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.

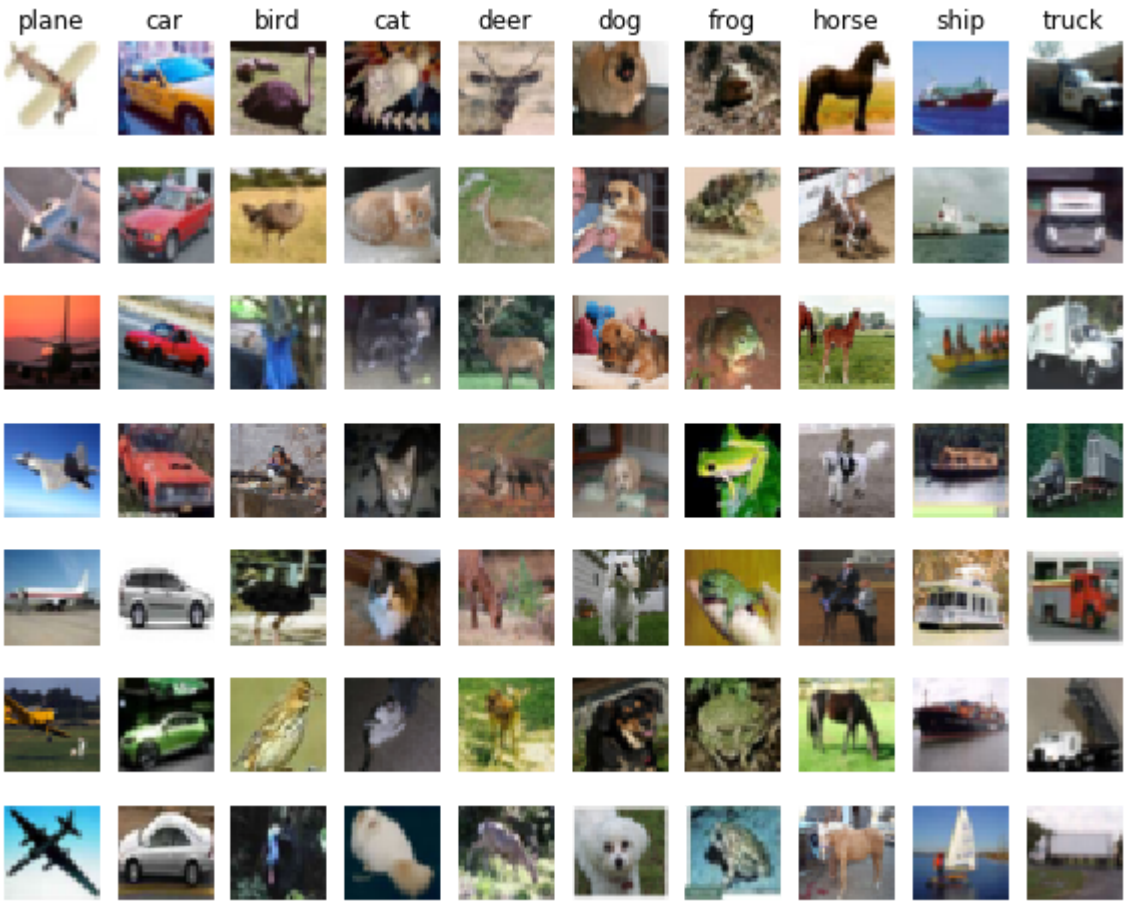
In [287]:
```
# Run some setup code for this notebook.

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 the noteb
ook
# 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-ipyt
hon
%load_ext autoreload
%autoreload 2
```

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

In [288]:
```
# 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 caus
e 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 [289]:
```
# Visualize some examples from the dataset.
# We show a few examples of training images from each class.
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
num_classes = len(classes)
samples_per_class = 7
for y, cls in enumerate(classes):
    idxs = np.flatnonzero(y_train == y) # This creates an index, and excludes any values for which the value is 0
    idxs = np.random.choice(idxs, samples_per_class, replace=False) # takes a # of samples per class randomly from idxs
    for i, idx in enumerate(idxs):
        plt_idx = i * num_classes + y + 1 # indicates the plot index for that image
        plt.subplot(samples_per_class, num_classes, plt_idx) # creates subplots
        plt.imshow(X_train[idx].astype('uint8')) # plots images
        plt.axis('off')
        if i == 0:
            plt.title(cls)
plt.show()
```



In [290]:
```
# Subsample the data for more efficient code execution in this exercise
num_training = 5000
mask = list(range(num_training))
X_train = X_train[mask]
y_train = y_train[mask]

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

# Reshape the image data into rows
# The -1 here infers the shape of the data based on previous dimensions
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 [291]:
```
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.

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.

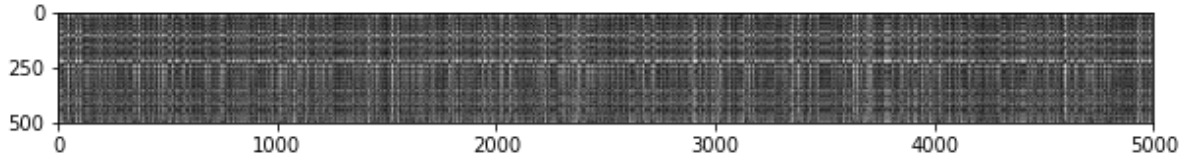
In [292]:
```
train_array = X_train[0]
test_array = X_test[0]
print(train_array)
print(test_array)
```

```
[ 59.  62.  63. ... 123.  92.  72.]
[158. 112.  49. ...  21.  67. 110.]
```

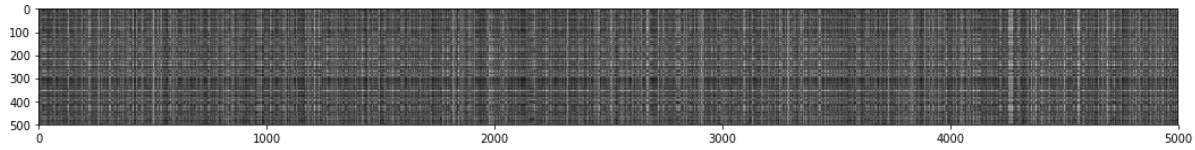
In [293]:
```
# Open cs231n/classifiers/k_nearest_neighbor.py and implement
# compute_distances_two_loops.

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

```
(500, 5000)
```

In [294]:
```
# 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()
```



In [295]:
```
# Doing a larger visualization
fig, ax = plt.subplots(figsize=(18, 2))
ax.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 :

- A bright row is caused by a test image that has a high distance value with respect to every training image
- A bright column is caused by a training image that has a high distance value with respect to every test image

Both of these scenarios can be caused by an image that is highly atypical, leading to large differences in pixels with most other images. The L2 distance penalizes heavy deviations more severely, so a highly atypical image would cause large distance values with most other images.

In [296]:
```
# 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 [297]:
```
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.

Your Answer :

1, 2 and 3 will not change the performance of the Nearest Neighbor classifier that uses L1 distance.

Your Explanation :

I simplify the i, j index by flattening the pixel array, and in this answer I refer to the pixel position with the index p

1. By subtracting the mean μ across all images, we are subtracting a constant for each pixel. In the subtraction of the L1 distance, the mean will be 'canceled out.

$$D_i = \sum_{p=1}^{3072} |(I_{i_{train}}^p - \mu) - (I_{i_{test}}^p - \mu)|$$
$$D_i = \sum_{p=1}^{3072} |I_{i_{train}}^p - I_{i_{test}}^p + \mu - \mu|$$
$$D_i = \sum_{p=1}^{3072} |I_{i_{train}}^p - I_{i_{test}}^p|$$

1. By subtracting the pixel-wise mean μ^p, the same will happen at each operation. The pixel-wise mean will be canceled out in each operation.

$$D_i = \sum_{p=1}^{3072} |(I_{i_{train}}^p - \mu^p) - (I_{i_{test}}^p - \mu^p)|$$
$$D_i = \sum_{p=1}^{3072} |I_{i_{train}}^p - I_{i_{test}}^p + \mu^p - \mu^p|$$

$$D_i = \sum_{p=1}^{3072} |I^p_{i_{train}} - I^p_{i_{test}}|$$

1. Subtracting the mean μ and dividing by the standard deviation σ does not change the KNN classifier. As in the answer for (1), the μ term makes no difference. The σ term is a constant that is divided over all distances, and is therefore a monotonic transformation of the distances, which will not change the ordering of the nearest neighbors.

$$D_i = \sum_{p=1}^{3072} |I^p_{i_{train}} - I^p_{i_{test}}|$$

$$D'_i = \sum_{p=1}^{3072} \left| \frac{(I^p_{i_{train}} - \mu)}{\sigma} - \frac{(I^p_{i_{test}} - \mu)}{\sigma} \right|$$

$$D'_i = \sum_{p=1}^{3072} \left| \frac{I^p_{i_{train}} - I^p_{i_{test}} + \mu - \mu}{\sigma} \right|$$

$$D'_i = \sum_{p=1}^{3072} \left| \frac{I^p_{i_{train}} - I^p_{i_{test}}}{\sigma} \right| = \frac{1}{\sigma} \sum_{p=1}^{3072} |I^p_{i_{train}} - I^p_{i_{test}}|$$

1. Subtracting the pixel-wise mean μ_p does not make any difference, as seen in answer to (2). However, dividing by the pixel-wise standard deviation σ_p creates a change in each pixel-wise difference, which distorts the overall distance depending on the pixels of each image. This is no longer a monotonic transformation for all images, and therefore will change the ordering of neighbors in the KNN algorithm.

$$D_i = \sum_{p=1}^{3072} |I^p_{i_{train}} - I^p_{i_{test}}|$$

$$D'_i = \sum_{p=1}^{3072} \left| \frac{(I^p_{i_{train}} - \mu^p)}{\sigma_p} - \frac{(I^p_{i_{test}} - \mu^p)}{\sigma_p} \right|$$

$$D'_i = \sum_{p=1}^{3072} \left| \frac{I^p_{i_{train}} - I^p_{i_{test}} + \mu^p - \mu^p}{\sigma_p} \right|$$

$$D'_i = \sum_{p=1}^{3072} \left| \frac{I^p_{i_{train}} - I^p_{i_{test}}}{\sigma_p} \right|$$

1. Rotating the coordinate axis will change the distance between any two datapoints for L1 distance, although that would not be the case for L2 distance. L1 can be visualized as a square in two dimensions, and the distance between two points of the square is not the same if the coordinate axis is rotated.

In [298]:
```
# Now lets speed up distance matrix computation by using partial vectorization
# with one loop. Implement the function compute_distances_one_loop and run the
# code below:
dists_one = classifier.compute_distances_one_loop(X_test)

