k-Nearest Neighbor (kNN) exercise

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

The kNN classifier consists of two stages:

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

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

In [1]:

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

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

from __future__ import print_function

# This is a bit of magic to make matplotlib figures appear inline in the noteboo
k
# 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-ipytho
n
%load_ext autoreload
%autoreload 2
```

In [2]:

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

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

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

# As a sanity check, we print out the size of the training and test data.
print('Training data shape: ', X_train.shape)
print('Training labels shape: ', y_train.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

```
Training data shape:  (50000, 32, 32, 3)
Training labels shape:  (50000,)
Test data shape:  (10000, 32, 32, 3)
Test labels shape:  (10000,)
```

In [3]:

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

In [4]:

```
# Subsample the data for more efficient code execution in this exercise
num_training = 5000
mask = list(range(num_training))
X_train = X_train[mask]
y_train = y_train[mask]
num_test = 500
mask = list(range(num_test))
X_test = X_test[mask]
y_test = y_test[mask]
```

In [5]:

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

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

In [6]:

```
from cs231n.classifiers import KNearestNeighbor

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

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

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

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

First, open `cs231n/classifiers/k_nearest_neighbor.py` and implement the function `compute_distances_two_loops` that uses a (very inefficient) double loop over all pairs of (test, train) examples and computes the distance matrix one element at a time.

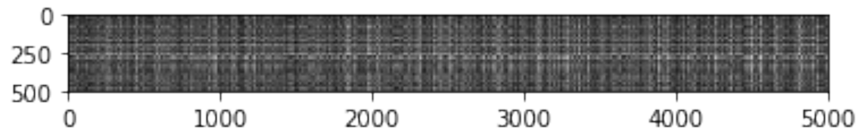
In [7]:

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

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

```
(500, 5000)
```

In [8]:

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



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

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

Your Answer:

A distinctly bright row indicates that this test example is disimilar (has large distance) to examples in the train set. A distinctly bright column indicates that this training example is dissimilar to test examples.

In [9]:

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

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

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

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

In [10]:

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

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

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

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

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

Your Answer: 1

Your explanation: let x1, x2 be the original vectors. L1 distance is then |x1 - x2| let m be the mean; |(x1 - m) - (x2 - m)| = |x1 - x2| let s be the std; |(x1 - m)/s - (x2 - m)/s| = |(x1 - x2)/s| let r be rotation matrix; x1_new and x2_new can then become perpendicular to x1 - x2

In [11]:

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

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

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

In [12]:

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

In [13]:

```
# check that the distance matrix agrees with the one we computed before:
difference = np.linalg.norm(dists - dists_two, ord='fro')
print('Difference was: %f' % (difference, ))
if difference < 0.001:
    print('Good! The distance matrices are the same')
else:
    print('Uh-oh! The distance matrices are different')
```

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

In [14]:

```
# 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 impl
ementation
```

```
Two loop version took 34.598362 seconds
One loop version took 90.215767 seconds
No loop version took 0.241169 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 [15]:

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

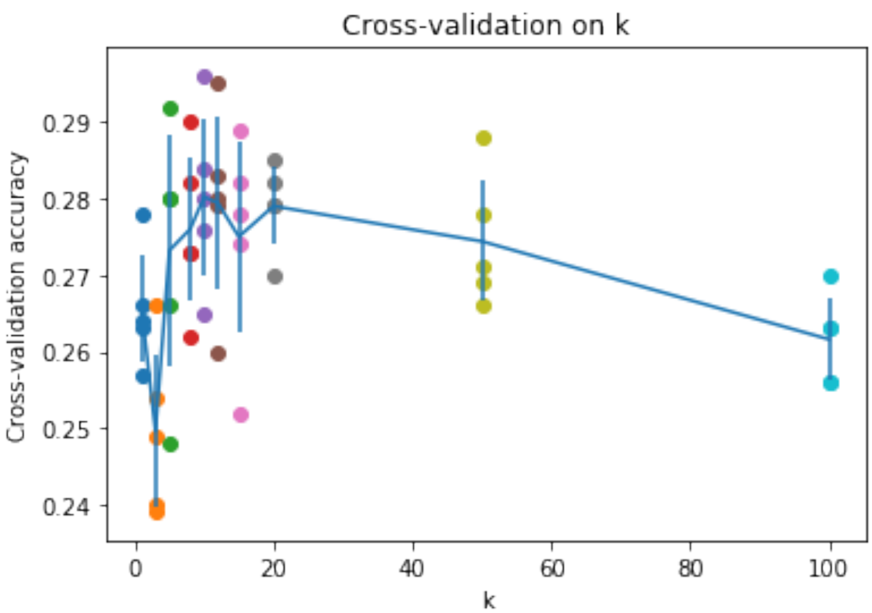
```python
X_train_folds = []
y_train_folds = []
################################################################################
# TODO:                                                                        #
# Split up the training data into folds. After splitting, X_train_folds and    #
# y_train_folds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

# A dictionary holding the accuracies for different values of k that we 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 = {}

# X_train, y_train, X_test, y_test
################################################################################
# TODO:                                                                        #
# Perform k-fold cross validation to find the best value of k. For each        #
# possible value of k, run the k-nearest-neighbor algorithm num_folds times,   #
# where in each case you use all but one of the folds as training data and the #
# last fold as a validation set. Store the accuracies for all fold and all     #
# values of k in the k_to_accuracies dictionary.                               #
################################################################################
for i in range(num_folds):
    X_train_i = np.concatenate(X_train_folds[:i] + X_train_folds[i+1:])
    y_train_i = np.concatenate(y_train_folds[:i] + y_train_folds[i+1:])

    X_test_i = X_train_folds[i]
    y_test_i = y_train_folds[i]

    for k in k_choices:
        classifier.train(X_train_i, y_train_i)
        y_test_pred = classifier.predict(X_test_i, k)
        num_correct = np.sum(y_test_pred == y_test_i)
        accuracy = float(num_correct) / len(y_test_i)
        #print('Got %d / %d correct => accuracy: %f' % (num_correct, len(y_test_i), accuracy))
        curr_dict = k_to_accuracies.get(k, [])
        curr_dict.append(accuracy)
        k_to_accuracies[k] = curr_dict

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

```
# Print out the computed accuracies

for k in sorted(k_to_accuracies):
    for accuracy in k_to_accuracies[k]:
        print('k = %d, accuracy = %f' % (k, accuracy))
```

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

In [16]:

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

# plot the trend line with error bars that correspond to standard deviation
accuracies_mean = np.array([np.mean(v) for k,v in sorted(k_to_accuracies.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 [18]:

```
# Based on the cross-validation results above, choose the best value for k,
# retrain the classifier using all the training data, and test it on the test
# data. You should be able to get above 28% accuracy on the test data.
best_k = 6

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

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

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

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

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

Your Answer: 1, 4

Your explanation: Training error of 1-NN is 0; the closest neighbor is always itself. For testing on the other hand, we introduce new examples that may be labeled better by considering more data. Since at test time KNN must calcualate distance to all training examples, time to classify test grows with train set size. KNN decision boundaries are arbitrary depending on distribution of data, most often not linear.

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

from __future__ import print_function

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

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

CIFAR-10 Data Loading and Preprocessing

In [2]:

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

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

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

# As a sanity check, we print out the size of the training and test data.
print('Training data shape: ', X_train.shape)
print('Training labels shape: ', y_train.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

```
Training data shape:  (50000, 32, 32, 3)
Training labels shape:  (50000,)
Test data shape:  (10000, 32, 32, 3)
Test labels shape:  (10000,)
```

In [3]:

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

In [4]:

```
# Split the data into train, val, and test sets. In addition we will
# create a small development set as a subset of the training data;
# we can use this for development so our code runs faster.
num_training = 49000
num_validation = 1000
num_test = 1000
num_dev = 500

# Our validation set will be num_validation points from the original
# training set.
mask = range(num_training, num_training + num_validation)
X_val = X_train[mask]
y_val = y_train[mask]

# Our training set will be the first num_train points from the original
# training set.
mask = range(num_training)
X_train = X_train[mask]
y_train = y_train[mask]

# We will also make a development set, which is a small subset of
# the training set.
mask = np.random.choice(num_training, num_dev, replace=False)
X_dev = X_train[mask]
y_dev = y_train[mask]

# We use the first num_test points of the original test set as our
# test set.
mask = range(num_test)
X_test = X_test[mask]
y_test = y_test[mask]

print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

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

In [5]:

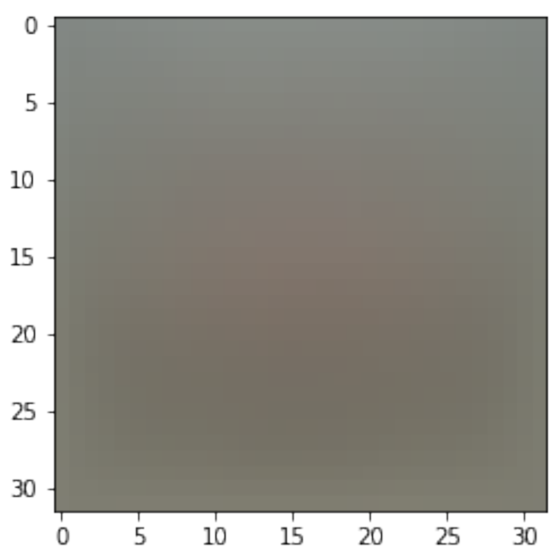
```
# Preprocessing: reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_val = np.reshape(X_val, (X_val.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

# 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 [6]:

```
# 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 i
mage
plt.show()
```

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



In [7]:

```
# second: subtract the mean image from train and test data
X_train -= mean_image
X_val -= mean_image
X_test -= mean_image
X_dev -= mean_image
```

In [8]:

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

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

```
(49000, 3073) (1000, 3073) (1000, 3073) (500, 3073)
```

SVM Classifier

Your code for this section will all be written inside **cs231n/classifiers/linear_svm.py**.

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

In [9]:

```
# 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: 8.828588
```

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 [10]:

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

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

# Numerically compute the gradient along several randomly chosen dimensions, and
# compare them with your analytically computed gradient. The numbers should matc
h
# 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)
print("****")

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

```
numerical: 8.951727 analytic: 8.951727, relative error: 4.156012e-11
numerical: 20.910217 analytic: 20.863606, relative error: 1.115787e-
03
numerical: 30.920956 analytic: 30.970534, relative error: 8.010515e-
04
numerical: 9.897412 analytic: 9.897412, relative error: 6.348008e-12
numerical: 17.382492 analytic: 17.382492, relative error: 2.776074e-
11
numerical: -7.964577 analytic: -8.001215, relative error: 2.294777e-
03
numerical: 10.839305 analytic: 10.839305, relative error: 2.566910e-
11
numerical: -23.446868 analytic: -23.446868, relative error: 7.532532
e-12
numerical: -11.284455 analytic: -11.284455, relative error: 1.187367
e-11
numerical: 19.946680 analytic: 19.946680, relative error: 1.960008e-
11
****
numerical: -4.493948 analytic: -4.524749, relative error: 3.415241e-
03
numerical: 7.383087 analytic: 7.383087, relative error: 2.509571e-11
numerical: 17.474259 analytic: 17.474259, relative error: 2.633592e-
12
numerical: -4.830305 analytic: -4.830305, relative error: 8.046636e-
11
numerical: -7.765563 analytic: -7.765563, relative error: 6.169870e-
11
numerical: 6.656028 analytic: 6.656028, relative error: 3.426538e-11
numerical: 24.989061 analytic: 24.952233, relative error: 7.374411e-
04
numerical: -9.231988 analytic: -9.243166, relative error: 6.050103e-
04
numerical: 21.204751 analytic: 21.224611, relative error: 4.680770e-
04
numerical: 0.307938 analytic: 0.307938, relative error: 9.428135e-10
```

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: Discrepancy can be caused by an incorrect numerical gradient due to the loss function being indifferentiable and hence taking the difference across such a boundary is invalid, such as at 0. Changing the margin would change how often / where we switch from no loss to loss; the effect on frequency is unclear (depends on how often the difference betwen class scores hits this new boundary).

In [11]:

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

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

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

```
Naive loss: 8.828588e+00 computed in 0.085618s
Vectorized loss: 8.828588e+00 computed in 0.027589s
difference: -0.000000
```

In [12]:

```
# 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 0.081041s
Vectorized loss and gradient: computed in 0.011492s
difference: 0.000000
```

Stochastic Gradient Descent

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

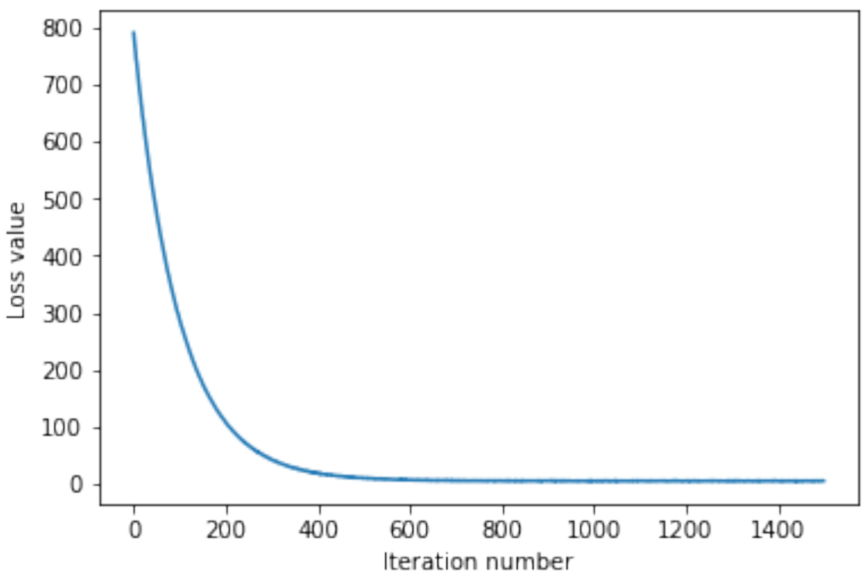
In [13]:

