2019211959 文昂

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) 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 transferring 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
```

In [2]:
```
# This is a bit of magic to make matplotlib figures appear inline in the notebook
# rather than in a new window.
%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

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

In [3]:
```
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'
```

In [4]:
```
# 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
```

In [5]:
```
X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)
```

In [6]:
```
# 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 [7]:
```
# 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 [8]:
```
# Subsample the data for more efficient code execution in this exercise
num_training = 5000
mask = list(range(num_training))
X_train = X_train[mask]
y_train = y_train[mask]

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

# Reshape the image data into rows
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 [9]:
```
from cs231n.classifiers import KNearestNeighbor

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

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

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

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

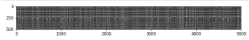
Note: For the three distance computations that we require you to implement in this notebook, you may not use the np.linalg.norm() function that numpy provides.

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

In [10]:
```
# 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 [11]:
```
# We can visualize the distance matrix: each row is a single test example and
# its distances to training examples
plt.imshow(dists, interpolation='none')
plt.show()
```



Inline Question 1

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

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

Your Answer: *1. No train case is close to the test case and it may be a noise. 2. No test case is close to this train case and it may be a outlier.*

In [12]:
```
# 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 [13]:
```
y_test_pred = classifier.predict_labels(dists, k=5)
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
```

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

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

Inline Question 2

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

the mean μ across all pixels over all images is

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

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

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

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

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

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

Your Answer: *1, 3*

Your Explanation: *1 and 3 are linear transformations.*

In [14]:
```
# Now lets speed up distance matrix computation by using partial vectorization
# with one loop. Implement the function compute_distances_one_loop and run the
# code below:
dists_one = classifier.compute_distances_one_loop(X_test)

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

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

In [15]:
```
# Now implement the fully vectorized version inside compute_distances_no_loops
# and run the code
dists_two = classifier.compute_distances_no_loops(X_test)

# check that the distance matrix agrees with the one we computed before:
difference = np.linalg.norm(dists - dists_two, ord='fro')
print('No loop difference was: %f' % (difference, ))
if difference < 0.001:
    print('Good! The distance matrices are the same')
else:
    print('Uh-oh! The distance matrices are different')
```

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

In [16]:
```
# Let's compare how fast the implementations are
def time_function(f, *args):
    """
    Call a function f with args and return the time (in seconds) that it took to execute.
    """
    import time
    tic = time.time()
    f(*args)
    toc = time.time()
    return toc - tic

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

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

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

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

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

```
Two loop version took 43.303088 seconds
One loop version took 95.277233 seconds
No loop version took 0.523000 seconds
```

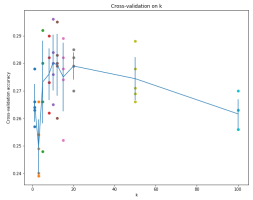
Cross-validation

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 [17]:
```
num_folds = 5
k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50, 100]

X_train_folds = []
y_train_folds = []
################################################################################
# TODO:                                                                        #
# Split up the training data into folds. After splitting, X_train_folds and   #
# y_train_folds should each be lists of length num_folds, where               #
# y_train_folds[i] is the label vector for the points in X_train_folds[i].     #
# Hint: Look up the numpy array_split function.                                #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

# pass
X_train_folds = np.array_split(X_train, num_folds)
y_train_folds = np.array_split(y_train, num_folds)

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

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


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

#pass
classifier = KNearestNeighbor()
for k in k_choices:
    accuracies = []
    for fold in range(num_folds):
        temp_X = X_train_folds[:]
        temp_y = y_train_folds[:]
        X_val_fold = temp_X.pop(fold)
        y_val_fold = temp_y.pop(fold)
        temp_X = np.array([y for x in temp_X for y in x])
        temp_y = np.array([y for x in temp_y for y in x])
        classifier.train(temp_X, temp_y)
        y_val_pred = classifier.predict(X_val_fold,k=k)
        num_correct = np.sum(y_val_fold == y_val_pred)
        accuracies.append(num_correct / y_val_fold.shape[0])
    k_to_accuracies[k] = accuracies

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

# Print out the computed accuracies
for k in sorted(k_to_accuracies):
    for accuracy in k_to_accuracies[k]:
        print('k = %d, accuracy = %f' % (k, accuracy))
```