# To ensure that our vectorized implementation is correct, we make sure that i
t
# agrees with the naive implementation. There are many ways to decide whether
# two matrices are similar; one of the simplest is the Frobenius norm. In case
# you haven't seen it before, the Frobenius norm of two matrices is the square
# root of the squared sum of differences of all elements; in other words, resh
ape
# 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 [299]:
```
# 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('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 [300]:
```
# Let's compare how fast the implementations are
def time_function(f, *args):
    """
    Call a function f with args and return the time (in seconds) that it took
 to execute.
    """
    import time
    tic = time.time()
    f(*args)
    toc = time.time()
    return toc - tic

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

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

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

# You should see significantly faster performance with the fully vectorized im
plementation!

# 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 28.558652 seconds
One loop version took 66.755361 seconds
No loop version took 0.216421 seconds
```

Cross-validation

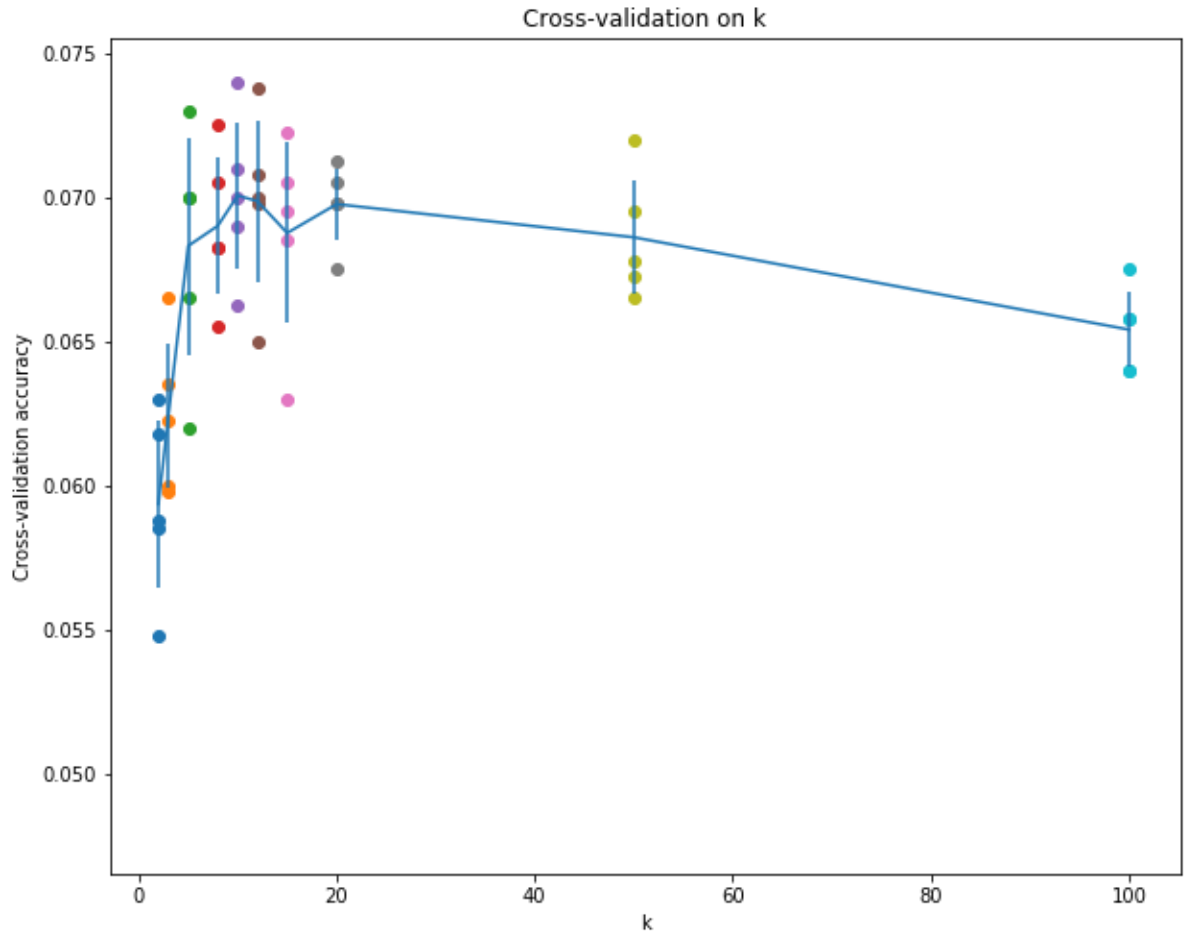
We have implemented the k-Nearest Neighbor classifier but we set the value k = 5 arbitrarily. We will now determine the best value of this hyperparameter with cross-validation.

In [301]:
```
num_folds = 5
k_choices = [2, 3, 5, 8, 10, 12, 15, 20, 50, 100]

X_train_folds = []
y_train_folds = []

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

################################################################################
##
# TODO:
#
# Perform k-fold cross validation to find the best value of k. For each
#
# possible value of k, run the k-nearest-neighbor algorithm num_folds times,
#
# where in each case you use all but one of the folds as training data and the
#
# last fold as a validation set. Store the accuracies for all fold and all
#
# values of k in the k_to_accuracies dictionary.
#
################################################################################
##
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

for k in k_choices:
    accuracy_list = []
    for fold in range(num_folds):
        # Creating folds for X_training
        x_train_fold = X_train_folds[:fold] + X_train_folds[1+fold:]
        x_train_fold = np.reshape(x_train_fold, (-1, X_train.shape[1]))
        # Creating folds for Y_training
        y_train_fold = y_train_folds[:fold] + y_train_folds[1+fold:]
        y_train_fold = np.reshape(y_train_fold, (-1, 1))
        # Extracting the validation fold for the image and label
        x_val_fold = X_train_folds[fold]
        y_val_fold = y_train_folds[fold]

        # number of tests is the shape of x_train_fold
        num_test_fold = x_train_fold.shape[0]
        knn = KNearestNeighbor() # initializing KNN class
        knn.train(x_train_fold, y_train_fold) # training data on the training
 folds
        dist_fold = knn.compute_distances_no_loops(x_val_fold) # computing dis
tance between x training and x validation
        y_val_pred = knn.predict_labels(dist_fold, k=k) # predicting labels fo
```

```
r the validation set
        num_correct_fold = np.sum(y_val_pred == y_val_fold) # computing number
of correct predictions
        accuracy = float(num_correct_fold) / num_test_fold
        accuracy_list.append(accuracy) # storing accuracy in a list for this f
old
    k_to_accuracies[k] = accuracy_list # storing list in dictionary