```
# 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 789.930274
iteration 100 / 1500: loss 286.482087
iteration 200 / 1500: loss 108.001109
iteration 300 / 1500: loss 42.434364
iteration 400 / 1500: loss 18.525659
iteration 500 / 1500: loss 10.049786
iteration 600 / 1500: loss 7.151333
iteration 700 / 1500: loss 5.557743
iteration 800 / 1500: loss 5.563516
iteration 900 / 1500: loss 5.457811
iteration 1000 / 1500: loss 5.429021
iteration 1100 / 1500: loss 5.470428
iteration 1200 / 1500: loss 5.677682
iteration 1300 / 1500: loss 5.349342
iteration 1400 / 1500: loss 5.057903
That took 8.804816s
```

In [14]:

```
# 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 [15]:

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

```
training accuracy: 0.371061
validation accuracy: 0.380000
```

In [19]:

```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of about 0.4 on the validation set.
learning_rates = [1e-7, 2e-7, 4e-7]
regularization_strengths = [1e4, 2e4, 3e4]

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

```
.

best_params = None

################################################################################
# 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.                                      #
################################################################################
for l in learning_rates:
    for r in regularization_strengths:
        svm = LinearSVM()
        curr_loss = svm.train(X_train, y_train, learning_rate=l, reg=r,
                              num_iters=1500, verbose=True)
        y_train_pred = svm.predict(X_train)
        y_train_acc = np.mean(y_train == y_train_pred)
        y_val_pred = svm.predict(X_val)
        y_val_acc = np.mean(y_val == y_val_pred)
        results[(l, r)] = (y_train_acc, y_val_acc)
        if y_val_acc > best_val:
            best_svm = svm
            best_val = y_val_acc
            best_params = (l, r)
################################################################################
#                              END OF YOUR CODE                                #
################################################################################

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

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

```
iteration 0 / 1500: loss 329.812881
iteration 100 / 1500: loss 213.884048
iteration 200 / 1500: loss 143.058440
iteration 300 / 1500: loss 96.680817
iteration 400 / 1500: loss 66.274272
iteration 500 / 1500: loss 45.652302
iteration 600 / 1500: loss 32.018373
iteration 700 / 1500: loss 22.933892
```

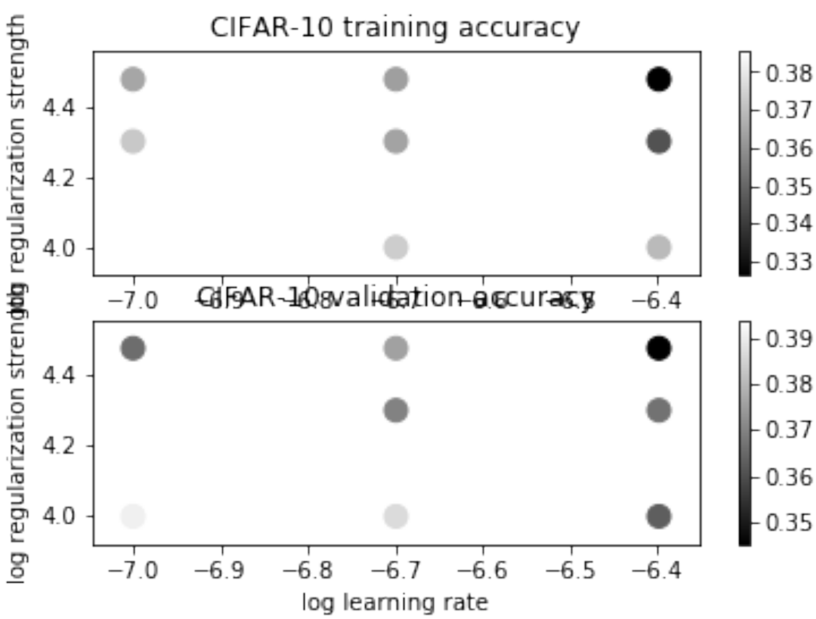
```
iteration 800 / 1500: loss 16.960706
iteration 900 / 1500: loss 13.204388
iteration 1000 / 1500: loss 9.905237
iteration 1100 / 1500: loss 8.624346
iteration 1200 / 1500: loss 7.805575
iteration 1300 / 1500: loss 6.477198
iteration 1400 / 1500: loss 5.921364
iteration 0 / 1500: loss 641.142059
iteration 100 / 1500: loss 284.166304
iteration 200 / 1500: loss 129.317181
iteration 300 / 1500: loss 60.885383
iteration 400 / 1500: loss 29.764385
iteration 500 / 1500: loss 16.033179
iteration 600 / 1500: loss 10.002331
iteration 700 / 1500: loss 7.221273
iteration 800 / 1500: loss 6.113470
iteration 900 / 1500: loss 5.898611
iteration 1000 / 1500: loss 5.647968
iteration 1100 / 1500: loss 4.748945
iteration 1200 / 1500: loss 5.250188
iteration 1300 / 1500: loss 5.457525
iteration 1400 / 1500: loss 5.639028
iteration 0 / 1500: loss 939.636591
iteration 100 / 1500: loss 281.393841
iteration 200 / 1500: loss 87.032507
iteration 300 / 1500: loss 30.018033
iteration 400 / 1500: loss 12.058377
iteration 500 / 1500: loss 7.707717
iteration 600 / 1500: loss 6.032632
iteration 700 / 1500: loss 5.942680
iteration 800 / 1500: loss 6.080024
iteration 900 / 1500: loss 5.455763
iteration 1000 / 1500: loss 5.577070
iteration 1100 / 1500: loss 5.593244
iteration 1200 / 1500: loss 5.278550
iteration 1300 / 1500: loss 5.370906
iteration 1400 / 1500: loss 5.448102
iteration 0 / 1500: loss 334.880441
iteration 100 / 1500: loss 145.749133
iteration 200 / 1500: loss 67.019249
iteration 300 / 1500: loss 31.935991
iteration 400 / 1500: loss 16.874266
iteration 500 / 1500: loss 10.518698
iteration 600 / 1500: loss 7.668947
iteration 700 / 1500: loss 6.304107
iteration 800 / 1500: loss 6.116000
iteration 900 / 1500: loss 5.828414
iteration 1000 / 1500: loss 5.485808
iteration 1100 / 1500: loss 4.864327
iteration 1200 / 1500: loss 5.101638
iteration 1300 / 1500: loss 5.231340
iteration 1400 / 1500: loss 5.233736
iteration 0 / 1500: loss 647.722236
```

```
iteration 100 / 1500: loss 129.030187
iteration 200 / 1500: loss 29.647306
iteration 300 / 1500: loss 10.381281
iteration 400 / 1500: loss 6.229066
iteration 500 / 1500: loss 5.287628
iteration 600 / 1500: loss 5.034977
iteration 700 / 1500: loss 5.086883
iteration 800 / 1500: loss 4.900851
iteration 900 / 1500: loss 5.071654
iteration 1000 / 1500: loss 4.995439
iteration 1100 / 1500: loss 5.349900
iteration 1200 / 1500: loss 4.650375
iteration 1300 / 1500: loss 5.032852
iteration 1400 / 1500: loss 6.045765
iteration 0 / 1500: loss 943.513361
iteration 100 / 1500: loss 87.700810
iteration 200 / 1500: loss 12.691915
iteration 300 / 1500: loss 6.496296
iteration 400 / 1500: loss 5.773099
iteration 500 / 1500: loss 5.794557
iteration 600 / 1500: loss 6.278239
iteration 700 / 1500: loss 5.329029
iteration 800 / 1500: loss 5.808075
iteration 900 / 1500: loss 5.530021
iteration 1000 / 1500: loss 5.997961
iteration 1100 / 1500: loss 5.569494
iteration 1200 / 1500: loss 5.711300
iteration 1300 / 1500: loss 5.374030
iteration 1400 / 1500: loss 5.113521
iteration 0 / 1500: loss 333.361183
iteration 100 / 1500: loss 67.944055
iteration 200 / 1500: loss 17.117683
iteration 300 / 1500: loss 7.442095
iteration 400 / 1500: loss 5.785281
iteration 500 / 1500: loss 5.392824
iteration 600 / 1500: loss 5.244573
iteration 700 / 1500: loss 4.841607
iteration 800 / 1500: loss 5.774334
iteration 900 / 1500: loss 5.524201
iteration 1000 / 1500: loss 5.364770
iteration 1100 / 1500: loss 4.612733
iteration 1200 / 1500: loss 5.402273
iteration 1300 / 1500: loss 5.708605
iteration 1400 / 1500: loss 4.692609
iteration 0 / 1500: loss 637.681800
iteration 100 / 1500: loss 28.715797
iteration 200 / 1500: loss 6.398879
iteration 300 / 1500: loss 5.323034
iteration 400 / 1500: loss 5.697103
iteration 500 / 1500: loss 5.497378
iteration 600 / 1500: loss 4.910578
iteration 700 / 1500: loss 5.173264
iteration 800 / 1500: loss 6.031431
```

```
iteration 900 / 1500: loss 5.393203
iteration 1000 / 1500: loss 5.647651
iteration 1100 / 1500: loss 5.567541
iteration 1200 / 1500: loss 5.107683
iteration 1300 / 1500: loss 5.211513
iteration 1400 / 1500: loss 5.565286
iteration 0 / 1500: loss 951.727452
iteration 100 / 1500: loss 12.247992
iteration 200 / 1500: loss 5.634766
iteration 300 / 1500: loss 6.143236
iteration 400 / 1500: loss 5.552660
iteration 500 / 1500: loss 5.745416
iteration 600 / 1500: loss 5.421951
iteration 700 / 1500: loss 5.998420
iteration 800 / 1500: loss 5.485592
iteration 900 / 1500: loss 5.410751
iteration 1000 / 1500: loss 5.983546
iteration 1100 / 1500: loss 5.506991
iteration 1200 / 1500: loss 5.668893
iteration 1300 / 1500: loss 5.570115
iteration 1400 / 1500: loss 5.794653
lr 1.000000e-07 reg 1.000000e+04 train accuracy: 0.385469 val accura
cy: 0.391000
lr 1.000000e-07 reg 2.000000e+04 train accuracy: 0.372694 val accura
cy: 0.394000
lr 1.000000e-07 reg 3.000000e+04 train accuracy: 0.364878 val accura
cy: 0.366000
lr 2.000000e-07 reg 1.000000e+04 train accuracy: 0.373816 val accura
cy: 0.387000
lr 2.000000e-07 reg 2.000000e+04 train accuracy: 0.364000 val accura
cy: 0.370000
lr 2.000000e-07 reg 3.000000e+04 train accuracy: 0.363102 val accura
cy: 0.376000
lr 4.000000e-07 reg 1.000000e+04 train accuracy: 0.369327 val accura
cy: 0.363000
lr 4.000000e-07 reg 2.000000e+04 train accuracy: 0.345408 val accura
cy: 0.367000
lr 4.000000e-07 reg 3.000000e+04 train accuracy: 0.326265 val accura
cy: 0.345000
best validation accuracy achieved during cross-validation: 0.394000
(1e-07, 20000.0)
```

In [20]:

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

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

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



In [21]:

```
# 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 [22]:

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

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

Inline question 2:

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

Your answer: The visualized SVM weights look like blurry versions of each of their respective classes. They represent the average pixel values of each class (that maximize dot product with examples of this 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

from __future__ import print_function

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

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

In [2]:

```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000, num
_dev=500):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to 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'

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

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

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

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

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

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


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

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

```
print('Test labels shape: ', y_test.shape)

print('dev data shape: ', X_dev.shape)
print('dev labels shape: ', y_dev.shape)
```

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

Softmax Classifier

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

In [4]:

```
# 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, softmax_loss_vectoriz
ed
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)
#loss, grad = softmax_loss_vectorized(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: 2.337193
sanity check: 2.302585
```

Inline Question 1:

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

Your answer: There are 10 classes, and we initialized the weight matrix randomly s.t. each of the 10 classes should receive approximately the same score. Then softmax(xi) = e^score_yi / sum e^score_j ~= 1/10; softmax loss is average of -log(softmax(xi)).

In [5]:

```
# 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: 0.567849 analytic: 0.567849, relative error: 1.022578e-07
numerical: 1.100079 analytic: 1.100079, relative error: 2.135127e-08
numerical: -2.462063 analytic: -2.462063, relative error: 2.582816e-
09
numerical: 2.206644 analytic: 2.206644, relative error: 1.724082e-08
numerical: 3.106253 analytic: 3.106253, relative error: 1.736512e-08
numerical: -1.265471 analytic: -1.265471, relative error: 1.625567e-
08
numerical: -0.510681 analytic: -0.510681, relative error: 6.360931e-
08
numerical: 1.432267 analytic: 1.432267, relative error: 5.457080e-08
numerical: 0.851139 analytic: 0.851139, relative error: 8.969135e-10
numerical: 1.133305 analytic: 1.133304, relative error: 4.096189e-08
numerical: 1.635839 analytic: 1.635840, relative error: 2.319247e-08
numerical: 0.500283 analytic: 0.500283, relative error: 1.672209e-08
numerical: 1.967745 analytic: 1.967745, relative error: 3.375964e-08
numerical: -1.071423 analytic: -1.071423, relative error: 3.551398e-
08
numerical: -1.091325 analytic: -1.091325, relative error: 5.064061e-
08
numerical: -2.049836 analytic: -2.049836, relative error: 3.272740e-
08
numerical: 0.887626 analytic: 0.887626, relative error: 5.251666e-09
numerical: 2.146585 analytic: 2.146585, relative error: 2.256034e-08
numerical: 2.850963 analytic: 2.850963, relative error: 9.449120e-09
numerical: -0.237759 analytic: -0.237759, relative error: 8.487378e-
08
```

In [6]:

```
# Now that we have a naive implementation of the softmax loss function and its g
radient,
# implement a vectorized version in softmax_loss_vectorized.
# The two versions should compute the same results, but the vectorized version s
hould 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, 0.00
0005)
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: 2.337193e+00 computed in 0.071997s
vectorized loss: 2.337193e+00 computed in 0.022640s
Loss difference: 0.000000
Gradient difference: 0.000000
```