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

In [18]:
```
# 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 [19]:
```
# Based on the cross-validation results above, choose the best value for k,
# retrain the classifier using all the training data, and test it on the test
# data. You should be able to get above 28% accuracy on the test data.
best_k = 10

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

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

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

Inline Question 3

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

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

YourAnswer : 2, 4

YourExplanation : *1. KNN is not a linear classifier and the decision of it is locally linear. 2. The training error of a 1-NN is 0 while that of 5-NN is changing according to the rules of the vote. 3. If there is noise in some data, it is easy for 1-NN to overfit. 4. Because the volume of searches increases.*

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-ipython
%load_ext autoreload
%autoreload 2
```

CIFAR-10 Data Loading and Preprocessing

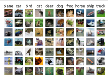
In [2]:
```
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

# 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
```

In [3]:
```
X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)
```

In [4]:
```
# 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 [5]:
```
# 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 [6]:
```
# 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)
```

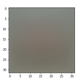
```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

In [7]:
```
# 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 [8]:
```
# Preprocessing: subtract the mean image
# first: compute the image mean based on the training data
mean_image = np.mean(X_train, axis=0)
print(mean_image[:10]) # print a few of the elements
plt.figure(figsize=(4,4))
plt.imshow(mean_image.reshape((32,32,3)).astype('uint8')) # visualize the mean image
plt.show()
```

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



In [9]:
```
# 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 [10]:
```
# 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 [11]:
```
# Evaluate the naive implementation of the loss we provided for you:
from cs231n.classifiers.linear_svm import svm_loss_naive
import time

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

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

```
loss: 9.254637
```

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 [12]:
```
# Once you've implemented the gradient, recompute it with the code below
# and gradient check it with the function we provided for you

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

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

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

```
numerical: 2.452031 analytic: 2.452031, relative error: 7.571552e-11
numerical: 19.209757 analytic: 19.209757, relative error: 1.740935e-11
numerical: 29.990930 analytic: 29.990930, relative error: 2.262353e-11
numerical: 5.004458 analytic: 5.004458, relative error: 1.726856e-13
numerical: 9.109520 analytic: 9.109520, relative error: 3.652644e-12
numerical: 14.638411 analytic: 14.638411, relative error: 2.570840e-11
numerical: 0.384930 analytic: 0.384930, relative error: 8.239109e-10
numerical: 9.042550 analytic: 9.040080, relative error: 1.365907e-04
numerical: 5.031353 analytic: 5.031353, relative error: 7.011881e-11
numerical: -0.491690 analytic: -0.491690, relative error: 9.814800e-11
numerical: -7.813292 analytic: -7.798386, relative error: 9.547834e-04
numerical: 13.373353 analytic: 13.374432, relative error: 4.030998e-05
numerical: 2.693615 analytic: 2.690084, relative error: 6.559506e-04
numerical: 30.081313 analytic: 30.077136, relative error: 6.943178e-05
numerical: 1.441859 analytic: 1.436947, relative error: 1.706308e-03
numerical: 13.884603 analytic: 13.880282, relative error: 1.556285e-04
numerical: -12.100700 analytic: -11.996633, relative error: 4.318584e-03
numerical: 12.511174 analytic: 12.507256, relative error: 1.566297e-04
numerical: 21.756433 analytic: 21.756358, relative error: 1.717397e-06
numerical: -19.071731 analytic: -19.077124, relative error: 1.413771e-04
```

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: *The gradient in the code is computed numerically. Because the max function is continuous but not differentiable at 0, the calculated computed gradient may be different from the analytically computed one.*

In [13]:
```
# Next implement the function svm_loss_vectorized; for now only compute the loss;
# we will implement the gradient in a moment.
tic = time.time()
loss_naive, grad_naive = svm_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Naive loss: %e computed in %fs' % (loss_naive, toc - tic))

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

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

```
Naive loss: 9.254637e+00 computed in 0.195888s
Vectorized loss: 9.254637e+00 computed in 0.333811s
difference: 0.000000
```