# *****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 = 2, accuracy = 0.058750
k = 2, accuracy = 0.054750
k = 2, accuracy = 0.058500
k = 2, accuracy = 0.061750
k = 2, accuracy = 0.063000
k = 3, accuracy = 0.059750
k = 3, accuracy = 0.062250
k = 3, accuracy = 0.060000
k = 3, accuracy = 0.066500
k = 3, accuracy = 0.063500
k = 5, accuracy = 0.062000
k = 5, accuracy = 0.066500
k = 5, accuracy = 0.070000
k = 5, accuracy = 0.073000
k = 5, accuracy = 0.070000
k = 8, accuracy = 0.065500
k = 8, accuracy = 0.070500
k = 8, accuracy = 0.068250
k = 8, accuracy = 0.072500
k = 8, accuracy = 0.068250
k = 10, accuracy = 0.066250
k = 10, accuracy = 0.074000
k = 10, accuracy = 0.069000
k = 10, accuracy = 0.071000
k = 10, accuracy = 0.070000
k = 12, accuracy = 0.065000
k = 12, accuracy = 0.073750
k = 12, accuracy = 0.069750
k = 12, accuracy = 0.070750
k = 12, accuracy = 0.070000
k = 15, accuracy = 0.063000
k = 15, accuracy = 0.072250
k = 15, accuracy = 0.069500
k = 15, accuracy = 0.070500
k = 15, accuracy = 0.068500
k = 20, accuracy = 0.067500
k = 20, accuracy = 0.069750
k = 20, accuracy = 0.069750
k = 20, accuracy = 0.070500
k = 20, accuracy = 0.071250
k = 50, accuracy = 0.067750
k = 50, accuracy = 0.072000
k = 50, accuracy = 0.069500
k = 50, accuracy = 0.067250
k = 50, accuracy = 0.066500
k = 100, accuracy = 0.064000
k = 100, accuracy = 0.067500
k = 100, accuracy = 0.065750
k = 100, accuracy = 0.064000
k = 100, accuracy = 0.065750
```

In [302]:
```
# plot the raw observations
for k in k_choices:
    accuracies = k_to_accuracies[k]
    plt.scatter([k] * len(accuracies), accuracies)

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



In [303]:
```
# Based on the cross-validation results above, choose the best value for k,
# retrain the classifier using all the training data, and test it on the test
# data. You should be able to get above 28% accuracy on the test data.
best_k = 10

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

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

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

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 are true

Your Explanation :

1. Decision boundaries are determined by the position of the k-nearest neighbors. This is not necessarily linear.
2. Within the training sample, the error with a 1-NN will always be smaller than a 5-NN because it classifies correctly all of the training images. We increase k to avoid overfitting in the hope that we will have reduced error in the training sample.
3. Testing error will not always be lower for a 1-NN. If we overfitted the model in the training sample with a 1-NN, then a 5-NN will likely have reduced error in the test sample.
4. At the time of prediction, KNN must compare to every image in the training set. Therefore, the time needed to classify images increases with the size of the training set.

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 [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 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-ipyt
hon
%load_ext autoreload
%autoreload 2
```

CIFAR-10 Data Loading and Preprocessing

In [175]:
```
# 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 caus
e 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 [176]:
```
# Visualize some examples from the dataset.
# We show a few examples of training images from each class.
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
num_classes = len(classes)
samples_per_class = 7
for y, cls in enumerate(classes):
    idxs = np.flatnonzero(y_train == y)
    idxs = np.random.choice(idxs, samples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt_idx = i * num_classes + y + 1
        plt.subplot(samples_per_class, num_classes, plt_idx)
        plt.imshow(X_train[idx].astype('uint8'))
        plt.axis('off')
        if i == 0:
            plt.title(cls)
plt.show()
```

In [177]:
```
# 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) # obtains 500 sa
mples from 0 to 49000
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)
```

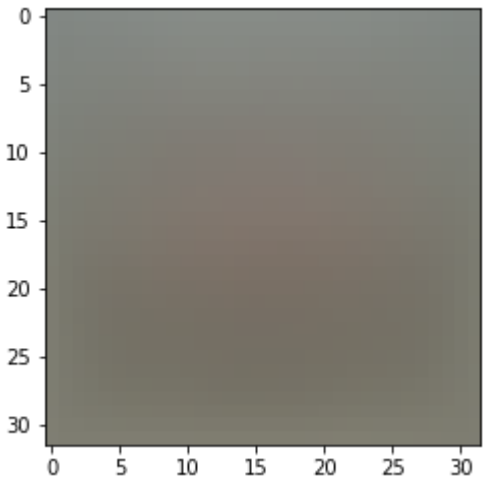
```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

In [178]:
```
# 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 [179]:
```
# 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 our SVM
# only has to worry about optimizing a single weight matrix W.
X_train = np.hstack([X_train, np.ones((X_train.shape[0], 1))])
X_val = np.hstack([X_val, np.ones((X_val.shape[0], 1))])
X_test = np.hstack([X_test, np.ones((X_test.shape[0], 1))])
X_dev = np.hstack([X_dev, np.ones((X_dev.shape[0], 1))])

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

```
[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 [180]:
```
# 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: 9.424864
```

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 [181]:
```
# 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, a
nd
# compare them with your analytically computed gradient. The numbers should ma
tch
# 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: -9.350163 analytic: -9.350163, relative error: 1.648079e-11
numerical: 5.451811 analytic: 5.451811, relative error: 2.652621e-11
numerical: -3.907819 analytic: -3.907819, relative error: 2.130522e-10
numerical: 5.821162 analytic: 5.821162, relative error: 1.218057e-11
numerical: -13.780473 analytic: -13.780473, relative error: 1.066364e-11
numerical: -2.277885 analytic: -2.257920, relative error: 4.401535e-03
numerical: -10.605833 analytic: -10.605833, relative error: 4.553179e-13
numerical: -12.919596 analytic: -12.919596, relative error: 1.390991e-11
numerical: 4.561826 analytic: 4.558914, relative error: 3.193290e-04
numerical: 17.002048 analytic: 17.002048, relative error: 6.420338e-12
numerical: -11.586712 analytic: -11.586712, relative error: 2.845768e-11
numerical: 9.370009 analytic: 9.370009, relative error: 3.354411e-12
numerical: 2.200947 analytic: 2.200947, relative error: 2.509063e-11
numerical: -22.934765 analytic: -22.934765, relative error: 4.997473e-12
numerical: 10.364227 analytic: 10.364227, relative error: 2.295733e-11
numerical: 36.094749 analytic: 36.094749, relative error: 1.892760e-12
numerical: 19.844339 analytic: 19.844339, relative error: 2.671561e-12
numerical: -19.187354 analytic: -19.187354, relative error: 3.343048e-13
numerical: 34.561351 analytic: 34.561351, relative error: 2.243488e-12
numerical: 6.760268 analytic: 6.760268, relative error: 6.623105e-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*

$Your Answer:$

This can happen when we're computing the gradient very close to the point where it is not differentiable. Before the condition is met, the gradient is 0. While this will be the result from the analytical computation, the numerical computation can have a different result. If h (the derivative step size) is larger than the difference to get past the non-differentiable point, then the numerical gradient at that point will be positive, although in reality it should be 0.

A simple example. Consider the following function:

$$f(x) = \begin{cases} 0, & \text{if } x < 3 \\ x, & \text{if } x \geq 3 \end{cases}$$

With the following scenario:

$$h = 0.02$$
$$x = 2.99$$

In this case, the analytical gradient would be 0, but the numerical gradient would be 1.

If the margin Δ is large, then there will be less cases that are *at the margin*, and therefore the frequency of this effect occurring would be reduced.

In [182]:
```
# Next implement the function svm_loss_vectorized; for now only compute the lo
ss;
# 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 fa
ster.
print('difference: %f' % (loss_naive - loss_vectorized))
```

```
Naive loss: 9.424864e+00 computed in 0.506835s
Vectorized loss: 9.424864e+00 computed in 0.005985s
difference: -0.000000
```

In [183]:
```
# Complete the implementation of svm_loss_vectorized, and compute the gradient
# of the loss function in a vectorized way.

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

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

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