In [15]:

```
# 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
best_params = None
learning_rates = [5e-8, 1e-7, 5e-7]
regularization_strengths = [1e4, 2.5e4, 5e4]

for l in learning_rates:
    for r in regularization_strengths:
        soft = Softmax()
        curr_loss = soft.train(X_train, y_train, learning_rate=l, reg=r,
                      num_iters=1500, verbose=True)
        y_train_pred = soft.predict(X_train)
        y_train_acc = np.mean(y_train == y_train_pred)
        y_val_pred = soft.predict(X_val)
        y_val_acc = np.mean(y_val == y_val_pred)
        results[(l, r)] = (y_train_acc, y_val_acc)
        if y_val_acc > best_val:
            best_softmax = soft
            best_val = y_val_acc
            best_params = (l, r)

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

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

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

```
iteration 0 / 1500: loss 314.378890
iteration 100 / 1500: loss 256.343739
iteration 200 / 1500: loss 210.005278
iteration 300 / 1500: loss 171.735479
iteration 400 / 1500: loss 140.718794
iteration 500 / 1500: loss 115.403647
iteration 600 / 1500: loss 94.755554
iteration 700 / 1500: loss 77.763083
iteration 800 / 1500: loss 63.903976
iteration 900 / 1500: loss 52.885643
```

```
iteration 1000 / 1500: loss 43.416895
iteration 1100 / 1500: loss 35.993682
iteration 1200 / 1500: loss 29.782689
iteration 1300 / 1500: loss 24.691988
iteration 1400 / 1500: loss 20.494863
iteration 0 / 1500: loss 784.983024
iteration 100 / 1500: loss 474.939836
iteration 200 / 1500: loss 288.524443
iteration 300 / 1500: loss 175.513011
iteration 400 / 1500: loss 106.944714
iteration 500 / 1500: loss 65.541919
iteration 600 / 1500: loss 40.464141
iteration 700 / 1500: loss 25.454078
iteration 800 / 1500: loss 16.140550
iteration 900 / 1500: loss 10.592509
iteration 1000 / 1500: loss 7.199630
iteration 1100 / 1500: loss 5.176507
iteration 1200 / 1500: loss 3.956560
iteration 1300 / 1500: loss 3.203671
iteration 1400 / 1500: loss 2.765133
iteration 0 / 1500: loss 1540.514962
iteration 100 / 1500: loss 565.337485
iteration 200 / 1500: loss 208.378826
iteration 300 / 1500: loss 77.735027
iteration 400 / 1500: loss 29.853285
iteration 500 / 1500: loss 12.270621
iteration 600 / 1500: loss 5.854505
iteration 700 / 1500: loss 3.491227
iteration 800 / 1500: loss 2.644319
iteration 900 / 1500: loss 2.365326
iteration 1000 / 1500: loss 2.205371
iteration 1100 / 1500: loss 2.186416
iteration 1200 / 1500: loss 2.182300
iteration 1300 / 1500: loss 2.171551
iteration 1400 / 1500: loss 2.079532
iteration 0 / 1500: loss 310.737400
iteration 100 / 1500: loss 206.861984
iteration 200 / 1500: loss 138.510751
iteration 300 / 1500: loss 93.337344
iteration 400 / 1500: loss 63.009215
iteration 500 / 1500: loss 42.864810
iteration 600 / 1500: loss 29.195303
iteration 700 / 1500: loss 20.230915
iteration 800 / 1500: loss 14.230522
iteration 900 / 1500: loss 10.132090
iteration 1000 / 1500: loss 7.520559
iteration 1100 / 1500: loss 5.596569
iteration 1200 / 1500: loss 4.496602
iteration 1300 / 1500: loss 3.632488
iteration 1400 / 1500: loss 3.178314
iteration 0 / 1500: loss 771.884038
iteration 100 / 1500: loss 283.140275
iteration 200 / 1500: loss 104.975657
```

```
iteration 300 / 1500: loss 39.794251
iteration 400 / 1500: loss 15.861718
iteration 500 / 1500: loss 7.124673
iteration 600 / 1500: loss 3.922526
iteration 700 / 1500: loss 2.728889
iteration 800 / 1500: loss 2.305585
iteration 900 / 1500: loss 2.191911
iteration 1000 / 1500: loss 2.108062
iteration 1100 / 1500: loss 2.040261
iteration 1200 / 1500: loss 2.122895
iteration 1300 / 1500: loss 2.086892
iteration 1400 / 1500: loss 2.070165
iteration 0 / 1500: loss 1569.807557
iteration 100 / 1500: loss 211.542831
iteration 200 / 1500: loss 30.125867
iteration 300 / 1500: loss 5.896750
iteration 400 / 1500: loss 2.632760
iteration 500 / 1500: loss 2.200729
iteration 600 / 1500: loss 2.157806
iteration 700 / 1500: loss 2.166693
iteration 800 / 1500: loss 2.115097
iteration 900 / 1500: loss 2.131973
iteration 1000 / 1500: loss 2.153319
iteration 1100 / 1500: loss 2.122642
iteration 1200 / 1500: loss 2.152296
iteration 1300 / 1500: loss 2.166793
iteration 1400 / 1500: loss 2.107457
iteration 0 / 1500: loss 312.407420
iteration 100 / 1500: loss 42.712501
iteration 200 / 1500: loss 7.417511
iteration 300 / 1500: loss 2.716093
iteration 400 / 1500: loss 2.092821
iteration 500 / 1500: loss 2.029881
iteration 600 / 1500: loss 2.016174
iteration 700 / 1500: loss 2.095447
iteration 800 / 1500: loss 1.936520
iteration 900 / 1500: loss 2.098716
iteration 1000 / 1500: loss 2.050053
iteration 1100 / 1500: loss 1.981984
iteration 1200 / 1500: loss 2.037626
iteration 1300 / 1500: loss 1.978531
iteration 1400 / 1500: loss 1.960792
iteration 0 / 1500: loss 783.468668
iteration 100 / 1500: loss 6.962788
iteration 200 / 1500: loss 2.155492
iteration 300 / 1500: loss 2.089235
iteration 400 / 1500: loss 2.045556
iteration 500 / 1500: loss 2.068149
iteration 600 / 1500: loss 2.066446
iteration 700 / 1500: loss 2.076635
iteration 800 / 1500: loss 2.057205
iteration 900 / 1500: loss 2.138281
iteration 1000 / 1500: loss 2.095712
```

```
iteration 1100 / 1500: loss 2.062105
iteration 1200 / 1500: loss 2.111504
iteration 1300 / 1500: loss 2.067248
iteration 1400 / 1500: loss 2.052028
iteration 0 / 1500: loss 1557.437666
iteration 100 / 1500: loss 2.220458
iteration 200 / 1500: loss 2.141399
iteration 300 / 1500: loss 2.189879
iteration 400 / 1500: loss 2.147661
iteration 500 / 1500: loss 2.144514
iteration 600 / 1500: loss 2.096290
iteration 700 / 1500: loss 2.167907
iteration 800 / 1500: loss 2.161713
iteration 900 / 1500: loss 2.168270
iteration 1000 / 1500: loss 2.180228
iteration 1100 / 1500: loss 2.179335
iteration 1200 / 1500: loss 2.163926
iteration 1300 / 1500: loss 2.086831
iteration 1400 / 1500: loss 2.185150
lr 5.000000e-08 reg 1.000000e+04 train accuracy: 0.297510 val accura
cy: 0.303000
lr 5.000000e-08 reg 2.500000e+04 train accuracy: 0.326735 val accura
cy: 0.331000
lr 5.000000e-08 reg 5.000000e+04 train accuracy: 0.311776 val accura
cy: 0.327000
lr 1.000000e-07 reg 1.000000e+04 train accuracy: 0.353204 val accura
cy: 0.366000
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.329041 val accura
cy: 0.337000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.307714 val accura
cy: 0.329000
lr 5.000000e-07 reg 1.000000e+04 train accuracy: 0.350714 val accura
cy: 0.365000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.325653 val accura
cy: 0.341000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.301816 val accura
cy: 0.315000
best validation accuracy achieved during cross-validation: 0.366000
(1e-07, 10000.0)
```

In [16]:

```
# 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: 0.354000
```

Inline Question - *True or False*

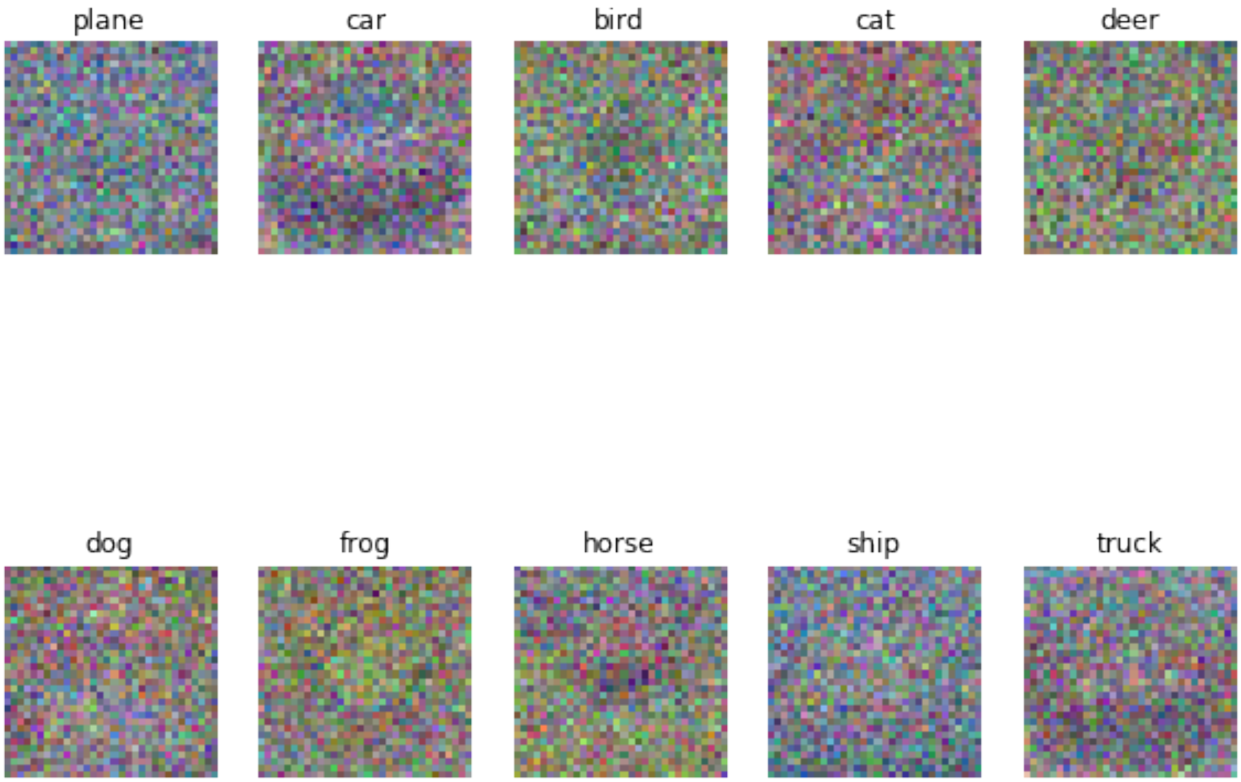
It's possible to add a new datapoint to a training set that would leave the SVM loss unchanged, but this is not the case with the Softmax classifier loss.

Your answer: True

Your explanation: With SVM loss, as long as the new image is predicted with the correct label over all other labels by greater than the margin, there will be an additional loss of 0. With softmax, as we predict the correct class with more confidence the additional loss approaches 0 (e^score_yi --> inf, log(e^score_yi / sum e^score_j --> 1) but never truly reaches.

In [17]:

```
# Visualize the learned weights for each class
w = best_softmax.W[:-1,:] # strip out the bias
w = w.reshape(32, 32, 3, 10)

w_min, w_max = np.min(w), np.max(w)

classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship'
, 'truck']
for i in range(10):
    plt.subplot(2, 5, i + 1)

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


plane
car
bird
cat
deer
dog
frog
horse
ship
truck

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 [8]:

```
# A bit of setup

import numpy as np
import matplotlib.pyplot as plt

from cs231n.classifiers.neural_net import TwoLayerNet
from cs231n.classifiers.softmax import softmax_loss_vectorized

from __future__ import print_function

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

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

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

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

We will use the class `TwoLayerNet` in the file `cs231n/classifiers/neural_net.py` to represent instances of our network. The network parameters are stored in the instance variable `self.params` where keys are string parameter names and values are numpy arrays. Below, we initialize toy data and a toy model that we will use to develop your implementation.

In [9]:

```
# 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 [10]:

```
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 regularizaion loss.

In [11]:

```
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:
1.7963408538435033e-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 [12]:

```
from cs231n.gradient_check import eval_numerical_gradient

# Use numeric gradient checking to check your implementation of the backward pas
s.
# 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, g
rads[param_name])))
```

```
W2 max relative error: 3.440708e-09
b2 max relative error: 3.865112e-11
W1 max relative error: 3.561318e-09
b1 max relative error: 1.555471e-09
```