In [14]:
```
# 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 0.167899s
Vectorized loss and gradient: computed in 0.026988s
difference: 0.000000
```

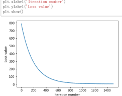
Stochastic Gradient Descent

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 [15]:
```
# 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 788.315377
iteration 100 / 1500: loss 471.521338
iteration 200 / 1500: loss 287.290971
iteration 300 / 1500: loss 175.350124
iteration 400 / 1500: loss 107.898217
iteration 500 / 1500: loss 67.116508
iteration 600 / 1500: loss 42.116039
iteration 700 / 1500: loss 27.434075
iteration 800 / 1500: loss 18.703082
iteration 900 / 1500: loss 13.584855
iteration 1000 / 1500: loss 10.243567
iteration 1100 / 1500: loss 8.054647
iteration 1200 / 1500: loss 7.244570
iteration 1300 / 1500: loss 6.767564
iteration 1400 / 1500: loss 6.083590
That took 69.789852s
```

In [16]:
```
# 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 [17]:
```
# 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: 0.382857
validation accuracy: 0.396000
```

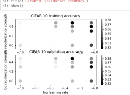
In [18]:

```python
# 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, 3e-7, 5e-7, 9e-7]
regularization_strengths = [2.5e4, 1e4, 3e4, 2e4]

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

################################################################################
# TODO:                                                                        #
# Write code that chooses the best hyperparameters by tuning on the validation #
# set. For each combination of hyperparameters, train a linear SVM on the      #
# training set, compute its accuracy on the training and validation sets, and  #
# store these numbers in the results dictionary. In addition, store the best   #
# validation accuracy in best_val and the LinearSVM object that achieves this  #
# accuracy in best_svm.                                                        #
#                                                                              #
# Hint: You should use a small value for num_iters as you develop your         #
# validation code so that the SVMs don't take much time to train; once you are #
# confident that your validation code works, you should rerun the validation   #
# code with a larger value for num_iters.                                      #
################################################################################
# Your code
num_iters = 1500
for learning_rate in learning_rates:
    for regularization_strength in regularization_strengths:
        svm = LinearSVM()
        loss_hist = svm.train(X_train, y_train, learning_rate=learning_rate, reg=regularization_strength,
                              num_iters=num_iters, verbose=True)
        y_train_pred = svm.predict(X_train)
        y_val_pred = svm.predict(X_val)
        y_train_acc = np.mean(y_train_pred == y_train)
        y_val_acc = np.mean(y_val_pred == y_val)
        results[(learning_rate,regularization_strength)] = [y_train_acc, y_val_acc]
        if y_val_acc > best_val:
            best_val = y_val_acc
            best_svm = svm
################################################################################
#                              END OF YOUR CODE                                #
################################################################################