```
Naive loss and gradient: computed in 0.498191s
Vectorized loss and gradient: computed in 0.005977s
difference: 0.000000
```

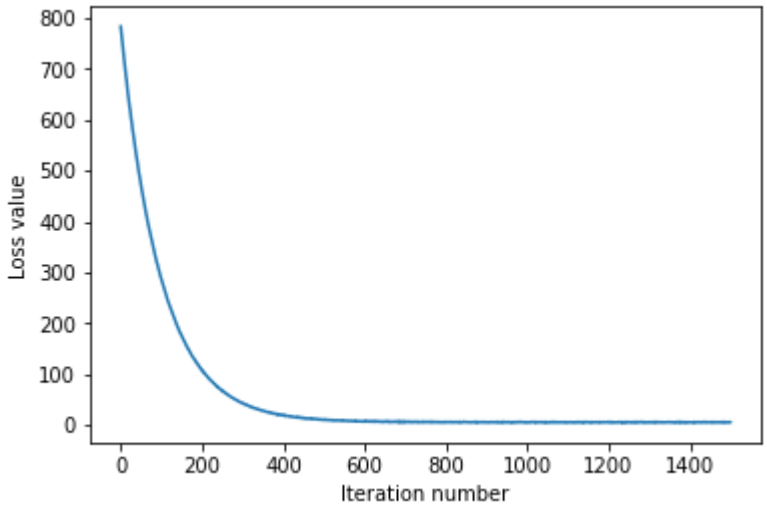
Stochastic Gradient Descent

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 [205]:
```
# 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 783.299545
iteration 100 / 1500: loss 286.286823
iteration 200 / 1500: loss 107.427802
iteration 300 / 1500: loss 42.363943
iteration 400 / 1500: loss 19.936079
iteration 500 / 1500: loss 10.215250
iteration 600 / 1500: loss 7.758142
iteration 700 / 1500: loss 5.582131
iteration 800 / 1500: loss 5.653924
iteration 900 / 1500: loss 5.255042
iteration 1000 / 1500: loss 4.845088
iteration 1100 / 1500: loss 5.008648
iteration 1200 / 1500: loss 5.929530
iteration 1300 / 1500: loss 5.744002
iteration 1400 / 1500: loss 5.209273
That took 14.559296s
```

In [206]:
```
# 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 [207]:
```
# Write the LinearSVM.predict function and evaluate the performance on both th
e
# 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: 0.366939
validation accuracy: 0.377000
```

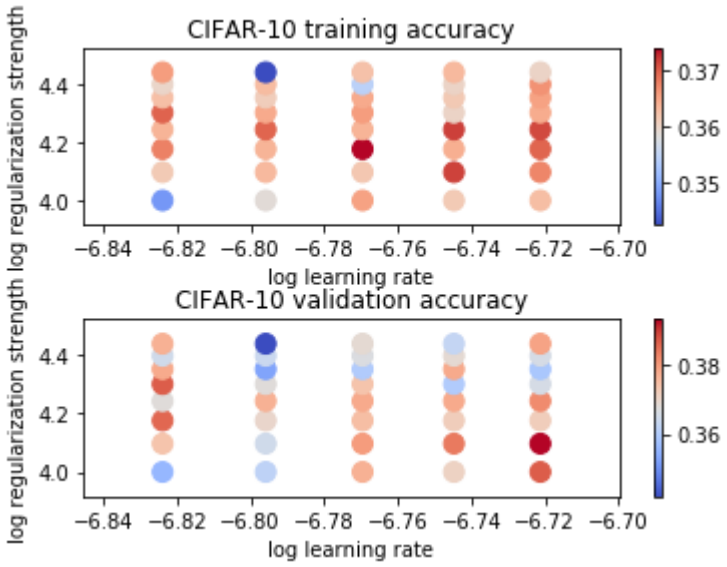
In [223]:
```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of about 0.39 on the validation set.

# Note: you may see runtime/overflow warnings during hyper-parameter search.
# 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 fractio
n
# 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 ra
te.

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

# Provided as a reference. You may or may not want to change these hyperparame
ters
learning_rates = np.arange(1.5,2,0.1) * 1e-7
regularization_strengths = np.arange(1,3,0.25)*1e4

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

for rate in learning_rates:
    for strength in regularization_strengths:
```

```
        # Initializing and training the class
        svm = LinearSVM()
        svm.train(X_train, y_train, learning_rate=rate, reg=strength, num_iter
s = 500)

        # Predicting values for training and validations sets
        y_train_pred = svm.predict(X_train)
        y_val_pred = svm.predict(X_val)

        # Verifying accuracy in training and validation samples
        train_accuracy = np.mean(y_train == y_train_pred)
        val_accuracy = np.mean(y_val == y_val_pred)

        # Storing results in the dictionary
        results[(rate, strength)] = (train_accuracy, val_accuracy)

        # Storing svm object and accuracy for best performing learning rate an
d regularization strength
        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_v
al)
```

```
lr 1.500000e-07 reg 1.000000e+04 train accuracy: 0.348000 val accuracy: 0.356
000
lr 1.500000e-07 reg 1.250000e+04 train accuracy: 0.361469 val accuracy: 0.374
000
lr 1.500000e-07 reg 1.500000e+04 train accuracy: 0.367918 val accuracy: 0.386
000
lr 1.500000e-07 reg 1.750000e+04 train accuracy: 0.363816 val accuracy: 0.368
000
lr 1.500000e-07 reg 2.000000e+04 train accuracy: 0.369939 val accuracy: 0.387
000
lr 1.500000e-07 reg 2.250000e+04 train accuracy: 0.362776 val accuracy: 0.378
000
lr 1.500000e-07 reg 2.500000e+04 train accuracy: 0.359898 val accuracy: 0.365
000
lr 1.500000e-07 reg 2.750000e+04 train accuracy: 0.365776 val accuracy: 0.377
000
lr 1.600000e-07 reg 1.000000e+04 train accuracy: 0.358673 val accuracy: 0.362
000
lr 1.600000e-07 reg 1.250000e+04 train accuracy: 0.363327 val accuracy: 0.365
000
lr 1.600000e-07 reg 1.500000e+04 train accuracy: 0.363816 val accuracy: 0.370
000
lr 1.600000e-07 reg 1.750000e+04 train accuracy: 0.369796 val accuracy: 0.377
000
lr 1.600000e-07 reg 2.000000e+04 train accuracy: 0.364673 val accuracy: 0.368
000
lr 1.600000e-07 reg 2.250000e+04 train accuracy: 0.360939 val accuracy: 0.353
000
lr 1.600000e-07 reg 2.500000e+04 train accuracy: 0.363041 val accuracy: 0.364
000
lr 1.600000e-07 reg 2.750000e+04 train accuracy: 0.342245 val accuracy: 0.342
000
lr 1.700000e-07 reg 1.000000e+04 train accuracy: 0.365449 val accuracy: 0.377
000
lr 1.700000e-07 reg 1.250000e+04 train accuracy: 0.361898 val accuracy: 0.380
000
lr 1.700000e-07 reg 1.500000e+04 train accuracy: 0.374102 val accuracy: 0.375
000
lr 1.700000e-07 reg 1.750000e+04 train accuracy: 0.363796 val accuracy: 0.378
000
lr 1.700000e-07 reg 2.000000e+04 train accuracy: 0.365816 val accuracy: 0.374
000
lr 1.700000e-07 reg 2.250000e+04 train accuracy: 0.364755 val accuracy: 0.360
000
lr 1.700000e-07 reg 2.500000e+04 train accuracy: 0.354469 val accuracy: 0.367
000
lr 1.700000e-07 reg 2.750000e+04 train accuracy: 0.362408 val accuracy: 0.369
000
lr 1.800000e-07 reg 1.000000e+04 train accuracy: 0.361449 val accuracy: 0.371
000
lr 1.800000e-07 reg 1.250000e+04 train accuracy: 0.371857 val accuracy: 0.384
000
lr 1.800000e-07 reg 1.500000e+04 train accuracy: 0.364347 val accuracy: 0.372
000
lr 1.800000e-07 reg 1.750000e+04 train accuracy: 0.371959 val accuracy: 0.378
000
lr 1.800000e-07 reg 2.000000e+04 train accuracy: 0.360122 val accuracy: 0.360
```

```
000
lr 1.800000e-07 reg 2.250000e+04 train accuracy: 0.361633 val accuracy: 0.378
000
lr 1.800000e-07 reg 2.500000e+04 train accuracy: 0.360082 val accuracy: 0.369
000
lr 1.800000e-07 reg 2.750000e+04 train accuracy: 0.363286 val accuracy: 0.363
000
lr 1.900000e-07 reg 1.000000e+04 train accuracy: 0.362837 val accuracy: 0.387
000
lr 1.900000e-07 reg 1.250000e+04 train accuracy: 0.367571 val accuracy: 0.393
000
lr 1.900000e-07 reg 1.500000e+04 train accuracy: 0.369755 val accuracy: 0.372
000
lr 1.900000e-07 reg 1.750000e+04 train accuracy: 0.371367 val accuracy: 0.382
000
lr 1.900000e-07 reg 2.000000e+04 train accuracy: 0.364592 val accuracy: 0.366
000
lr 1.900000e-07 reg 2.250000e+04 train accuracy: 0.365510 val accuracy: 0.359
000
lr 1.900000e-07 reg 2.500000e+04 train accuracy: 0.366653 val accuracy: 0.366
000
lr 1.900000e-07 reg 2.750000e+04 train accuracy: 0.360102 val accuracy: 0.379
000
best validation accuracy achieved during cross-validation: 0.393000
```