Train the network

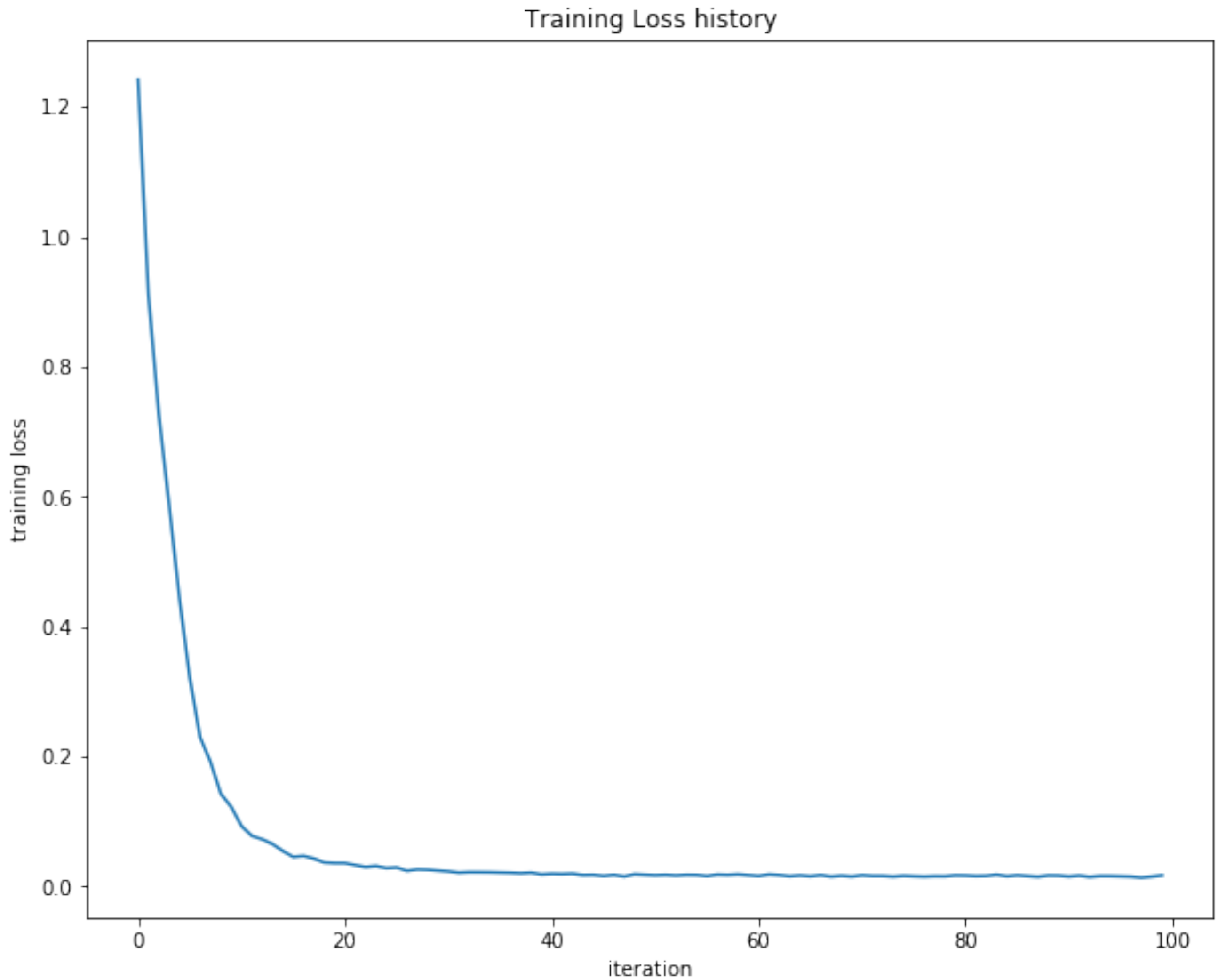
To train the network we will use stochastic gradient descent (SGD), similar to the SVM and Softmax classifiers. Look at the function `TwoLayerNet.train` and fill in the missing sections to implement the training procedure. This should be very similar to the training procedure you used for the SVM and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

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

In [7]:

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

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

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

```
Final training loss:  0.017149607938732093
```



Load the data

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

In [8]:

```python
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'

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

    # Subsample the data
```

```python
    mask = list(range(num_training, num_training + num_validation))

    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = list(range(num_training))
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = list(range(num_test))
    X_test = X_test[mask]
    y_test = y_test[mask]

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

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

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


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

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

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

Train a network

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

In [9]:

```
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
net = TwoLayerNet(input_size, hidden_size, num_classes)

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

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

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

Debug the training

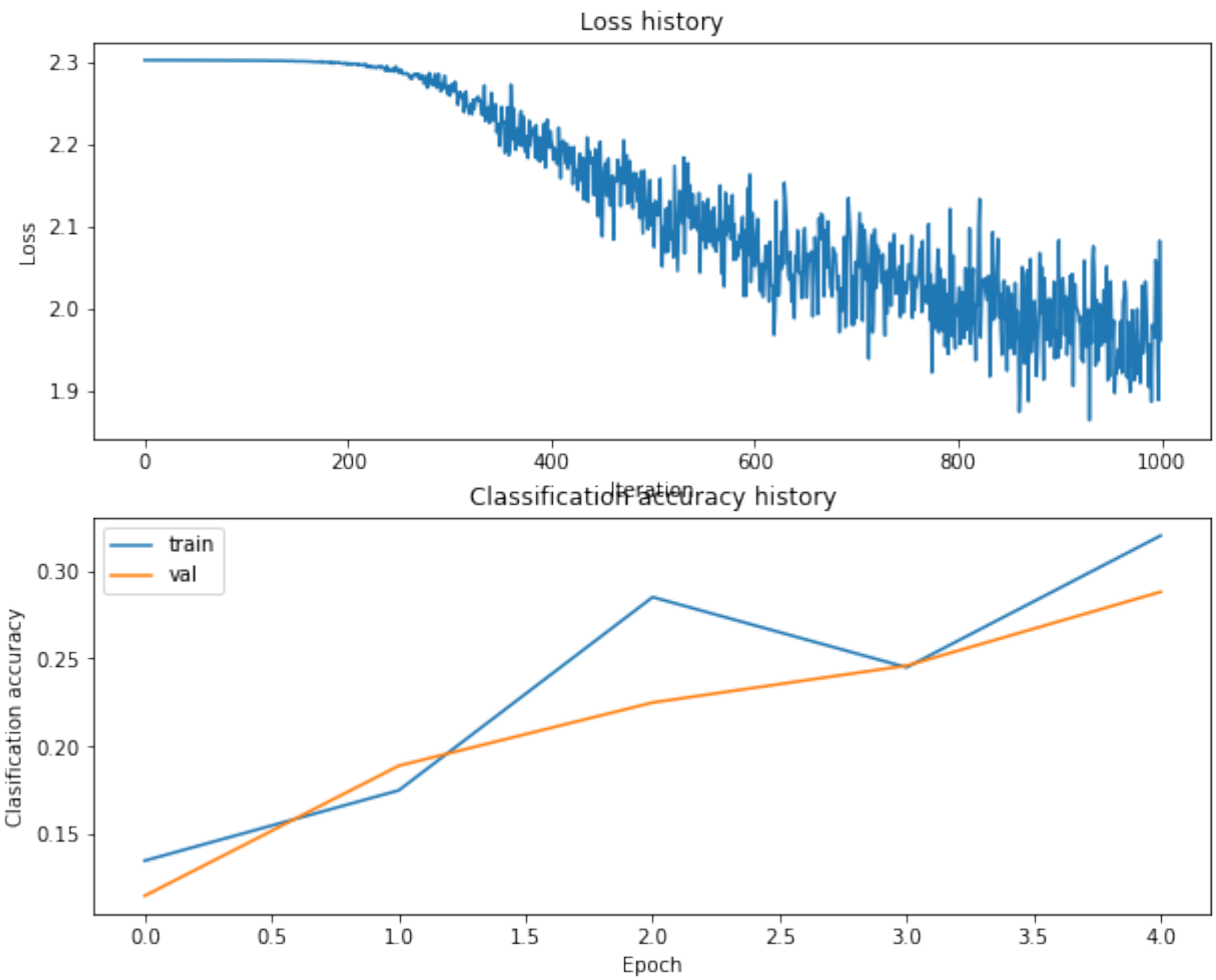
With the default parameters we provided above, you should get a validation accuracy of about 0.29 on the validation set. This isn't very good.

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

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

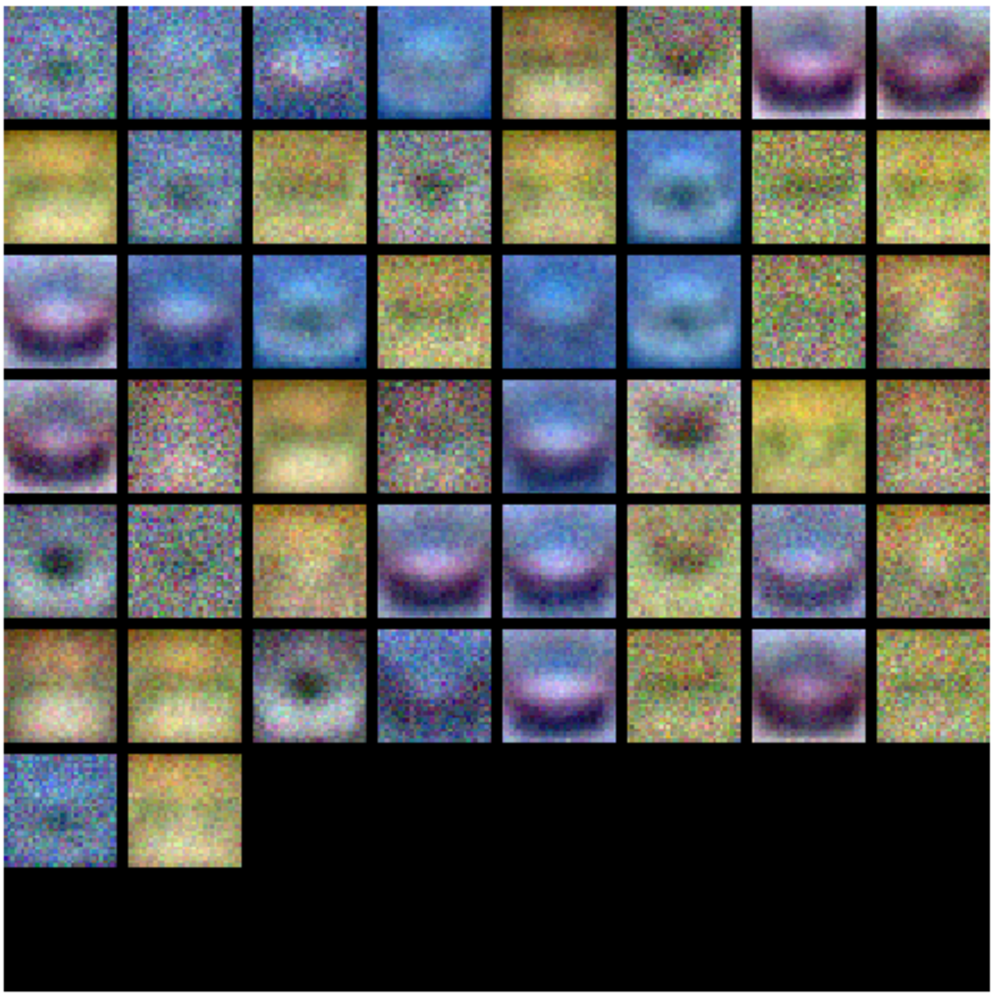
In [10]:

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

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



In [11]:

```
from cs231n.vis_utils import visualize_grid

# Visualize the weights of the network

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

show_net_weights(net)
```



Tune your hyperparameters

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

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

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

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

In [23]:

```
################################################################################
#
# 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.
#
################################################################################
#
# Your code
################################################################################
#
```

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

results = {}
best_val = -1
best_net = None
best_param = None
learning_rates = [1e-3, 8e-4, 6e-4]
regularization_strengths = [0.4, 0.5, 0.6, 0.7, 0.8]

for l in learning_rates:
    for r in regularization_strengths:
        net = TwoLayerNet(input_size, hidden_size, num_classes)
        curr_loss = net.train(X_train, y_train, X_val, y_val, learning_rate=l, r
eg=r, num_iters=1500)
        y_train_pred = net.predict(X_train)
        y_train_acc = np.mean(y_train == y_train_pred)
        #print('training accuracy: %f' % (np.mean(y_train == y_train_pred), ))
        y_val_pred = net.predict(X_val)
        y_val_acc = np.mean(y_val == y_val_pred)
        #print('validation accuracy: %f' % (np.mean(y_val == y_val_pred), ))
        results[(l, r)] = (y_train_acc, y_val_acc)
        if y_val_acc > best_val:
            best_net = net
            best_val = y_val_acc
            best_params = (l, r)


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

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

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

```
lr 6.000000e-04 reg 4.000000e-01 train accuracy: 0.483612 val accura
cy: 0.476000
lr 6.000000e-04 reg 5.000000e-01 train accuracy: 0.481449 val accura
cy: 0.461000
lr 6.000000e-04 reg 6.000000e-01 train accuracy: 0.474714 val accura
cy: 0.455000
lr 6.000000e-04 reg 7.000000e-01 train accuracy: 0.470959 val accura
cy: 0.463000
lr 6.000000e-04 reg 8.000000e-01 train accuracy: 0.471449 val accura
cy: 0.460000
lr 8.000000e-04 reg 4.000000e-01 train accuracy: 0.501959 val accura
cy: 0.493000
lr 8.000000e-04 reg 5.000000e-01 train accuracy: 0.494918 val accura
cy: 0.476000
lr 8.000000e-04 reg 6.000000e-01 train accuracy: 0.491082 val accura
cy: 0.479000
lr 8.000000e-04 reg 7.000000e-01 train accuracy: 0.488184 val accura
cy: 0.465000
lr 8.000000e-04 reg 8.000000e-01 train accuracy: 0.481857 val accura
cy: 0.479000
lr 1.000000e-03 reg 4.000000e-01 train accuracy: 0.500918 val accura
cy: 0.479000
lr 1.000000e-03 reg 5.000000e-01 train accuracy: 0.495286 val accura
cy: 0.476000
lr 1.000000e-03 reg 6.000000e-01 train accuracy: 0.492388 val accura
cy: 0.468000
lr 1.000000e-03 reg 7.000000e-01 train accuracy: 0.491714 val accura
cy: 0.469000
lr 1.000000e-03 reg 8.000000e-01 train accuracy: 0.493122 val accura
cy: 0.468000
best validation accuracy achieved during cross-validation: 0.493000
(0.0008, 0.4)
```

In [24]:

```
# visualize the weights of the best network
show_net_weights(best_net)
```

Run on the test set

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

In [25]:

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

```
Test accuracy:  0.481
```

Inline Question

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

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

Your answer: 1, 3

Your explanation:

Testing accuracy lower than training accuracy means the model is overfitting. More data and regularization decreases the effect of specific datapoints / weights. More hidden units increases the complexity of the model which is more prone to overfit.