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

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

```
iteration 0 / 1500: loss 790.219767
iteration 100 / 1500: loss 472.011884
iteration 200 / 1500: loss 286.989557
iteration 300 / 1500: loss 174.741656
iteration 400 / 1500: loss 107.018729
iteration 500 / 1500: loss 66.587612
iteration 600 / 1500: loss 42.166330
iteration 700 / 1500: loss 27.017495
iteration 800 / 1500: loss 18.447512
iteration 900 / 1500: loss 13.482718
iteration 1000 / 1500: loss 10.362644
iteration 1100 / 1500: loss 8.873749
iteration 1200 / 1500: loss 6.639200
iteration 1300 / 1500: loss 6.473011
iteration 1400 / 1500: loss 6.692804
iteration 0 / 1500: loss 334.844997
iteration 100 / 1500: loss 261.063546
iteration 200 / 1500: loss 212.969688
iteration 300 / 1500: loss 174.562281
iteration 400 / 1500: loss 142.771403
iteration 500 / 1500: loss 118.647027
iteration 600 / 1500: loss 96.281958
iteration 700 / 1500: loss 79.698489
iteration 800 / 1500: loss 65.873221
iteration 900 / 1500: loss 54.578220
iteration 1000 / 1500: loss 45.371784
iteration 1100 / 1500: loss 37.891210
iteration 1200 / 1500: loss 31.477967
iteration 1300 / 1500: loss 26.182357
iteration 1400 / 1500: loss 22.950482
iteration 0 / 1500: loss 956.265381
iteration 100 / 1500: loss 517.370490
iteration 200 / 1500: loss 282.630807
iteration 300 / 1500: loss 157.299686
iteration 400 / 1500: loss 87.431304
iteration 500 / 1500: loss 50.613561
iteration 600 / 1500: loss 29.811360
iteration 700 / 1500: loss 18.225339
iteration 800 / 1500: loss 12.222907
iteration 900 / 1500: loss 9.534534
iteration 1000 / 1500: loss 7.803495
iteration 1100 / 1500: loss 6.420227
iteration 1200 / 1500: loss 6.136582
iteration 1300 / 1500: loss 5.744283
iteration 1400 / 1500: loss 5.587963
iteration 0 / 1500: loss 643.881492
iteration 100 / 1500: loss 424.803568
iteration 200 / 1500: loss 283.898217
iteration 300 / 1500: loss 191.311775
iteration 400 / 1500: loss 128.511391
iteration 500 / 1500: loss 87.527689
iteration 600 / 1500: loss 59.669233
iteration 700 / 1500: loss 41.861217
iteration 800 / 1500: loss 29.060017
iteration 900 / 1500: loss 21.734147
iteration 1000 / 1500: loss 16.045647
iteration 1100 / 1500: loss 12.468817
iteration 1200 / 1500: loss 9.976301
iteration 1300 / 1500: loss 8.592530
iteration 1400 / 1500: loss 7.404574
iteration 0 / 1500: loss 793.328412
iteration 100 / 1500: loss 174.786129
iteration 200 / 1500: loss 41.988560
iteration 300 / 1500: loss 13.949538
iteration 400 / 1500: loss 7.389388
iteration 500 / 1500: loss 5.854769
iteration 600 / 1500: loss 5.310714
iteration 700 / 1500: loss 5.192841
iteration 800 / 1500: loss 5.428966
iteration 900 / 1500: loss 5.740763
iteration 1000 / 1500: loss 5.763526
iteration 1100 / 1500: loss 5.473073
iteration 1200 / 1500: loss 5.679251
iteration 1300 / 1500: loss 5.538323
iteration 1400 / 1500: loss 5.504355
iteration 0 / 1500: loss 328.178888
iteration 100 / 1500: loss 174.211715
iteration 200 / 1500: loss 96.063286
iteration 300 / 1500: loss 54.477176
iteration 400 / 1500: loss 31.613778
iteration 500 / 1500: loss 19.659114
iteration 600 / 1500: loss 12.877982
iteration 700 / 1500: loss 9.632663
iteration 800 / 1500: loss 7.272801
iteration 900 / 1500: loss 6.595213
iteration 1000 / 1500: loss 5.774770
iteration 1100 / 1500: loss 5.943636
iteration 1200 / 1500: loss 5.144475
iteration 1300 / 1500: loss 5.093352
iteration 1400 / 1500: loss 5.096106
iteration 0 / 1500: loss 961.775820
iteration 100 / 1500: loss 156.888932
iteration 200 / 1500: loss 30.147292
iteration 300 / 1500: loss 9.735226
iteration 400 / 1500: loss 6.858836
iteration 500 / 1500: loss 5.819287