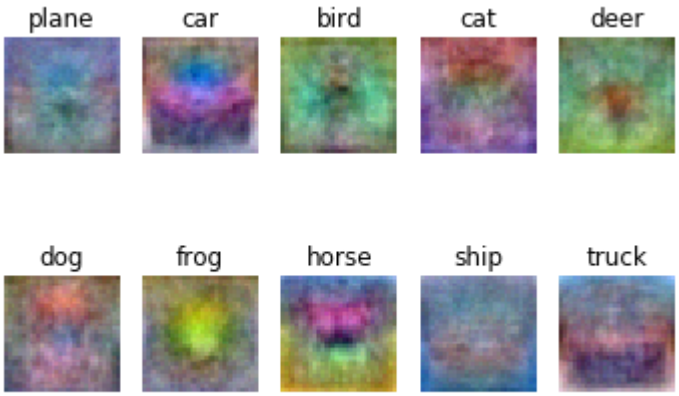
In [224]:
```
# Visualize the cross-validation results
import math
import pdb

# 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.coolwarm)
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.coolwarm)
plt.colorbar()
plt.xlabel('log learning rate')
plt.ylabel('log regularization strength')
plt.title('CIFAR-10 validation accuracy')
plt.show()
```



In [225]:
```
# Evaluate the best svm on test set
y_test_pred = best_svm.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
```

```
linear SVM on raw pixels final test set accuracy: 0.372000
```

In [222]:
```
# Visualize the learned weights for each class.
# Depending on your choice of learning rate and regularization strength, these
may
# or may not be nice to look at.
w = best_svm.W[:-1,:] # strip out the bias
w = w.reshape(32, 32, 3, 10)
w_min, w_max = np.min(w), np.max(w)
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'shi
p', '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 SVM weights look like a very broad template for the class. Lets take an example. For **deer**, you can see a green background, likely because most pictures of deer have grass or mountains in the background. You can also see a brown triangular shape in the middle, and what seem to be antlers. This must be because there's enough pictures of deer facing the front that antlers can be made out.

Each template represents the 'average' image for that class.

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 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-ipyt
hon
%load_ext autoreload
%autoreload 2
```

In [8]:
```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000, n
um_dev=500):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    it for the linear classifier. These are the same steps as we used for the
    SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = '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)

    # subsample the data
    mask = list(range(num_training, num_training + num_validation))
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = list(range(num_training))
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = list(range(num_test))
    X_test = X_test[mask]
    y_test = y_test[mask]
    mask = np.random.choice(num_training, num_dev, replace=False)
    X_dev = X_train[mask]
    y_dev = y_train[mask]

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

    # Normalize the data: subtract the mean image
    mean_image = np.mean(X_train, axis = 0)
    X_train -= mean_image
    X_val -= mean_image
    X_test -= mean_image
    X_dev -= mean_image

    # add bias dimension and transform into columns
    X_train = np.hstack([X_train, np.ones((X_train.shape[0], 1))])
    X_val = np.hstack([X_val, np.ones((X_val.shape[0], 1))])
    X_test = np.hstack([X_test, np.ones((X_test.shape[0], 1))])
    X_dev = np.hstack([X_dev, np.ones((X_dev.shape[0], 1))])

    return X_train, y_train, X_val, y_val, X_test, y_test, X_dev, y_dev
```

```
# Invoke the above function to get our data.
X_train, y_train, X_val, y_val, X_test, y_test, X_dev, y_dev = get_CIFAR10_dat
a()
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 [23]:
```
# 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: 2.340740
sanity check: 2.302585
```

Inline Question 1

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

Your Answer :

The softmax function can be interpreted as a probabilistic function (i.e. what is the probability of this class being correct?). Because we are initializing a completely random matrix of weights, we would expect to be right 10% of the time, as there are 10 classes in our dataset. Therefore, we expect the element inside the log function to be 0.1, and the loss function is computed as -log(0.1).

In [27]:
```
# 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: 1.163003 analytic: 1.163003, relative error: 6.793176e-09
numerical: 0.911954 analytic: 0.911954, relative error: 3.059526e-08
numerical: -1.051291 analytic: -1.051291, relative error: 1.324196e-08
numerical: -0.151287 analytic: -0.151287, relative error: 8.907440e-09
numerical: 1.021437 analytic: 1.021436, relative error: 4.528574e-08
numerical: -0.176680 analytic: -0.176680, relative error: 2.781467e-07
numerical: 3.187457 analytic: 3.187456, relative error: 3.193587e-08
numerical: 1.941594 analytic: 1.941594, relative error: 5.920190e-08
numerical: 0.245621 analytic: 0.245621, relative error: 1.969984e-07
numerical: 1.822653 analytic: 1.822653, relative error: 1.837011e-08
numerical: 4.748783 analytic: 4.748783, relative error: 1.110412e-08
numerical: -0.942401 analytic: -0.942401, relative error: 5.066722e-08
numerical: -0.438947 analytic: -0.438947, relative error: 1.733962e-08
numerical: 0.194306 analytic: 0.194306, relative error: 4.594160e-08
numerical: -0.682544 analytic: -0.682545, relative error: 8.936635e-08
numerical: -0.989251 analytic: -0.989251, relative error: 3.548463e-08
numerical: -0.857313 analytic: -0.857313, relative error: 2.284582e-08
numerical: -1.728127 analytic: -1.728128, relative error: 3.733127e-08
numerical: 1.134523 analytic: 1.134523, relative error: 5.559696e-08
numerical: 4.115487 analytic: 4.115487, relative error: 1.711893e-08
```

In [40]:
```
# 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: 2.340740e+00 computed in 0.079465s
vectorized loss: 2.340740e+00 computed in 0.003337s
Loss difference: 0.000000
Gradient difference: 0.000000
```

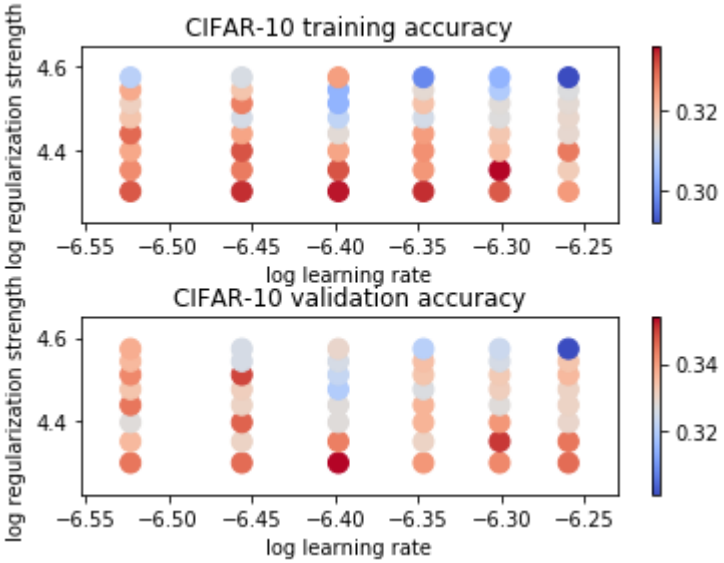
In [60]:
```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of over 0.35 on the validation set.

from cs231n.classifiers import Softmax
results = {}
best_val = -1
best_softmax = None

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

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

learning_rates = np.arange(3,6,0.5) * 1e-7
regularization_strengths = np.arange(2,4,0.25)*1e4

# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
for rate in learning_rates:
    for strength in regularization_strengths:

        # Initializing and training the model
        softmax = Softmax()
        softmax.train(X_train, y_train, learning_rate=rate, reg=strength, num_
iters = 300)

        # Predicting values for training and validations sets
        y_train_pred = softmax.predict(X_train)
        y_val_pred = softmax.predict(X_val)

        # Verifying accuracy in training and validation samples
        train_accuracy = np.mean(y_train == y_train_pred)
        val_accuracy = np.mean(y_val == y_val_pred)

        # Storing results in the dictionary
        results[(rate, strength)] = (train_accuracy, val_accuracy)

        # Storing svm object and accuracy for best performing learning rate an
d regularization strength
        if val_accuracy > best_val:
            best_val = val_accuracy
            best_softmax = softmax
```