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 [26]:

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

from __future__ import print_function

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

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

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

Load data

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

In [27]:

```
from cs231n.features import color_histogram_hsv, hog_feature

def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000):
    # Load the raw CIFAR-10 data
    cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

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

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

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

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

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

```
Clear previously loaded data.
```

Extract Features

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

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

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

In [28]:

```
from cs231n.features import *

# 10, 11,
num_color_bins = 16 # Number of bins in the color histogram
feature_fns = [hog_feature, lambda img: color_histogram_hsv(img, nbin=num_color_
bins)]
X_train_feats = extract_features(X_train, feature_fns, verbose=True)
X_val_feats = extract_features(X_val, feature_fns)
X_test_feats = extract_features(X_test, feature_fns)

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

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

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

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

Train SVM on features

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

In [13]:

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

from cs231n.classifiers.linear_classifier import LinearSVM

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

results = {}
best_val = -1   # The highest validation accuracy that we have seen so far.
best_svm = None # The LinearSVM object that achieved the highest validation rate
.
best_params = 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.       #
################################################################################
for l in learning_rates:
    for r in regularization_strengths:
        svm = LinearSVM()
        curr_loss = svm.train(X_train_feats, y_train, learning_rate=l, reg=r,
                              num_iters=1500, verbose=True)
        y_train_pred = svm.predict(X_train_feats)
        y_train_acc = np.mean(y_train == y_train_pred)
        y_val_pred = svm.predict(X_val_feats)
        y_val_acc = np.mean(y_val == y_val_pred)
        results[(l, r)] = (y_train_acc, y_val_acc)
        if y_val_acc > best_val:
            best_svm = svm
            best_val = y_val_acc
            best_params = (l, r)