iteration 600 / 1500: loss 5.164156
iteration 700 / 1500: loss 5.526017
iteration 800 / 1500: loss 5.296243
iteration 900 / 1500: loss 5.557637
iteration 1000 / 1500: loss 5.835592
iteration 1100 / 1500: loss 6.115562
iteration 1200 / 1500: loss 5.211916
iteration 1300 / 1500: loss 6.230010
iteration 1400 / 1500: loss 5.590479
iteration 0 / 1500: loss 649.703605
iteration 100 / 1500: loss 192.659047
iteration 200 / 1500: loss 60.568694
iteration 300 / 1500: loss 21.411031
iteration 400 / 1500: loss 10.098990
iteration 500 / 1500: loss 6.849355
iteration 600 / 1500: loss 5.402935
iteration 700 / 1500: loss 5.454206
iteration 800 / 1500: loss 5.333657
iteration 900 / 1500: loss 5.128681
iteration 1000 / 1500: loss 5.438856
iteration 1100 / 1500: loss 5.034164
iteration 1200 / 1500: loss 5.236016
iteration 1300 / 1500: loss 5.841541
iteration 1400 / 1500: loss 5.258642
iteration 0 / 1500: loss 786.439910
iteration 100 / 1500: loss 65.244665
iteration 200 / 1500: loss 10.384875
iteration 300 / 1500: loss 5.728256
iteration 400 / 1500: loss 5.980500
iteration 500 / 1500: loss 5.379235
iteration 600 / 1500: loss 5.676797
iteration 700 / 1500: loss 6.168010
iteration 800 / 1500: loss 6.119763
iteration 900 / 1500: loss 5.796301
iteration 1000 / 1500: loss 6.016598
iteration 1100 / 1500: loss 5.789551
iteration 1200 / 1500: loss 5.321337
iteration 1300 / 1500: loss 5.969726
iteration 1400 / 1500: loss 6.058110
iteration 0 / 1500: loss 329.704506
iteration 100 / 1500: loss 115.309552
iteration 200 / 1500: loss 44.449079
iteration 300 / 1500: loss 19.448898
iteration 400 / 1500: loss 10.343293
iteration 500 / 1500: loss 7.269884
iteration 600 / 1500: loss 5.910680
iteration 700 / 1500: loss 5.597869
iteration 800 / 1500: loss 5.560121
iteration 900 / 1500: loss 5.302210
iteration 1000 / 1500: loss 5.259050
iteration 1100 / 1500: loss 5.436046
iteration 1200 / 1500: loss 5.315781
iteration 1300 / 1500: loss 5.011242
iteration 1400 / 1500: loss 5.728202
iteration 0 / 1500: loss 941.231202
iteration 100 / 1500: loss 49.464871
iteration 200 / 1500: loss 8.026114
iteration 300 / 1500: loss 6.047594
iteration 400 / 1500: loss 6.022923
iteration 500 / 1500: loss 5.475224
iteration 600 / 1500: loss 5.895134
iteration 700 / 1500: loss 5.484748
iteration 800 / 1500: loss 6.013086
iteration 900 / 1500: loss 5.687881
iteration 1000 / 1500: loss 6.480750
iteration 1100 / 1500: loss 5.563863
iteration 1200 / 1500: loss 6.273441
iteration 1300 / 1500: loss 6.249854
iteration 1400 / 1500: loss 5.342856
iteration 0 / 1500: loss 635.503932
iteration 100 / 1500: loss 86.530842
iteration 200 / 1500: loss 15.393769
iteration 300 / 1500: loss 6.926607
iteration 400 / 1500: loss 6.147205
iteration 500 / 1500: loss 5.905572
iteration 600 / 1500: loss 5.444106
iteration 700 / 1500: loss 5.326389
iteration 800 / 1500: loss 6.229804
iteration 900 / 1500: loss 5.793452
iteration 1000 / 1500: loss 5.709813
iteration 1100 / 1500: loss 5.611583
iteration 1200 / 1500: loss 5.334913
iteration 1300 / 1500: loss 5.664465
iteration 1400 / 1500: loss 6.274194
iteration 0 / 1500: loss 783.290887
iteration 100 / 1500: loss 14.159910
iteration 200 / 1500: loss 6.407096
iteration 300 / 1500: loss 6.582207
iteration 400 / 1500: loss 6.453461
iteration 500 / 1500: loss 6.080634
iteration 600 / 1500: loss 7.236226
iteration 700 / 1500: loss 7.186806
iteration 800 / 1500: loss 6.553036
iteration 900 / 1500: loss 7.597359
iteration 1000 / 1500: loss 7.418662
iteration 1100 / 1500: loss 6.408434
iteration 1200 / 1500: loss 6.277035
iteration 1300 / 1500: loss 6.399299
iteration 1400 / 1500: loss 5.934452
iteration 0 / 1500: loss 332.377985
iteration 100 / 1500: loss 53.947678