```
# *****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_v
al)
```

```
lr 3.000000e-07 reg 2.000000e+04 train accuracy: 0.331653 val accuracy: 0.345
000
lr 3.000000e-07 reg 2.250000e+04 train accuracy: 0.326878 val accuracy: 0.336
000
lr 3.000000e-07 reg 2.500000e+04 train accuracy: 0.323694 val accuracy: 0.328
000
lr 3.000000e-07 reg 2.750000e+04 train accuracy: 0.330082 val accuracy: 0.345
000
lr 3.000000e-07 reg 3.000000e+04 train accuracy: 0.319653 val accuracy: 0.334
000
lr 3.000000e-07 reg 3.250000e+04 train accuracy: 0.317612 val accuracy: 0.343
000
lr 3.000000e-07 reg 3.500000e+04 train accuracy: 0.323286 val accuracy: 0.336
000
lr 3.000000e-07 reg 3.750000e+04 train accuracy: 0.308776 val accuracy: 0.338
000
lr 3.500000e-07 reg 2.000000e+04 train accuracy: 0.334735 val accuracy: 0.346
000
lr 3.500000e-07 reg 2.250000e+04 train accuracy: 0.328367 val accuracy: 0.331
000
lr 3.500000e-07 reg 2.500000e+04 train accuracy: 0.331959 val accuracy: 0.347
000
lr 3.500000e-07 reg 2.750000e+04 train accuracy: 0.324102 val accuracy: 0.331
000
lr 3.500000e-07 reg 3.000000e+04 train accuracy: 0.312694 val accuracy: 0.332
000
lr 3.500000e-07 reg 3.250000e+04 train accuracy: 0.328041 val accuracy: 0.350
000
lr 3.500000e-07 reg 3.500000e+04 train accuracy: 0.319694 val accuracy: 0.326
000
lr 3.500000e-07 reg 3.750000e+04 train accuracy: 0.313122 val accuracy: 0.326
000
lr 4.000000e-07 reg 2.000000e+04 train accuracy: 0.335714 val accuracy: 0.354
000
lr 4.000000e-07 reg 2.250000e+04 train accuracy: 0.331898 val accuracy: 0.344
000
lr 4.000000e-07 reg 2.500000e+04 train accuracy: 0.324061 val accuracy: 0.328
000
lr 4.000000e-07 reg 2.750000e+04 train accuracy: 0.315020 val accuracy: 0.328
000
lr 4.000000e-07 reg 3.000000e+04 train accuracy: 0.309592 val accuracy: 0.320
000
lr 4.000000e-07 reg 3.250000e+04 train accuracy: 0.303551 val accuracy: 0.323
000
lr 4.000000e-07 reg 3.500000e+04 train accuracy: 0.303551 val accuracy: 0.326
000
lr 4.000000e-07 reg 3.750000e+04 train accuracy: 0.324796 val accuracy: 0.330
000
lr 4.500000e-07 reg 2.000000e+04 train accuracy: 0.334735 val accuracy: 0.341
000
lr 4.500000e-07 reg 2.250000e+04 train accuracy: 0.325551 val accuracy: 0.331
000
lr 4.500000e-07 reg 2.500000e+04 train accuracy: 0.326367 val accuracy: 0.337
000
lr 4.500000e-07 reg 2.750000e+04 train accuracy: 0.324918 val accuracy: 0.337
000
lr 4.500000e-07 reg 3.000000e+04 train accuracy: 0.312429 val accuracy: 0.327
```

```
000
lr 4.500000e-07 reg 3.250000e+04 train accuracy: 0.319959 val accuracy: 0.334
000
lr 4.500000e-07 reg 3.500000e+04 train accuracy: 0.315367 val accuracy: 0.335
000
lr 4.500000e-07 reg 3.750000e+04 train accuracy: 0.298000 val accuracy: 0.321
000
lr 5.000000e-07 reg 2.000000e+04 train accuracy: 0.331347 val accuracy: 0.343
000
lr 5.000000e-07 reg 2.250000e+04 train accuracy: 0.336571 val accuracy: 0.351
000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.321122 val accuracy: 0.341
000
lr 5.000000e-07 reg 2.750000e+04 train accuracy: 0.319347 val accuracy: 0.328
000
lr 5.000000e-07 reg 3.000000e+04 train accuracy: 0.314122 val accuracy: 0.332
000
lr 5.000000e-07 reg 3.250000e+04 train accuracy: 0.314673 val accuracy: 0.333
000
lr 5.000000e-07 reg 3.500000e+04 train accuracy: 0.307531 val accuracy: 0.326
000
lr 5.000000e-07 reg 3.750000e+04 train accuracy: 0.303490 val accuracy: 0.324
000
lr 5.500000e-07 reg 2.000000e+04 train accuracy: 0.325204 val accuracy: 0.346
000
lr 5.500000e-07 reg 2.250000e+04 train accuracy: 0.318327 val accuracy: 0.345
000
lr 5.500000e-07 reg 2.500000e+04 train accuracy: 0.328531 val accuracy: 0.330
000
lr 5.500000e-07 reg 2.750000e+04 train accuracy: 0.315796 val accuracy: 0.331
000
lr 5.500000e-07 reg 3.000000e+04 train accuracy: 0.316061 val accuracy: 0.331
000
lr 5.500000e-07 reg 3.250000e+04 train accuracy: 0.315204 val accuracy: 0.336
000
lr 5.500000e-07 reg 3.500000e+04 train accuracy: 0.313837 val accuracy: 0.334
000
lr 5.500000e-07 reg 3.750000e+04 train accuracy: 0.291653 val accuracy: 0.301
000
best validation accuracy achieved during cross-validation: 0.354000
```

In [61]:
```
# 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.coolwarm)
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.coolwarm)
plt.colorbar()
plt.xlabel('log learning rate')
plt.ylabel('log regularization strength')
plt.title('CIFAR-10 validation accuracy')
plt.show()
```



In [62]:
```
# 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: 0.348000
```

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.

$\mathit{Your Answer}$: True

$\mathit{Your Explanation}$: For the overall training loss to remain unchanged, the loss of the additional datapoint must be zero. In SVM, this can happen if the difference between the score of the true class and the score of every other class is greater than the SVM delta. Intuitively, if we are very sure that the new training point belongs to a certain class, the SVM loss for that datapoint would be zero.

In Softmax, for the loss to be zero we would need to have the correct class to have a score of 1 and every other class to have a score of zero. Mathematically, this is not possible. A score of zero is not possible because it would require the logarithmic function to take as input negative infinity. Intuitively, the scores reflect the probability of each class, and it is difficult to imagine that we can know with 100% certainty which class an image belongs to. There will always be at least small probabilities associated to other classes, which would make the loss for that image greater than zero.

In [63]:
```
# 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', 'shi
p', '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 [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 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-ipyt
hon
%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 [4]:
```
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 [6]:
```
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 [11]:
```
from cs231n.gradient_check import eval_numerical_gradient

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

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

# these should all be less than 1e-8 or so
for param_name in grads:
    f = lambda W: net.loss(X, y, reg=0.05)[0]
    param_grad_num = eval_numerical_gradient(f, net.params[param_name], verbose=False)
    print('%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: 4.447625e-11
W1 max relative error: 3.561318e-09
b1 max relative error: 2.738420e-09
```

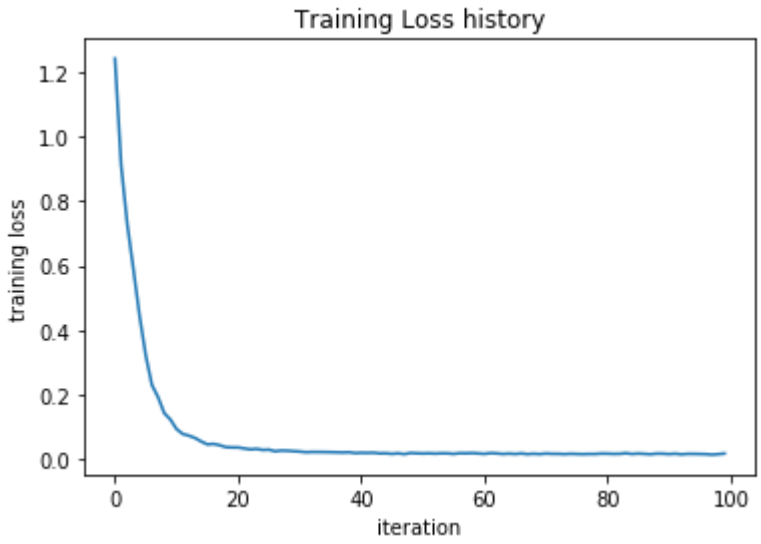
Train the network

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 prepare
    it for the two-layer neural net classifier. These are the same steps as
    we used for the SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = '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)

    # 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 [25]:
```
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 2.302957
iteration 100 / 1000: loss 2.302576
iteration 200 / 1000: loss 2.298401
iteration 300 / 1000: loss 2.263794
iteration 400 / 1000: loss 2.199835
iteration 500 / 1000: loss 2.103390
iteration 600 / 1000: loss 2.033097
iteration 700 / 1000: loss 2.042297
iteration 800 / 1000: loss 2.007256
iteration 900 / 1000: loss 1.976159
Validation accuracy:  0.285
```

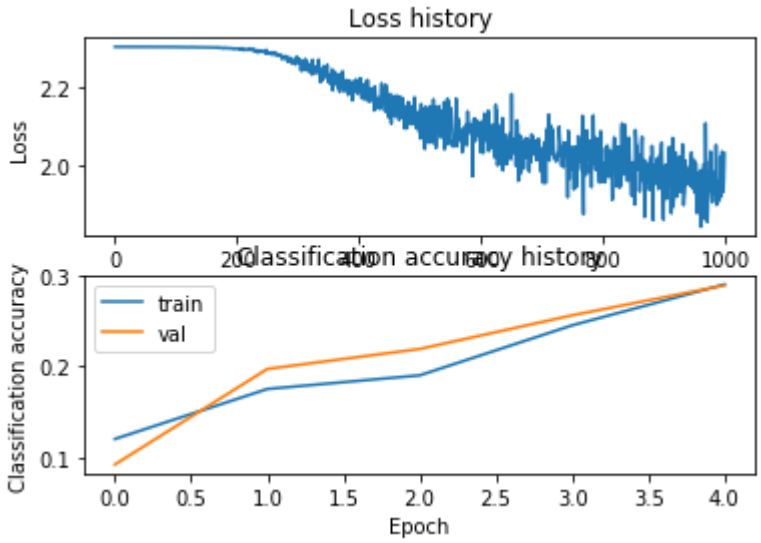
Debug the training

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.