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

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

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

iteration 0 / 1500: loss 10.596329

```
iteration 0 / 1500: loss 10.590529
iteration 100 / 1500: loss 10.578521
iteration 200 / 1500: loss 10.546396
iteration 300 / 1500: loss 10.499155
iteration 400 / 1500: loss 10.469077
iteration 500 / 1500: loss 10.447657
iteration 600 / 1500: loss 10.414133
iteration 700 / 1500: loss 10.401073
iteration 800 / 1500: loss 10.364895
iteration 900 / 1500: loss 10.327907
iteration 1000 / 1500: loss 10.310513
iteration 1100 / 1500: loss 10.287885
iteration 1200 / 1500: loss 10.259661
iteration 1300 / 1500: loss 10.233150
iteration 1400 / 1500: loss 10.191958
iteration 0 / 1500: loss 16.635261
iteration 100 / 1500: loss 15.920372
iteration 200 / 1500: loss 15.261402
iteration 300 / 1500: loss 14.666318
iteration 400 / 1500: loss 14.126514
iteration 500 / 1500: loss 13.648507
iteration 600 / 1500: loss 13.197804
iteration 700 / 1500: loss 12.798200
iteration 800 / 1500: loss 12.440185
iteration 900 / 1500: loss 12.117257
iteration 1000 / 1500: loss 11.824610
iteration 1100 / 1500: loss 11.544930
iteration 1200 / 1500: loss 11.300652
iteration 1300 / 1500: loss 11.081145
iteration 1400 / 1500: loss 10.881673
iteration 0 / 1500: loss 24.597960
iteration 100 / 1500: loss 21.769524
iteration 200 / 1500: loss 19.453502
iteration 300 / 1500: loss 17.565047
iteration 400 / 1500: loss 16.009198
iteration 500 / 1500: loss 14.734748
iteration 600 / 1500: loss 13.695182
iteration 700 / 1500: loss 12.845738
iteration 800 / 1500: loss 12.145728
iteration 900 / 1500: loss 11.574274
iteration 1000 / 1500: loss 11.106844
iteration 1100 / 1500: loss 10.726786
iteration 1200 / 1500: loss 10.413845
iteration 1300 / 1500: loss 10.156120
iteration 1400 / 1500: loss 9.942285
iteration 0 / 1500: loss 41.215168
iteration 100 / 1500: loss 30.586929
iteration 200 / 1500: loss 23.459343
iteration 300 / 1500: loss 18.687359
iteration 400 / 1500: loss 15.492110
iteration 500 / 1500: loss 13.353427
iteration 600 / 1500: loss 11.912268
iteration 700 / 1500: loss 10.955144
iteration 800 / 1500: loss 10.307624
```

```
iteration 900 / 1500: loss 9.876250
iteration 1000 / 1500: loss 9.587906
iteration 1100 / 1500: loss 9.392915
iteration 1200 / 1500: loss 9.264058
iteration 1300 / 1500: loss 9.175502
iteration 1400 / 1500: loss 9.116656
iteration 0 / 1500: loss 56.992314
iteration 100 / 1500: loss 35.318036
iteration 200 / 1500: loss 23.428484
iteration 300 / 1500: loss 16.912829
iteration 400 / 1500: loss 13.338948
iteration 500 / 1500: loss 11.377607
iteration 600 / 1500: loss 10.304441
iteration 700 / 1500: loss 9.713867
iteration 800 / 1500: loss 9.392074
iteration 900 / 1500: loss 9.214761
iteration 1000 / 1500: loss 9.117002
iteration 1100 / 1500: loss 9.064144
iteration 1200 / 1500: loss 9.034676
iteration 1300 / 1500: loss 9.018974
iteration 1400 / 1500: loss 9.009795
iteration 0 / 1500: loss 71.806144
iteration 100 / 1500: loss 37.177618
iteration 200 / 1500: loss 21.640609
iteration 300 / 1500: loss 14.671909
iteration 400 / 1500: loss 11.544970
iteration 500 / 1500: loss 10.142297
iteration 600 / 1500: loss 9.512133
iteration 700 / 1500: loss 9.229340
iteration 800 / 1500: loss 9.102448
iteration 900 / 1500: loss 9.045769
iteration 1000 / 1500: loss 9.020374
iteration 1100 / 1500: loss 9.009014
iteration 1200 / 1500: loss 9.003781
iteration 1300 / 1500: loss 9.001471
iteration 1400 / 1500: loss 9.000446
iteration 0 / 1500: loss 88.089470
iteration 100 / 1500: loss 38.027785
iteration 200 / 1500: loss 19.648601
iteration 300 / 1500: loss 12.908922
iteration 400 / 1500: loss 10.437089
iteration 500 / 1500: loss 9.525816
iteration 600 / 1500: loss 9.192766
iteration 700 / 1500: loss 9.070484
iteration 800 / 1500: loss 9.025501
iteration 900 / 1500: loss 9.009187
iteration 1000 / 1500: loss 9.003053
iteration 1100 / 1500: loss 9.000979
iteration 1200 / 1500: loss 9.000162
iteration 1300 / 1500: loss 8.999800
iteration 1400 / 1500: loss 8.999701
iteration 0 / 1500: loss 171.061714
iteration 100 / 1500: loss 30.716381
```

```
iteration 200 / 1500: loss 11.909215
iteration 300 / 1500: loss 9.389579
iteration 400 / 1500: loss 9.052185
iteration 500 / 1500: loss 9.006856
iteration 600 / 1500: loss 9.000763
iteration 700 / 1500: loss 8.999907
iteration 800 / 1500: loss 8.999812
iteration 900 / 1500: loss 8.999824
iteration 1000 / 1500: loss 8.999816
iteration 1100 / 1500: loss 8.999858
iteration 1200 / 1500: loss 8.999816
iteration 1300 / 1500: loss 8.999841
iteration 1400 / 1500: loss 8.999859
iteration 0 / 1500: loss 10.649769
iteration 100 / 1500: loss 10.601363
iteration 200 / 1500: loss 10.510027
iteration 300 / 1500: loss 10.454438
iteration 400 / 1500: loss 10.401835
iteration 500 / 1500: loss 10.344445
iteration 600 / 1500: loss 10.300307
iteration 700 / 1500: loss 10.260208
iteration 800 / 1500: loss 10.193068
iteration 900 / 1500: loss 10.161515
iteration 1000 / 1500: loss 10.106179
iteration 1100 / 1500: loss 10.050118
iteration 1200 / 1500: loss 10.007931
iteration 1300 / 1500: loss 9.982270
iteration 1400 / 1500: loss 9.938138
iteration 0 / 1500: loss 17.387181
iteration 100 / 1500: loss 15.866039
iteration 200 / 1500: loss 14.617982
iteration 300 / 1500: loss 13.600652
iteration 400 / 1500: loss 12.771689
iteration 500 / 1500: loss 12.079819
iteration 600 / 1500: loss 11.524240
iteration 700 / 1500: loss 11.064045
iteration 800 / 1500: loss 10.686178
iteration 900 / 1500: loss 10.379494
iteration 1000 / 1500: loss 10.134963
iteration 1100 / 1500: loss 9.930588
iteration 1200 / 1500: loss 9.757500
iteration 1300 / 1500: loss 9.618199
iteration 1400 / 1500: loss 9.504882
iteration 0 / 1500: loss 24.795060
iteration 100 / 1500: loss 19.577447
iteration 200 / 1500: loss 16.091228
iteration 300 / 1500: loss 13.756382
iteration 400 / 1500: loss 12.181977
iteration 500 / 1500: loss 11.134599
iteration 600 / 1500: loss 10.432101
iteration 700 / 1500: loss 9.959688
iteration 800 / 1500: loss 9.640130
iteration 900 / 1500: loss 9.429855
```

```
iteration 1000 / 1500: loss 9.287308
iteration 1100 / 1500: loss 9.190578
iteration 1200 / 1500: loss 9.125790
iteration 1300 / 1500: loss 9.084592
iteration 1400 / 1500: loss 9.056440
iteration 0 / 1500: loss 41.984678
iteration 100 / 1500: loss 23.792966
iteration 200 / 1500: loss 15.632379
iteration 300 / 1500: loss 11.978102
iteration 400 / 1500: loss 10.336052
iteration 500 / 1500: loss 9.596862
iteration 600 / 1500: loss 9.268282
iteration 700 / 1500: loss 9.120058
iteration 800 / 1500: loss 9.053599
iteration 900 / 1500: loss 9.022866
iteration 1000 / 1500: loss 9.009820
iteration 1100 / 1500: loss 9.003661
iteration 1200 / 1500: loss 9.001228
iteration 1300 / 1500: loss 9.000169
iteration 1400 / 1500: loss 8.999427
iteration 0 / 1500: loss 54.622317
iteration 100 / 1500: loss 22.694929
iteration 200 / 1500: loss 13.106683
iteration 300 / 1500: loss 10.235222
iteration 400 / 1500: loss 9.369835
iteration 500 / 1500: loss 9.110756
iteration 600 / 1500: loss 9.032846
iteration 700 / 1500: loss 9.009534
iteration 800 / 1500: loss 9.002205
iteration 900 / 1500: loss 9.000426
iteration 1000 / 1500: loss 8.999723
iteration 1100 / 1500: loss 8.999557
iteration 1200 / 1500: loss 8.999541
iteration 1300 / 1500: loss 8.999424
iteration 1400 / 1500: loss 8.999423
iteration 0 / 1500: loss 73.772330
iteration 100 / 1500: loss 21.988922
iteration 200 / 1500: loss 11.608119
iteration 300 / 1500: loss 9.521271
iteration 400 / 1500: loss 9.104405
iteration 500 / 1500: loss 9.020349
iteration 600 / 1500: loss 9.003797
iteration 700 / 1500: loss 9.000310
iteration 800 / 1500: loss 8.999705
iteration 900 / 1500: loss 8.999640
iteration 1000 / 1500: loss 8.999554
iteration 1100 / 1500: loss 8.999570
iteration 1200 / 1500: loss 8.999551
iteration 1300 / 1500: loss 8.999696
iteration 1400 / 1500: loss 8.999587
iteration 0 / 1500: loss 87.630900
iteration 100 / 1500: loss 19.528651
iteration 200 / 1500: loss 10.410475
```

```
iteration 300 / 1500: loss 9.188930
iteration 400 / 1500: loss 9.024899
iteration 500 / 1500: loss 9.003082
iteration 600 / 1500: loss 9.000108
iteration 700 / 1500: loss 8.999695
iteration 800 / 1500: loss 8.999671
iteration 900 / 1500: loss 8.999654
iteration 1000 / 1500: loss 8.999692
iteration 1100 / 1500: loss 8.999623
iteration 1200 / 1500: loss 8.999664
iteration 1300 / 1500: loss 8.999582
iteration 1400 / 1500: loss 8.999663
iteration 0 / 1500: loss 169.096718
iteration 100 / 1500: loss 11.814801
iteration 200 / 1500: loss 9.049346
iteration 300 / 1500: loss 9.000670
iteration 400 / 1500: loss 8.999870
iteration 500 / 1500: loss 8.999839
iteration 600 / 1500: loss 8.999842
iteration 700 / 1500: loss 8.999862
iteration 800 / 1500: loss 8.999791
iteration 900 / 1500: loss 8.999797
iteration 1000 / 1500: loss 8.999793
iteration 1100 / 1500: loss 8.999786
iteration 1200 / 1500: loss 8.999813
iteration 1300 / 1500: loss 8.999809
iteration 1400 / 1500: loss 8.999826
iteration 0 / 1500: loss 10.604011
iteration 100 / 1500: loss 10.472676
iteration 200 / 1500: loss 10.364896
iteration 300 / 1500: loss 10.261710
iteration 400 / 1500: loss 10.167433
iteration 500 / 1500: loss 10.074861
iteration 600 / 1500: loss 9.979947
iteration 700 / 1500: loss 9.914838
iteration 800 / 1500: loss 9.828529
iteration 900 / 1500: loss 9.781575
iteration 1000 / 1500: loss 9.710182
iteration 1100 / 1500: loss 9.670637
iteration 1200 / 1500: loss 9.611616
iteration 1300 / 1500: loss 9.563630
iteration 1400 / 1500: loss 9.509502
iteration 0 / 1500: loss 16.966789
iteration 100 / 1500: loss 14.350894
iteration 200 / 1500: loss 12.579941
iteration 300 / 1500: loss 11.394810
iteration 400 / 1500: loss 10.613195
iteration 500 / 1500: loss 10.076459
iteration 600 / 1500: loss 9.720416
iteration 700 / 1500: loss 9.480665
iteration 800 / 1500: loss 9.319702
iteration 900 / 1500: loss 9.213474
iteration 1000 / 1500: loss 9.142741
```

```
iteration 1100 / 1500: loss 9.094373
iteration 1200 / 1500: loss 9.061986
iteration 1300 / 1500: loss 9.040417
iteration 1400 / 1500: loss 9.025286
iteration 0 / 1500: loss 25.081834
iteration 100 / 1500: loss 16.212045
iteration 200 / 1500: loss 12.230897
iteration 300 / 1500: loss 10.446150
iteration 400 / 1500: loss 9.647403
iteration 500 / 1500: loss 9.290825
iteration 600 / 1500: loss 9.128026
iteration 700 / 1500: loss 9.057532
iteration 800 / 1500: loss 9.024519
iteration 900 / 1500: loss 9.010335
iteration 1000 / 1500: loss 9.003811
iteration 1100 / 1500: loss 9.000578
iteration 1200 / 1500: loss 8.999315
iteration 1300 / 1500: loss 8.998705
iteration 1400 / 1500: loss 8.998336
iteration 0 / 1500: loss 41.144295
iteration 100 / 1500: loss 15.447825
iteration 200 / 1500: loss 10.292492
iteration 300 / 1500: loss 9.259019
iteration 400 / 1500: loss 9.051028
iteration 500 / 1500: loss 9.009489
iteration 600 / 1500: loss 9.001051
iteration 700 / 1500: loss 8.999504
iteration 800 / 1500: loss 8.999237
iteration 900 / 1500: loss 8.999150
iteration 1000 / 1500: loss 8.999052
iteration 1100 / 1500: loss 8.999022
iteration 1200 / 1500: loss 8.999202
iteration 1300 / 1500: loss 8.999341
iteration 1400 / 1500: loss 8.999178
iteration 0 / 1500: loss 58.600333
iteration 100 / 1500: loss 13.433920
iteration 200 / 1500: loss 9.395946
iteration 300 / 1500: loss 9.034582
iteration 400 / 1500: loss 9.002482
iteration 500 / 1500: loss 8.999674
iteration 600 / 1500: loss 8.999379
iteration 700 / 1500: loss 8.999499
iteration 800 / 1500: loss 8.999400
iteration 900 / 1500: loss 8.999474
iteration 1000 / 1500: loss 8.999220
iteration 1100 / 1500: loss 8.999395
iteration 1200 / 1500: loss 8.999348
iteration 1300 / 1500: loss 8.999499
iteration 1400 / 1500: loss 8.999431
iteration 0 / 1500: loss 75.681033
iteration 100 / 1500: loss 11.651563
iteration 200 / 1500: loss 9.104614
iteration 300 / 1500: loss 9.003786
```

```
iteration 400 / 1500: loss 8.999635
iteration 500 / 1500: loss 8.999590
iteration 600 / 1500: loss 8.999578
iteration 700 / 1500: loss 8.999654
iteration 800 / 1500: loss 8.999546
iteration 900 / 1500: loss 8.999539
iteration 1000 / 1500: loss 8.999626
iteration 1100 / 1500: loss 8.999588
iteration 1200 / 1500: loss 8.999535
iteration 1300 / 1500: loss 8.999426
iteration 1400 / 1500: loss 8.999521
iteration 0 / 1500: loss 90.584749
iteration 100 / 1500: loss 10.432575
iteration 200 / 1500: loss 9.025119
iteration 300 / 1500: loss 9.000137
iteration 400 / 1500: loss 8.999540
iteration 500 / 1500: loss 8.999557
iteration 600 / 1500: loss 8.999643
iteration 700 / 1500: loss 8.999586
iteration 800 / 1500: loss 8.999671
iteration 900 / 1500: loss 8.999660
iteration 1000 / 1500: loss 8.999664
iteration 1100 / 1500: loss 8.999622
iteration 1200 / 1500: loss 8.999616
iteration 1300 / 1500: loss 8.999588
iteration 1400 / 1500: loss 8.999640
iteration 0 / 1500: loss 173.694308
iteration 100 / 1500: loss 9.046457
iteration 200 / 1500: loss 8.999853
iteration 300 / 1500: loss 8.999820
iteration 400 / 1500: loss 8.999819
iteration 500 / 1500: loss 8.999787
iteration 600 / 1500: loss 8.999841
iteration 700 / 1500: loss 8.999827
iteration 800 / 1500: loss 8.999826
iteration 900 / 1500: loss 8.999828
iteration 1000 / 1500: loss 8.999815
iteration 1100 / 1500: loss 8.999838
iteration 1200 / 1500: loss 8.999837
iteration 1300 / 1500: loss 8.999824
iteration 1400 / 1500: loss 8.999834
iteration 0 / 1500: loss 10.597976
iteration 100 / 1500: loss 10.422787
iteration 200 / 1500: loss 10.261017
iteration 300 / 1500: loss 10.116479
iteration 400 / 1500: loss 9.975429
iteration 500 / 1500: loss 9.873461
iteration 600 / 1500: loss 9.773045
iteration 700 / 1500: loss 9.676795
iteration 800 / 1500: loss 9.597652
iteration 900 / 1500: loss 9.521525
iteration 1000 / 1500: loss 9.463032
iteration 1100 / 1500: loss 9.420537
```

```
iteration 1200 / 1500: loss 9.366744
iteration 1300 / 1500: loss 9.323150
iteration 1400 / 1500: loss 9.282847
iteration 0 / 1500: loss 17.233463
iteration 100 / 1500: loss 13.505898
iteration 200 / 1500: loss 11.479238
iteration 300 / 1500: loss 10.357619
iteration 400 / 1500: loss 9.741784
iteration 500 / 1500: loss 9.404683
iteration 600 / 1500: loss 9.222597
iteration 700 / 1500: loss 9.120459
iteration 800 / 1500: loss 9.062282
iteration 900 / 1500: loss 9.032998
iteration 1000 / 1500: loss 9.016069
iteration 1100 / 1500: loss 9.007595
iteration 1200 / 1500: loss 9.002323
iteration 1300 / 1500: loss 8.999683
iteration 1400 / 1500: loss 8.998030
iteration 0 / 1500: loss 25.399315
iteration 100 / 1500: loss 13.916744
iteration 200 / 1500: loss 10.472596
iteration 300 / 1500: loss 9.441882
iteration 400 / 1500: loss 9.129961
iteration 500 / 1500: loss 9.038050
iteration 600 / 1500: loss 9.010227
iteration 700 / 1500: loss 9.001496
iteration 800 / 1500: loss 8.999173
iteration 900 / 1500: loss 8.998699
iteration 1000 / 1500: loss 8.997935
iteration 1100 / 1500: loss 8.998308
iteration 1200 / 1500: loss 8.998244
iteration 1300 / 1500: loss 8.998004
iteration 1400 / 1500: loss 8.998457
iteration 0 / 1500: loss 41.763473
iteration 100 / 1500: loss 11.928966
iteration 200 / 1500: loss 9.260535
iteration 300 / 1500: loss 9.022207
iteration 400 / 1500: loss 9.001227
iteration 500 / 1500: loss 8.999098
iteration 600 / 1500: loss 8.999064
iteration 700 / 1500: loss 8.999171
iteration 800 / 1500: loss 8.999094
iteration 900 / 1500: loss 8.999110
iteration 1000 / 1500: loss 8.999224
iteration 1100 / 1500: loss 8.999188
iteration 1200 / 1500: loss 8.999206
iteration 1300 / 1500: loss 8.999165
iteration 1400 / 1500: loss 8.998876
iteration 0 / 1500: loss 57.657941
iteration 100 / 1500: loss 10.287605
iteration 200 / 1500: loss 9.033523
iteration 300 / 1500: loss 9.000453
iteration 400 / 1500: loss 8.999401
```

```
iteration 500 / 1500: loss 8.999362
iteration 600 / 1500: loss 8.999563
iteration 700 / 1500: loss 8.999465
iteration 800 / 1500: loss 8.999545
iteration 900 / 1500: loss 8.999398
iteration 1000 / 1500: loss 8.999354
iteration 1100 / 1500: loss 8.999484
iteration 1200 / 1500: loss 8.999383
iteration 1300 / 1500: loss 8.999403
iteration 1400 / 1500: loss 8.999410
iteration 0 / 1500: loss 75.892270
iteration 100 / 1500: loss 9.520302
iteration 200 / 1500: loss 9.003729
iteration 300 / 1500: loss 8.999623
iteration 400 / 1500: loss 8.999502
iteration 500 / 1500: loss 8.999656
iteration 600 / 1500: loss 8.999631
iteration 700 / 1500: loss 8.999546
iteration 800 / 1500: loss 8.999476
iteration 900 / 1500: loss 8.999507
iteration 1000 / 1500: loss 8.999578
iteration 1100 / 1500: loss 8.999559
iteration 1200 / 1500: loss 8.999521
iteration 1300 / 1500: loss 8.999553
iteration 1400 / 1500: loss 8.999527
iteration 0 / 1500: loss 89.518852
iteration 100 / 1500: loss 9.181799
iteration 200 / 1500: loss 9.000124
iteration 300 / 1500: loss 8.999649
iteration 400 / 1500: loss 8.999701
iteration 500 / 1500: loss 8.999619
iteration 600 / 1500: loss 8.999628
iteration 700 / 1500: loss 8.999640
iteration 800 / 1500: loss 8.999594
iteration 900 / 1500: loss 8.999611
iteration 1000 / 1500: loss 8.999734
iteration 1100 / 1500: loss 8.999569
iteration 1200 / 1500: loss 8.999629
iteration 1300 / 1500: loss 8.999596
iteration 1400 / 1500: loss 8.999583
iteration 0 / 1500: loss 161.811135
iteration 100 / 1500: loss 9.000467
iteration 200 / 1500: loss 8.999852
iteration 300 / 1500: loss 8.999852
iteration 400 / 1500: loss 8.999797
iteration 500 / 1500: loss 8.999783
iteration 600 / 1500: loss 8.999849
iteration 700 / 1500: loss 8.999827
iteration 800 / 1500: loss 8.999826
iteration 900 / 1500: loss 8.999832
iteration 1000 / 1500: loss 8.999842
iteration 1100 / 1500: loss 8.999863
iteration 1200 / 1500: loss 8.999851
```

```
iteration 1300 / 1500: loss 8.999799
iteration 1400 / 1500: loss 8.999831
iteration 0 / 1500: loss 10.729373
iteration 100 / 1500: loss 10.465829
iteration 200 / 1500: loss 10.264722
iteration 300 / 1500: loss 10.064632
iteration 400 / 1500: loss 9.909230
iteration 500 / 1500: loss 9.758948
iteration 600 / 1500: loss 9.651990
iteration 700 / 1500: loss 9.559395
iteration 800 / 1500: loss 9.472006
iteration 900 / 1500: loss 9.403993
iteration 1000 / 1500: loss 9.333529
iteration 1100 / 1500: loss 9.281952
iteration 1200 / 1500: loss 9.235019
iteration 1300 / 1500: loss 9.203810
iteration 1400 / 1500: loss 9.165482
iteration 0 / 1500: loss 17.327822
iteration 100 / 1500: loss 12.731024
iteration 200 / 1500: loss 10.664237
iteration 300 / 1500: loss 9.747534
iteration 400 / 1500: loss 9.337824
iteration 500 / 1500: loss 9.147404
iteration 600 / 1500: loss 9.063699
iteration 700 / 1500: loss 9.027401
iteration 800 / 1500: loss 9.010290
iteration 900 / 1500: loss 9.002423
iteration 1000 / 1500: loss 8.998465
iteration 1100 / 1500: loss 8.996756
iteration 1200 / 1500: loss 8.996777
iteration 1300 / 1500: loss 8.996360
iteration 1400 / 1500: loss 8.996326
iteration 0 / 1500: loss 24.396710
iteration 100 / 1500: loss 12.090010
iteration 200 / 1500: loss 9.618799
iteration 300 / 1500: loss 9.122673
iteration 400 / 1500: loss 9.023045
iteration 500 / 1500: loss 9.003485
iteration 600 / 1500: loss 8.999619
iteration 700 / 1500: loss 8.998247
iteration 800 / 1500: loss 8.998359
iteration 900 / 1500: loss 8.997749
iteration 1000 / 1500: loss 8.998128
iteration 1100 / 1500: loss 8.998368
iteration 1200 / 1500: loss 8.998521
iteration 1300 / 1500: loss 8.998490
iteration 1400 / 1500: loss 8.998483
iteration 0 / 1500: loss 42.884748
iteration 100 / 1500: loss 10.346457
iteration 200 / 1500: loss 9.052748
iteration 300 / 1500: loss 9.001272
iteration 400 / 1500: loss 8.999069
iteration 500 / 1500: loss 8.998997
```

```
iteration 600 / 1500: loss 8.999062
iteration 700 / 1500: loss 8.999167
iteration 800 / 1500: loss 8.999007
iteration 900 / 1500: loss 8.999224
iteration 1000 / 1500: loss 8.999115
iteration 1100 / 1500: loss 8.999282
iteration 1200 / 1500: loss 8.999243
iteration 1300 / 1500: loss 8.999141
iteration 1400 / 1500: loss 8.999122
iteration 0 / 1500: loss 57.042659
iteration 100 / 1500: loss 9.372103
iteration 200 / 1500: loss 9.002303
iteration 300 / 1500: loss 8.999517
iteration 400 / 1500: loss 8.999503
iteration 500 / 1500: loss 8.999315
iteration 600 / 1500: loss 8.999387
iteration 700 / 1500: loss 8.999357
iteration 800 / 1500: loss 8.999399
