and regularization s
trength. #
# 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 ca
reful    #
# you should be able to get accuracy of near 0.44 on the validation se
t.       #
######################################################################
##########
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

pass

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

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

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

In [ ]:
```
# Evaluate your trained SVM on the test set: you should be able to get
at least 0.40
y_test_pred = best_svm.predict(X_test_feats)
test_accuracy = np.mean(y_test == y_test_pred)
print(test_accuracy)
```

In [ ]:
```
# An important way to gain intuition about how an algorithm works is t
o
# visualize the mistakes that it makes. In this visualization, we show
examples
# of images that are misclassified by our current system. The first co
lumn
# 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', 'hors
e', 'ship', 'truck']
for cls, cls_name in enumerate(classes):
    idxs = np.where((y_test != cls) & (y_test_pred == cls))[0]
    idxs = np.random.choice(idxs, examples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt.subplot(examples_per_class, len(classes), i * len(classes)
+ cls + 1)
        plt.imshow(X_test[idx].astype('uint8'))
        plt.axis('off')
        if i == 0:
            plt.title(cls_name)
plt.show()
```

Inline question 1:

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

$Your Answer :$

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 [ ]:
```
# 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)
```

In [ ]:
```
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 wa
nt to    #
# cross-validate various parameters as in previous sections. Store you
r best   #
# model in the best_net variable.
#
########################################################################
##########
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

pass

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

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

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