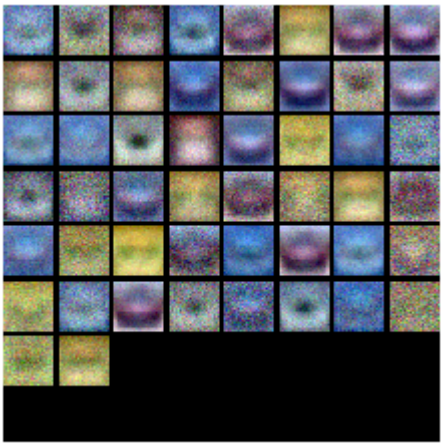
One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

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 [26]:
```
# 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 [27]:
```
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 :

1. First, I increased the learning rate. The linear loss history graph indicated that the learning rate was not large enough. By increasing the learning rate, we're able to obtain an exponential decrease in the loss function.

1. I then decided to increase the hidden layer size. A larger hidden layer size would be able to learn more 'templates' for the classes in the layer. The previous parameter of 50 might not be capturing all of the first-order templates in the images, and I found that increasing the size of the layer led to better results. I adjusted this step by visually looking at the grid visualization.

1. I increased the batch size. A larger batch size should lead to smoother learning, and I found that I was able to increase batch size from 200 to 600 with good results and without running into computing time issues.

1. I did not find significant performance alterations by varying the regularization strength around the default value.

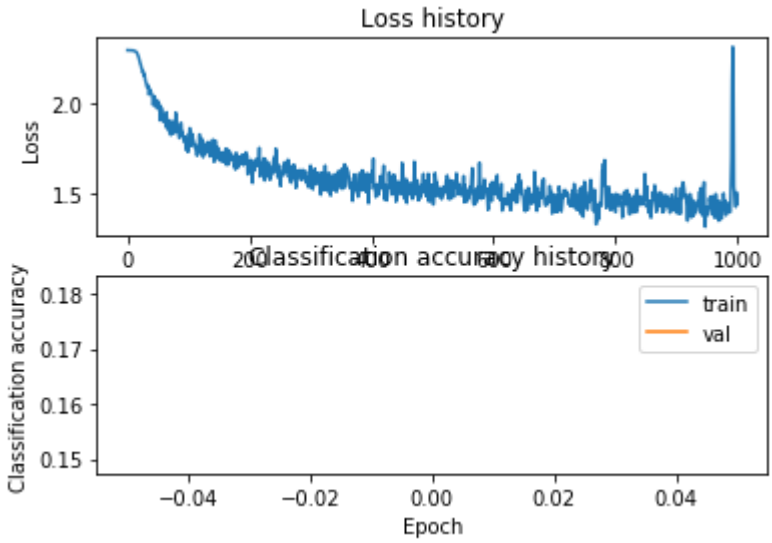
1. By increasing the learning rate, I found spikes in the loss function at over 500 iterations. This could be caused because large step sizes can overshoot in the gradient descent process. Therefore, I adjusted the rate of decay to capture the benefits of the exponential decrease in loss in the first iterations, but reduce spikes in later iterations.

In [172]:
```
best_net = None # store the best model into this
best_val = -1
results = {}

################################################################################
###
# TODO: Tune hyperparameters using the validation set. Store your best trained
#
# model in best_net.
#
#
#
# To help debug your network, it may help to use visualizations similar to the
#
# ones we used above; these visualizations will have significant qualitative
#
# differences from the ones we saw above for the poorly tuned network.
#
#
#
# Tweaking hyperparameters by hand can be fun, but you might find it useful to
#
# write code to sweep through possible combinations of hyperparameters
#
# automatically like we did on the previous exercises.
#
################################################################################
###
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

# Hyperparameters to tune:
#    (1) hidden layer size
#    (2) learning rate
#    (3) number of training epochs
#    (4) regularization strength

input_size = 32 * 32 * 3
num_classes = 10

hidden_size = [80]
learning_rate = [2e-3]
batch_size = [600]
regularization_strength = [0.15, 0.2]

for hsize in hidden_size:
    for lrate in learning_rate:
        for bsize in batch_size:
            for strength in regularization_strength:
                net = TwoLayerNet(input_size, hsize, num_classes)

                stats = net.train(X_train, y_train, X_val, y_val,
                num_iters=1000, batch_size=bsize,
                learning_rate=lrate, learning_rate_decay=0.8,
                reg=strength, verbose=True)

                # Predict on the validation set
```

```
                train_accuracy = (net.predict(X_train) == y_train).mean()
                val_acc = (net.predict(X_val) == y_val).mean()
                print('Validation accuracy: ', val_acc)

                # Storing results in the dictionary
                results[(hsize, lrate, bsize, strength)] = (train_accuracy, va
l_acc)

                if val_acc > best_val:
                    best_val = val_acc
                    best_net = net

for key, value in results.items():
    print(key, value)

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

```
iteration 0 / 1000: loss 2.302971
iteration 100 / 1000: loss 1.750370
iteration 200 / 1000: loss 1.690052
iteration 300 / 1000: loss 1.583977
iteration 400 / 1000: loss 1.531178
iteration 500 / 1000: loss 1.508255
iteration 600 / 1000: loss 1.478004
iteration 700 / 1000: loss 1.441283
iteration 800 / 1000: loss 1.381839
iteration 900 / 1000: loss 1.397324
Validation accuracy:  0.505
iteration 0 / 1000: loss 2.303077
iteration 100 / 1000: loss 1.827552
iteration 200 / 1000: loss 1.644781
iteration 300 / 1000: loss 1.653851
iteration 400 / 1000: loss 1.566276
iteration 500 / 1000: loss 1.448219
iteration 600 / 1000: loss 1.497596
iteration 700 / 1000: loss 1.484926
iteration 800 / 1000: loss 1.391328
iteration 900 / 1000: loss 1.447019
Validation accuracy:  0.506
(80, 0.002, 600, 0.15) (0.5362653061224489, 0.505)
(80, 0.002, 600, 0.2) (0.5311224489795918, 0.506)
```

In [173]:
```
# 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 [174]:
```
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.figure(figsize = (10,10))

    plt.imshow(visualize_grid(W1, padding=3).astype('uint8'))
    plt.gca().axis('off')
    plt.figure(figsize=(200,200))
    plt.show()

show_net_weights(net)
```

```
<Figure size 14400x14400 with 0 Axes>
```

In [175]:
```
# 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:  0.506
```

In [176]:
```
# Visualize the weights of the best network
show_net_weights(best_net)
```

```
<Figure size 14400x14400 with 0 Axes>
```

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 [177]:
```
# 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:  0.526
```

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.

$Your Answer$: 3

$Your Explanation$: Increasing the regularization strength ensures that our model does not overfit on the training data, and therefore the gap between the performance in the training and testing samples should decrease.

Training on a larger dataset won't have a large effect on the performance gap. If the original dataset is small, then having a larger training dataset will also help to avoid overfitting, but this should not be a problem with 50,000 images.

Increasing the number of hidden units will not decrease the performance gap. If the number of hidden units is high and we increase it, then this could also lead to overfitting by over-learning the templates that are available in the training set but might not necessarily reflect the 'average' template for that class.