iteration 900 / 1500: loss 8.999391
iteration 1000 / 1500: loss 8.999494
iteration 1100 / 1500: loss 8.999353
iteration 1200 / 1500: loss 8.999349
iteration 1300 / 1500: loss 8.999480
iteration 1400 / 1500: loss 8.999298
iteration 0 / 1500: loss 72.398296
iteration 100 / 1500: loss 9.094401
iteration 200 / 1500: loss 8.999673
iteration 300 / 1500: loss 8.999496
iteration 400 / 1500: loss 8.999612
iteration 500 / 1500: loss 8.999604
iteration 600 / 1500: loss 8.999550
iteration 700 / 1500: loss 8.999600
iteration 800 / 1500: loss 8.999535
iteration 900 / 1500: loss 8.999512
iteration 1000 / 1500: loss 8.999676
iteration 1100 / 1500: loss 8.999496
iteration 1200 / 1500: loss 8.999633
iteration 1300 / 1500: loss 8.999523
iteration 1400 / 1500: loss 8.999590
iteration 0 / 1500: loss 90.342942
iteration 100 / 1500: loss 9.022753
iteration 200 / 1500: loss 8.999724
iteration 300 / 1500: loss 8.999726
iteration 400 / 1500: loss 8.999644
iteration 500 / 1500: loss 8.999647
iteration 600 / 1500: loss 8.999721
iteration 700 / 1500: loss 8.999716
iteration 800 / 1500: loss 8.999638
iteration 900 / 1500: loss 8.999684
iteration 1000 / 1500: loss 8.999610
iteration 1100 / 1500: loss 8.999708
iteration 1200 / 1500: loss 8.999627
iteration 1300 / 1500: loss 8.999707
```

```
iteration 1400 / 1500: loss 8.999655
iteration 0 / 1500: loss 173.556081
iteration 100 / 1500: loss 8.999862
iteration 200 / 1500: loss 8.999848
iteration 300 / 1500: loss 8.999832
iteration 400 / 1500: loss 8.999807
iteration 500 / 1500: loss 8.999828
iteration 600 / 1500: loss 8.999817
iteration 700 / 1500: loss 8.999855
iteration 800 / 1500: loss 8.999844
iteration 900 / 1500: loss 8.999846
iteration 1000 / 1500: loss 8.999830
iteration 1100 / 1500: loss 8.999879
iteration 1200 / 1500: loss 8.999808
iteration 1300 / 1500: loss 8.999812
iteration 1400 / 1500: loss 8.999814
iteration 0 / 1500: loss 10.570393
iteration 100 / 1500: loss 10.264311
iteration 200 / 1500: loss 10.047732
iteration 300 / 1500: loss 9.855832
iteration 400 / 1500: loss 9.684709
iteration 500 / 1500: loss 9.565026
iteration 600 / 1500: loss 9.451793
iteration 700 / 1500: loss 9.367290
iteration 800 / 1500: loss 9.297359
iteration 900 / 1500: loss 9.247584
iteration 1000 / 1500: loss 9.196811
iteration 1100 / 1500: loss 9.159311
iteration 1200 / 1500: loss 9.122963
iteration 1300 / 1500: loss 9.105164
iteration 1400 / 1500: loss 9.079747
iteration 0 / 1500: loss 16.981030
iteration 100 / 1500: loss 11.920799
iteration 200 / 1500: loss 10.069927
iteration 300 / 1500: loss 9.392087
iteration 400 / 1500: loss 9.143255
iteration 500 / 1500: loss 9.048390
iteration 600 / 1500: loss 9.015748
iteration 700 / 1500: loss 9.004352
iteration 800 / 1500: loss 8.999161
iteration 900 / 1500: loss 8.997283
iteration 1000 / 1500: loss 8.997040
iteration 1100 / 1500: loss 8.996720
iteration 1200 / 1500: loss 8.996523
iteration 1300 / 1500: loss 8.997189
iteration 1400 / 1500: loss 8.996433
iteration 0 / 1500: loss 25.307496
iteration 100 / 1500: loss 11.188444
iteration 200 / 1500: loss 9.293045
iteration 300 / 1500: loss 9.037573
iteration 400 / 1500: loss 9.003320
iteration 500 / 1500: loss 8.999025
iteration 600 / 1500: loss 8.998391
```

```
iteration 700 / 1500: loss 8.998195
iteration 800 / 1500: loss 8.998018
iteration 900 / 1500: loss 8.998440
iteration 1000 / 1500: loss 8.998252
iteration 1100 / 1500: loss 8.998526
iteration 1200 / 1500: loss 8.998471
iteration 1300 / 1500: loss 8.997971
iteration 1400 / 1500: loss 8.998179
iteration 0 / 1500: loss 41.368160
iteration 100 / 1500: loss 9.569448
iteration 200 / 1500: loss 9.008812
iteration 300 / 1500: loss 8.999039
iteration 400 / 1500: loss 8.999124
iteration 500 / 1500: loss 8.999072
iteration 600 / 1500: loss 8.999272
iteration 700 / 1500: loss 8.998984
iteration 800 / 1500: loss 8.999159
iteration 900 / 1500: loss 8.998962
iteration 1000 / 1500: loss 8.999010
iteration 1100 / 1500: loss 8.999078
iteration 1200 / 1500: loss 8.999084
iteration 1300 / 1500: loss 8.999181
iteration 1400 / 1500: loss 8.999083
iteration 0 / 1500: loss 54.815418
iteration 100 / 1500: loss 9.102831
iteration 200 / 1500: loss 8.999669
iteration 300 / 1500: loss 8.999447
iteration 400 / 1500: loss 8.999405
iteration 500 / 1500: loss 8.999452
iteration 600 / 1500: loss 8.999530
iteration 700 / 1500: loss 8.999408
iteration 800 / 1500: loss 8.999338
iteration 900 / 1500: loss 8.999496
iteration 1000 / 1500: loss 8.999509
iteration 1100 / 1500: loss 8.999203
iteration 1200 / 1500: loss 8.999437
iteration 1300 / 1500: loss 8.999345
iteration 1400 / 1500: loss 8.999330
iteration 0 / 1500: loss 75.362154
iteration 100 / 1500: loss 9.018300
iteration 200 / 1500: loss 8.999580
iteration 300 / 1500: loss 8.999650
iteration 400 / 1500: loss 8.999458
iteration 500 / 1500: loss 8.999642
iteration 600 / 1500: loss 8.999487
iteration 700 / 1500: loss 8.999515
iteration 800 / 1500: loss 8.999649
iteration 900 / 1500: loss 8.999542
iteration 1000 / 1500: loss 8.999501
iteration 1100 / 1500: loss 8.999602
iteration 1200 / 1500: loss 8.999582
iteration 1300 / 1500: loss 8.999585
iteration 1400 / 1500: loss 8.999560
```

```
iteration 0 / 1500: loss 86.802750
iteration 100 / 1500: loss 9.002371
iteration 200 / 1500: loss 8.999649
iteration 300 / 1500: loss 8.999634
iteration 400 / 1500: loss 8.999653
iteration 500 / 1500: loss 8.999657
iteration 600 / 1500: loss 8.999642
iteration 700 / 1500: loss 8.999632
iteration 800 / 1500: loss 8.999622
iteration 900 / 1500: loss 8.999620
iteration 1000 / 1500: loss 8.999635
iteration 1100 / 1500: loss 8.999676
iteration 1200 / 1500: loss 8.999669
iteration 1300 / 1500: loss 8.999626
iteration 1400 / 1500: loss 8.999655
iteration 0 / 1500: loss 161.531972
iteration 100 / 1500: loss 8.999815
iteration 200 / 1500: loss 8.999822
iteration 300 / 1500: loss 8.999835
iteration 400 / 1500: loss 8.999802
iteration 500 / 1500: loss 8.999814
iteration 600 / 1500: loss 8.999784
iteration 700 / 1500: loss 8.999858
iteration 800 / 1500: loss 8.999833
iteration 900 / 1500: loss 8.999802
iteration 1000 / 1500: loss 8.999793
iteration 1100 / 1500: loss 8.999823
iteration 1200 / 1500: loss 8.999889
iteration 1300 / 1500: loss 8.999873
iteration 1400 / 1500: loss 8.999832
iteration 0 / 1500: loss 10.587828
iteration 100 / 1500: loss 10.052891
iteration 200 / 1500: loss 9.702093
iteration 300 / 1500: loss 9.458461
iteration 400 / 1500: loss 9.306641
iteration 500 / 1500: loss 9.196640
iteration 600 / 1500: loss 9.128164
iteration 700 / 1500: loss 9.076964
iteration 800 / 1500: loss 9.044735
iteration 900 / 1500: loss 9.026516
iteration 1000 / 1500: loss 9.010755
iteration 1100 / 1500: loss 9.001037
iteration 1200 / 1500: loss 8.994874
iteration 1300 / 1500: loss 8.989765
iteration 1400 / 1500: loss 8.989583
iteration 0 / 1500: loss 16.568270
iteration 100 / 1500: loss 10.011221
iteration 200 / 1500: loss 9.133680
iteration 300 / 1500: loss 9.013916
iteration 400 / 1500: loss 8.998662
iteration 500 / 1500: loss 8.996388
iteration 600 / 1500: loss 8.996028
iteration 700 / 1500: loss 8.996395
```

```
iteration 800 / 1500: loss 8.996658
iteration 900 / 1500: loss 8.995615
iteration 1000 / 1500: loss 8.996330
iteration 1100 / 1500: loss 8.996738
iteration 1200 / 1500: loss 8.996399
iteration 1300 / 1500: loss 8.996673
iteration 1400 / 1500: loss 8.996325
iteration 0 / 1500: loss 24.854267
iteration 100 / 1500: loss 9.278244
iteration 200 / 1500: loss 9.002978
iteration 300 / 1500: loss 8.998335
iteration 400 / 1500: loss 8.998720
iteration 500 / 1500: loss 8.998517
iteration 600 / 1500: loss 8.997834
iteration 700 / 1500: loss 8.998193
iteration 800 / 1500: loss 8.997918
iteration 900 / 1500: loss 8.998207
iteration 1000 / 1500: loss 8.998369
iteration 1100 / 1500: loss 8.997978
iteration 1200 / 1500: loss 8.998069
iteration 1300 / 1500: loss 8.998181
iteration 1400 / 1500: loss 8.998478
iteration 0 / 1500: loss 41.654656
iteration 100 / 1500: loss 9.008239
iteration 200 / 1500: loss 8.998992
iteration 300 / 1500: loss 8.998919
iteration 400 / 1500: loss 8.999109
iteration 500 / 1500: loss 8.999208
iteration 600 / 1500: loss 8.999118
iteration 700 / 1500: loss 8.998965
iteration 800 / 1500: loss 8.999188
iteration 900 / 1500: loss 8.998938
iteration 1000 / 1500: loss 8.999082
iteration 1100 / 1500: loss 8.999238
iteration 1200 / 1500: loss 8.999082
iteration 1300 / 1500: loss 8.999304
iteration 1400 / 1500: loss 8.999151
iteration 0 / 1500: loss 58.108084
iteration 100 / 1500: loss 8.999538
iteration 200 / 1500: loss 8.999393
iteration 300 / 1500: loss 8.999497
iteration 400 / 1500: loss 8.999308
iteration 500 / 1500: loss 8.999375
iteration 600 / 1500: loss 8.999478
iteration 700 / 1500: loss 8.999545
iteration 800 / 1500: loss 8.999515
iteration 900 / 1500: loss 8.999319
iteration 1000 / 1500: loss 8.999495
iteration 1100 / 1500: loss 8.999382
iteration 1200 / 1500: loss 8.999423
iteration 1300 / 1500: loss 8.999396
iteration 1400 / 1500: loss 8.999458
iteration 0 / 1500: loss 75.523520
```

```
iteration 100 / 1500: loss 8.999494
iteration 200 / 1500: loss 8.999553
iteration 300 / 1500: loss 8.999547
iteration 400 / 1500: loss 8.999507
iteration 500 / 1500: loss 8.999614
iteration 600 / 1500: loss 8.999510
iteration 700 / 1500: loss 8.999666
iteration 800 / 1500: loss 8.999523
iteration 900 / 1500: loss 8.999602
iteration 1000 / 1500: loss 8.999673
iteration 1100 / 1500: loss 8.999549
iteration 1200 / 1500: loss 8.999615
iteration 1300 / 1500: loss 8.999557
iteration 1400 / 1500: loss 8.999568
iteration 0 / 1500: loss 85.872054
iteration 100 / 1500: loss 8.999647
iteration 200 / 1500: loss 8.999685
iteration 300 / 1500: loss 8.999686
iteration 400 / 1500: loss 8.999697
iteration 500 / 1500: loss 8.999716
iteration 600 / 1500: loss 8.999745
iteration 700 / 1500: loss 8.999635
iteration 800 / 1500: loss 8.999690
iteration 900 / 1500: loss 8.999684
iteration 1000 / 1500: loss 8.999669
iteration 1100 / 1500: loss 8.999610
iteration 1200 / 1500: loss 8.999605
iteration 1300 / 1500: loss 8.999643
iteration 1400 / 1500: loss 8.999704
iteration 0 / 1500: loss 173.709683
iteration 100 / 1500: loss 8.999843
iteration 200 / 1500: loss 8.999824
iteration 300 / 1500: loss 8.999836
iteration 400 / 1500: loss 8.999850
iteration 500 / 1500: loss 8.999813
iteration 600 / 1500: loss 8.999864
iteration 700 / 1500: loss 8.999886
iteration 800 / 1500: loss 8.999812
iteration 900 / 1500: loss 8.999853
iteration 1000 / 1500: loss 8.999815
iteration 1100 / 1500: loss 8.999854
iteration 1200 / 1500: loss 8.999830
iteration 1300 / 1500: loss 8.999827
iteration 1400 / 1500: loss 8.999812
iteration 0 / 1500: loss 10.644058
iteration 100 / 1500: loss 9.207834
iteration 200 / 1500: loss 9.011109
iteration 300 / 1500: loss 8.988933
iteration 400 / 1500: loss 8.984772
iteration 500 / 1500: loss 8.983945
iteration 600 / 1500: loss 8.983209
iteration 700 / 1500: loss 8.980983
iteration 800 / 1500: loss 8.981477
```

```
iteration 900 / 1500: loss 8.981896
iteration 1000 / 1500: loss 8.979910
iteration 1100 / 1500: loss 8.983869
iteration 1200 / 1500: loss 8.979598
iteration 1300 / 1500: loss 8.980929
iteration 1400 / 1500: loss 8.983668
iteration 0 / 1500: loss 17.379961
iteration 100 / 1500: loss 8.996374
iteration 200 / 1500: loss 8.995983
iteration 300 / 1500: loss 8.995851
iteration 400 / 1500: loss 8.997152
iteration 500 / 1500: loss 8.996199
iteration 600 / 1500: loss 8.996209
iteration 700 / 1500: loss 8.995864
iteration 800 / 1500: loss 8.995865
iteration 900 / 1500: loss 8.996624
iteration 1000 / 1500: loss 8.996295
iteration 1100 / 1500: loss 8.995846
iteration 1200 / 1500: loss 8.996987
iteration 1300 / 1500: loss 8.996483
iteration 1400 / 1500: loss 8.996271
iteration 0 / 1500: loss 25.867173
iteration 100 / 1500: loss 8.998509
iteration 200 / 1500: loss 8.998708
iteration 300 / 1500: loss 8.998210
iteration 400 / 1500: loss 8.998283
iteration 500 / 1500: loss 8.998032
iteration 600 / 1500: loss 8.998144
iteration 700 / 1500: loss 8.998215
iteration 800 / 1500: loss 8.998542
iteration 900 / 1500: loss 8.998573
iteration 1000 / 1500: loss 8.997999
iteration 1100 / 1500: loss 8.998693
iteration 1200 / 1500: loss 8.998112
iteration 1300 / 1500: loss 8.998575
iteration 1400 / 1500: loss 8.997898
iteration 0 / 1500: loss 42.014623
iteration 100 / 1500: loss 8.999297
iteration 200 / 1500: loss 8.998963
iteration 300 / 1500: loss 8.999306
iteration 400 / 1500: loss 8.999105
iteration 500 / 1500: loss 8.999178
iteration 600 / 1500: loss 8.999129
iteration 700 / 1500: loss 8.999370
iteration 800 / 1500: loss 8.999198
iteration 900 / 1500: loss 8.999138
iteration 1000 / 1500: loss 8.999369
iteration 1100 / 1500: loss 8.999068
iteration 1200 / 1500: loss 8.999063
iteration 1300 / 1500: loss 8.999171
iteration 1400 / 1500: loss 8.999220
iteration 0 / 1500: loss 57.341200
iteration 100 / 1500: loss 8.999549
```

```
iteration 200 / 1500: loss 8.999563
iteration 300 / 1500: loss 8.999709
iteration 400 / 1500: loss 8.999482
iteration 500 / 1500: loss 8.999421
iteration 600 / 1500: loss 8.999578
iteration 700 / 1500: loss 8.999588
iteration 800 / 1500: loss 8.999414
iteration 900 / 1500: loss 8.999363
iteration 1000 / 1500: loss 8.999493
iteration 1100 / 1500: loss 8.999473
iteration 1200 / 1500: loss 8.999486
iteration 1300 / 1500: loss 8.999490
iteration 1400 / 1500: loss 8.999658
iteration 0 / 1500: loss 74.177490
iteration 100 / 1500: loss 8.999791
iteration 200 / 1500: loss 8.999668
iteration 300 / 1500: loss 8.999729
iteration 400 / 1500: loss 8.999738
iteration 500 / 1500: loss 8.999627
iteration 600 / 1500: loss 8.999656
iteration 700 / 1500: loss 8.999694
iteration 800 / 1500: loss 8.999696
iteration 900 / 1500: loss 8.999695
iteration 1000 / 1500: loss 8.999639
iteration 1100 / 1500: loss 8.999769
iteration 1200 / 1500: loss 8.999578
iteration 1300 / 1500: loss 8.999685
iteration 1400 / 1500: loss 8.999714
iteration 0 / 1500: loss 93.935979
iteration 100 / 1500: loss 8.999766
iteration 200 / 1500: loss 8.999712
iteration 300 / 1500: loss 8.999835
iteration 400 / 1500: loss 8.999750
iteration 500 / 1500: loss 8.999812
iteration 600 / 1500: loss 8.999726
iteration 700 / 1500: loss 8.999796
iteration 800 / 1500: loss 8.999877
iteration 900 / 1500: loss 8.999825
iteration 1000 / 1500: loss 8.999736
iteration 1100 / 1500: loss 8.999741
iteration 1200 / 1500: loss 8.999696
iteration 1300 / 1500: loss 8.999707
iteration 1400 / 1500: loss 8.999687
iteration 0 / 1500: loss 172.675160
iteration 100 / 1500: loss 9.000009
iteration 200 / 1500: loss 8.999969
iteration 300 / 1500: loss 9.000000
iteration 400 / 1500: loss 8.999967
iteration 500 / 1500: loss 8.999976
iteration 600 / 1500: loss 8.999971
iteration 700 / 1500: loss 9.000044
iteration 800 / 1500: loss 8.999956
iteration 900 / 1500: loss 8.999927
```

```
iteration 1000 / 1500: loss 9.000006
iteration 1100 / 1500: loss 8.999998
iteration 1200 / 1500: loss 9.000052
iteration 1300 / 1500: loss 9.000013
iteration 1400 / 1500: loss 9.000010
lr 5.000000e-08 reg 1.000000e+03 train accuracy: 0.098429 val accura
cy: 0.098000
lr 5.000000e-08 reg 5.000000e+03 train accuracy: 0.113735 val accura
cy: 0.117000
lr 5.000000e-08 reg 1.000000e+04 train accuracy: 0.132571 val accura
cy: 0.123000
lr 5.000000e-08 reg 2.000000e+04 train accuracy: 0.177735 val accura
cy: 0.192000
lr 5.000000e-08 reg 3.000000e+04 train accuracy: 0.246490 val accura
cy: 0.239000
lr 5.000000e-08 reg 4.000000e+04 train accuracy: 0.382469 val accura
cy: 0.400000
lr 5.000000e-08 reg 5.000000e+04 train accuracy: 0.416327 val accura
cy: 0.408000
lr 5.000000e-08 reg 1.000000e+05 train accuracy: 0.417878 val accura
cy: 0.422000
lr 1.000000e-07 reg 1.000000e+03 train accuracy: 0.123122 val accura
cy: 0.134000
lr 1.000000e-07 reg 5.000000e+03 train accuracy: 0.151082 val accura
cy: 0.147000
lr 1.000000e-07 reg 1.000000e+04 train accuracy: 0.216490 val accura
cy: 0.228000
lr 1.000000e-07 reg 2.000000e+04 train accuracy: 0.405327 val accura
cy: 0.409000
lr 1.000000e-07 reg 3.000000e+04 train accuracy: 0.417612 val accura
cy: 0.419000
lr 1.000000e-07 reg 4.000000e+04 train accuracy: 0.422959 val accura
cy: 0.420000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.417020 val accura
cy: 0.426000
lr 1.000000e-07 reg 1.000000e+05 train accuracy: 0.423184 val accura
cy: 0.435000
lr 2.000000e-07 reg 1.000000e+03 train accuracy: 0.146694 val accura
cy: 0.144000
lr 2.000000e-07 reg 5.000000e+03 train accuracy: 0.291020 val accura
cy: 0.305000
lr 2.000000e-07 reg 1.000000e+04 train accuracy: 0.415959 val accura
cy: 0.406000
lr 2.000000e-07 reg 2.000000e+04 train accuracy: 0.417551 val accura
cy: 0.428000
lr 2.000000e-07 reg 3.000000e+04 train accuracy: 0.419061 val accura
cy: 0.438000
lr 2.000000e-07 reg 4.000000e+04 train accuracy: 0.416653 val accura
cy: 0.412000
lr 2.000000e-07 reg 5.000000e+04 train accuracy: 0.419898 val accura
cy: 0.420000
lr 2.000000e-07 reg 1.000000e+05 train accuracy: 0.412796 val accura
cy: 0.404000
```

```
lr 3.000000e-07 reg 1.000000e+03 train accuracy: 0.205796 val accura
cy: 0.216000
lr 3.000000e-07 reg 5.000000e+03 train accuracy: 0.397898 val accura
cy: 0.393000
lr 3.000000e-07 reg 1.000000e+04 train accuracy: 0.418143 val accura
cy: 0.426000
lr 3.000000e-07 reg 2.000000e+04 train accuracy: 0.419245 val accura
cy: 0.421000
lr 3.000000e-07 reg 3.000000e+04 train accuracy: 0.420061 val accura
cy: 0.429000
lr 3.000000e-07 reg 4.000000e+04 train accuracy: 0.414653 val accura
cy: 0.420000
lr 3.000000e-07 reg 5.000000e+04 train accuracy: 0.412490 val accura
cy: 0.413000
lr 3.000000e-07 reg 1.000000e+05 train accuracy: 0.414673 val accura
cy: 0.422000
lr 4.000000e-07 reg 1.000000e+03 train accuracy: 0.246531 val accura
cy: 0.249000
lr 4.000000e-07 reg 5.000000e+03 train accuracy: 0.417490 val accura
cy: 0.422000
lr 4.000000e-07 reg 1.000000e+04 train accuracy: 0.418898 val accura
cy: 0.421000
lr 4.000000e-07 reg 2.000000e+04 train accuracy: 0.417347 val accura
cy: 0.421000
lr 4.000000e-07 reg 3.000000e+04 train accuracy: 0.416531 val accura
cy: 0.410000
lr 4.000000e-07 reg 4.000000e+04 train accuracy: 0.414816 val accura
cy: 0.412000
lr 4.000000e-07 reg 5.000000e+04 train accuracy: 0.408816 val accura
cy: 0.404000
lr 4.000000e-07 reg 1.000000e+05 train accuracy: 0.420000 val accura
cy: 0.415000
lr 5.000000e-07 reg 1.000000e+03 train accuracy: 0.261265 val accura
cy: 0.303000
lr 5.000000e-07 reg 5.000000e+03 train accuracy: 0.416816 val accura
cy: 0.425000
lr 5.000000e-07 reg 1.000000e+04 train accuracy: 0.420408 val accura
cy: 0.432000
lr 5.000000e-07 reg 2.000000e+04 train accuracy: 0.412306 val accura
cy: 0.414000
lr 5.000000e-07 reg 3.000000e+04 train accuracy: 0.416204 val accura
cy: 0.407000
lr 5.000000e-07 reg 4.000000e+04 train accuracy: 0.417796 val accura
cy: 0.417000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.414102 val accura
cy: 0.418000
lr 5.000000e-07 reg 1.000000e+05 train accuracy: 0.407510 val accura
cy: 0.405000
lr 1.000000e-06 reg 1.000000e+03 train accuracy: 0.408980 val accura
cy: 0.401000
lr 1.000000e-06 reg 5.000000e+03 train accuracy: 0.416551 val accura
cy: 0.415000
lr 1.000000e-06 reg 1.000000e+04 train accuracy: 0.417694 val accura
```

```
cy: 0.412000
lr 1.000000e-06 reg 2.000000e+04 train accuracy: 0.415306 val accura
cy: 0.412000
lr 1.000000e-06 reg 3.000000e+04 train accuracy: 0.419918 val accura
cy: 0.420000
lr 1.000000e-06 reg 4.000000e+04 train accuracy: 0.407918 val accura
cy: 0.404000
lr 1.000000e-06 reg 5.000000e+04 train accuracy: 0.415796 val accura
cy: 0.411000
lr 1.000000e-06 reg 1.000000e+05 train accuracy: 0.405612 val accura
cy: 0.410000
lr 5.000000e-06 reg 1.000000e+03 train accuracy: 0.420694 val accura
cy: 0.421000
lr 5.000000e-06 reg 5.000000e+03 train accuracy: 0.411082 val accura
cy: 0.416000
lr 5.000000e-06 reg 1.000000e+04 train accuracy: 0.412694 val accura
cy: 0.416000
lr 5.000000e-06 reg 2.000000e+04 train accuracy: 0.411633 val accura
cy: 0.413000
lr 5.000000e-06 reg 3.000000e+04 train accuracy: 0.398939 val accura
cy: 0.408000
lr 5.000000e-06 reg 4.000000e+04 train accuracy: 0.364837 val accura
cy: 0.356000
lr 5.000000e-06 reg 5.000000e+04 train accuracy: 0.369918 val accura
cy: 0.374000
lr 5.000000e-06 reg 1.000000e+05 train accuracy: 0.325653 val accura
cy: 0.325000
best validation accuracy achieved during cross-validation: 0.438000
```

In [14]:

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

```
0.426
```

In [15]:

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

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

Inline question 1:

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

Some seem to make sense - for example, peripheral view of bird that can arguably look like a plane taking off or a dog on a grassy field standing on 4 legs mistaken for a horse. But many other misclassifications are hard to interpret.

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 [29]:

```
# 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, 161)
(49000, 160)
```

In [33]:

```
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.                                              #
################################################################################
results = {}
best_val = -1
best_net = None
best_params = None
learning_rates = [1e-1, 5e-1, 1e0]
regularization_strengths = [1e-3, 5e-3, 1e-2]

for l in learning_rates:
    for r in regularization_strengths:
        print(l)
        net = TwoLayerNet(input_dim, hidden_dim, num_classes)
        curr_loss = net.train(X_train_feats, y_train, X_val_feats, y_val, learni
ng_rate=l, reg=r, num_iters=1500)
        y_train_pred = net.predict(X_train_feats)
        y_train_acc = np.mean(y_train == y_train_pred)
        y_val_pred = net.predict(X_val_feats)
        y_val_acc = np.mean(y_val == y_val_pred)
        results[(l, r)] = (y_train_acc, y_val_acc)
        if y_val_acc > best_val:
            best_net = net
            best_val = y_val_acc
            best_params = (l, r)


################################################################################
#                              END OF YOUR CODE                                #
################################################################################
for lr, reg in sorted(results):
    train_accuracy, val_accuracy = results[(lr, reg)]
    print('lr %e reg %e train accuracy: %f val accuracy: %f' % (
                lr, reg, train_accuracy, val_accuracy))

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

```
0.1
0.1
0.1
0.5
0.5
0.5
1.0
1.0
1.0
lr 1.000000e-01 reg 1.000000e-03 train accuracy: 0.540918 val accura
cy: 0.525000
lr 1.000000e-01 reg 5.000000e-03 train accuracy: 0.528388 val accura
cy: 0.514000
lr 1.000000e-01 reg 1.000000e-02 train accuracy: 0.521020 val accura
cy: 0.520000
lr 5.000000e-01 reg 1.000000e-03 train accuracy: 0.660694 val accura
cy: 0.597000
lr 5.000000e-01 reg 5.000000e-03 train accuracy: 0.565143 val accura
cy: 0.559000
lr 5.000000e-01 reg 1.000000e-02 train accuracy: 0.514714 val accura
cy: 0.500000
lr 1.000000e+00 reg 1.000000e-03 train accuracy: 0.671020 val accura
cy: 0.560000
lr 1.000000e+00 reg 5.000000e-03 train accuracy: 0.555163 val accura
cy: 0.519000
lr 1.000000e+00 reg 1.000000e-02 train accuracy: 0.509469 val accura
cy: 0.486000
best validation accuracy achieved during cross-validation: 0.597000
(0.5, 0.001)
```

In [34]:

```
# 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)
```

```
0.577
```