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 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-ipyt
hon
%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 (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)

    # 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_colo
r_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 [6]:
```
print(X_train_feats.shape)
```

```
(49000, 155)
```

In [32]:
```
# Use the validation set to tune the learning rate and regularization strength

from cs231n.classifiers.linear_classifier import LinearSVM

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

learning_rates = [3e-8, 4e-8, 5e-8]
regularization_strengths = [5e5, 10e5, 15e5, 10e5, 25e5]

for rate in learning_rates:
    for strength in regularization_strengths:

        # Initializing and training the class
        svm = LinearSVM()
        svm.train(X_train_feats, y_train, learning_rate=rate, reg=strength, nu
m_iters = 500)

        # Predicting values for training and validations sets
        y_train_pred = svm.predict(X_train_feats)
        y_val_pred = svm.predict(X_val_feats)

        # Verifying accuracy in training and validation samples
        train_accuracy = np.mean(y_train == y_train_pred)
        val_accuracy = np.mean(y_val == y_val_pred)

        # Storing results in the dictionary
        results[(rate, strength)] = (train_accuracy, val_accuracy)

        # Storing svm object and accuracy for best performing learning rate an
d regularization strength
        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_v
al)
```

```
lr 3.000000e-08 reg 5.000000e+05 train accuracy: 0.410735 val accuracy: 0.404
000
lr 3.000000e-08 reg 1.000000e+06 train accuracy: 0.411551 val accuracy: 0.425
000
lr 3.000000e-08 reg 1.500000e+06 train accuracy: 0.405245 val accuracy: 0.422
000
lr 3.000000e-08 reg 2.500000e+06 train accuracy: 0.409429 val accuracy: 0.408
000
lr 4.000000e-08 reg 5.000000e+05 train accuracy: 0.415061 val accuracy: 0.419
000
lr 4.000000e-08 reg 1.000000e+06 train accuracy: 0.410041 val accuracy: 0.392
000
lr 4.000000e-08 reg 1.500000e+06 train accuracy: 0.403612 val accuracy: 0.405
000
lr 4.000000e-08 reg 2.500000e+06 train accuracy: 0.399959 val accuracy: 0.401
000
lr 5.000000e-08 reg 5.000000e+05 train accuracy: 0.414306 val accuracy: 0.413
000
lr 5.000000e-08 reg 1.000000e+06 train accuracy: 0.413449 val accuracy: 0.415
000
lr 5.000000e-08 reg 1.500000e+06 train accuracy: 0.398449 val accuracy: 0.393
000
lr 5.000000e-08 reg 2.500000e+06 train accuracy: 0.402918 val accuracy: 0.400
000
best validation accuracy achieved during cross-validation: 0.427000
```

In [33]:

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

```
0.421
```

In [21]:
```
# An important way to gain intuition about how an algorithm works is to
# visualize the mistakes that it makes. In this visualization, we show example
s
# 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', 'shi
p', '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?

$YourAnswer:$ They do make sense. Lets remember that the classification is done on two features: color histogram and histogram of oriented gradients. The images that are misclassified are being misclassified under these two dimensions.

For example, the template for planes knows two things: the gradients typically display objects with sharp, triangular edges and they typically exhibit a large amount of blue pixels. If we look at the misclassified images, we can see that they either have blue backgrounds (sky or ocean), or they have images that have sharp gradients, as you would see with a flying plane.

The example can be extended to other classes as well. The images that are misclassified as cats tend to have brown objects, such as horses or deer, but not with green backgrounds. This is probably because there are a large number of brown cats in our training set, and unlike horses or deer, these tend to be indoors with other background colors.

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 [34]:
```
# 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)
```

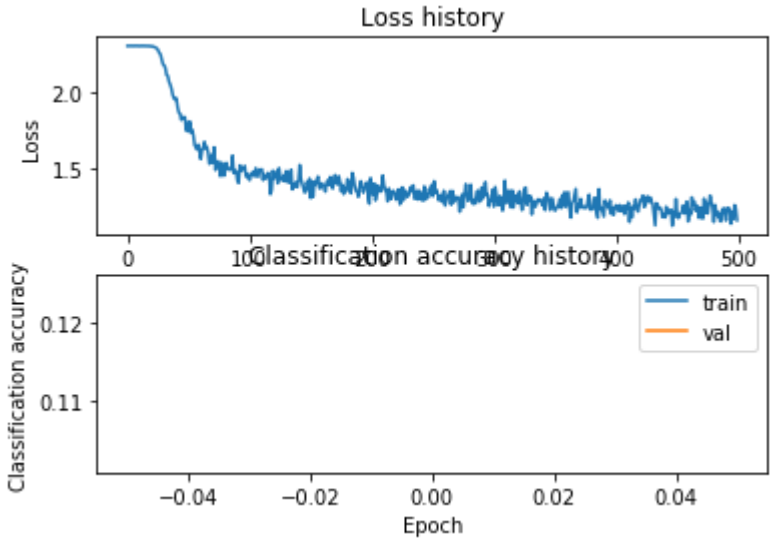
```
(49000, 154)
(49000, 153)
```

In [65]:
```
from cs231n.classifiers.neural_net import TwoLayerNet

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

################################################################################
##
# TODO: Train a two-layer neural network on image features. You may want to
#
# cross-validate various parameters as in previous sections. Store your best
#
# model in the best_net variable.
#
################################################################################
##
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
results = {}

best_net = None
best_val = -1

num_classes = 10

hidden_size = [500]
learning_rate = [0.4, 0.5, 0.6, 0.7]
batch_size = [600]
regularization_strength = [1e-3, 1e-4]
# regularization_strength = [0.15, 0.2, 5e5, 10e5, 15e5]

for hsize in hidden_size:
    for lrate in learning_rate:
        for bsize in batch_size:
            for strength in regularization_strength:
                net = TwoLayerNet(input_dim, hidden_dim, num_classes)

                stats = net.train(X_train_feats, y_train, X_val_feats, y_val,
                num_iters=500, batch_size=bsize,
                learning_rate=lrate, learning_rate_decay=0.95,
                reg=strength, verbose=True)

                # Predict on the validation set
                train_accuracy = (net.predict(X_train_feats) == y_train).mean
()
                val_acc = (net.predict(X_val_feats) == y_val).mean()
                print('Validation accuracy: ', val_acc)

                # Storing results in the dictionary
                results[(hsize, lrate, bsize, strength)] = (train_accuracy, va
l_acc)

                if val_acc > best_val:
                    best_val = val_acc
                    best_net = net
                    best_stats = stats
```

```
for key, value in results.items():
    print(key, value)

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

```
iteration 0 / 500: loss 2.302586
iteration 100 / 500: loss 1.638240
iteration 200 / 500: loss 1.434881
iteration 300 / 500: loss 1.318567
iteration 400 / 500: loss 1.422102
Validation accuracy:  0.54
iteration 0 / 500: loss 2.302585
iteration 100 / 500: loss 1.601356
iteration 200 / 500: loss 1.427459
iteration 300 / 500: loss 1.342795
iteration 400 / 500: loss 1.275326
Validation accuracy:  0.544
iteration 0 / 500: loss 2.302586
iteration 100 / 500: loss 1.521489
iteration 200 / 500: loss 1.346990
iteration 300 / 500: loss 1.259497
iteration 400 / 500: loss 1.339567
Validation accuracy:  0.542
iteration 0 / 500: loss 2.302585
iteration 100 / 500: loss 1.506749
iteration 200 / 500: loss 1.344090
iteration 300 / 500: loss 1.268000
iteration 400 / 500: loss 1.282829
Validation accuracy:  0.536
iteration 0 / 500: loss 2.302586
iteration 100 / 500: loss 1.482887
iteration 200 / 500: loss 1.420933
iteration 300 / 500: loss 1.286386
iteration 400 / 500: loss 1.261588
Validation accuracy:  0.541
iteration 0 / 500: loss 2.302585
iteration 100 / 500: loss 1.464681
iteration 200 / 500: loss 1.358397
iteration 300 / 500: loss 1.240861
iteration 400 / 500: loss 1.224083
Validation accuracy:  0.548
iteration 0 / 500: loss 2.302586
iteration 100 / 500: loss 1.421802
iteration 200 / 500: loss 1.345527
iteration 300 / 500: loss 1.247979
iteration 400 / 500: loss 1.257032
Validation accuracy:  0.568
iteration 0 / 500: loss 2.302585
iteration 100 / 500: loss 1.418466
iteration 200 / 500: loss 1.295206
iteration 300 / 500: loss 1.199321
iteration 400 / 500: loss 1.149484
Validation accuracy:  0.567
(500, 0.4, 600, 0.001) (0.563734693877551, 0.54)
(500, 0.4, 600, 0.0001) (0.568204081632653, 0.544)
(500, 0.5, 600, 0.001) (0.5736938775510204, 0.542)
(500, 0.5, 600, 0.0001) (0.5777142857142857, 0.536)
(500, 0.6, 600, 0.001) (0.5800408163265306, 0.541)
(500, 0.6, 600, 0.0001) (0.5930408163265306, 0.548)
(500, 0.7, 600, 0.001) (0.6075510204081632, 0.568)
(500, 0.7, 600, 0.0001) (0.6135918367346939, 0.567)
```

In [66]:
```
# Plot the loss function and train / validation accuracies
plt.subplot(2, 1, 1)
plt.plot(best_stats['loss_history'])
plt.title('Loss history')
plt.xlabel('Iteration')
plt.ylabel('Loss')

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



In [67]:
```
# 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)
```

```
0.563
```