iteration 200 / 1500: loss 13.122439
iteration 300 / 1500: loss 6.885925
iteration 400 / 1500: loss 5.647001
iteration 500 / 1500: loss 5.947450
iteration 600 / 1500: loss 5.432951
iteration 700 / 1500: loss 6.279119
iteration 800 / 1500: loss 6.459152
iteration 900 / 1500: loss 5.927388
iteration 1000 / 1500: loss 6.136099
iteration 1100 / 1500: loss 5.937135
iteration 1200 / 1500: loss 5.722358
iteration 1300 / 1500: loss 5.512494
iteration 1400 / 1500: loss 6.418576
iteration 0 / 1500: loss 955.513767
iteration 100 / 1500: loss 10.983862
iteration 200 / 1500: loss 7.121090
iteration 300 / 1500: loss 6.661412
iteration 400 / 1500: loss 6.811265
iteration 500 / 1500: loss 6.890177
iteration 600 / 1500: loss 6.998816
iteration 700 / 1500: loss 6.786876
iteration 800 / 1500: loss 6.939844
iteration 900 / 1500: loss 7.099917
iteration 1000 / 1500: loss 6.322078
iteration 1100 / 1500: loss 6.501955
iteration 1200 / 1500: loss 6.992972
iteration 1300 / 1500: loss 6.439532
iteration 1400 / 1500: loss 6.849745
iteration 0 / 1500: loss 635.725027
iteration 100 / 1500: loss 22.322484
iteration 200 / 1500: loss 7.478477
iteration 300 / 1500: loss 6.474384
iteration 400 / 1500: loss 5.974473
iteration 500 / 1500: loss 5.814265
iteration 600 / 1500: loss 6.505292
iteration 700 / 1500: loss 6.380384
iteration 800 / 1500: loss 6.719880
iteration 900 / 1500: loss 6.462918
iteration 1000 / 1500: loss 6.690373
iteration 1100 / 1500: loss 6.634218
iteration 1200 / 1500: loss 6.931001
iteration 1300 / 1500: loss 6.748102
iteration 1400 / 1500: loss 6.111694
lr 1.000000e-07 reg 1.000000e+04 train accuracy: 0.374429 val accuracy: 0.374000
lr 1.000000e-07 reg 2.000000e+04 train accuracy: 0.382000 val accuracy: 0.377000
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.381714 val accuracy: 0.387000
lr 1.000000e-07 reg 3.000000e+04 train accuracy: 0.379327 val accuracy: 0.382000
lr 3.000000e-07 reg 1.000000e+04 train accuracy: 0.381082 val accuracy: 0.389000
lr 3.000000e-07 reg 2.000000e+04 train accuracy: 0.381122 val accuracy: 0.385000
lr 3.000000e-07 reg 2.500000e+04 train accuracy: 0.365755 val accuracy: 0.383000
lr 3.000000e-07 reg 3.000000e+04 train accuracy: 0.360041 val accuracy: 0.371000
lr 5.000000e-07 reg 1.000000e+04 train accuracy: 0.362000 val accuracy: 0.371000
lr 5.000000e-07 reg 2.000000e+04 train accuracy: 0.364796 val accuracy: 0.363000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.346796 val accuracy: 0.370000
lr 5.000000e-07 reg 3.000000e+04 train accuracy: 0.317490 val accuracy: 0.317000
lr 9.000000e-07 reg 1.000000e+04 train accuracy: 0.334551 val accuracy: 0.342000
lr 9.000000e-07 reg 2.000000e+04 train accuracy: 0.323653 val accuracy: 0.322000
lr 9.000000e-07 reg 2.500000e+04 train accuracy: 0.317714 val accuracy: 0.337000
lr 9.000000e-07 reg 3.000000e+04 train accuracy: 0.341980 val accuracy: 0.355000
best validation accuracy achieved during cross-validation: 0.389000
```

In [19]:

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

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

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



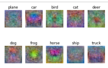
In [20]:

```python
# 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: 0.370000
```

In [21]:

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

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



Inline question 2:

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

Your answer: *Each type of weight visualization image roughly shows the shape of the object and the background color. When a shape or background color similar to this type appears in the picture, there is a high probability that it will be classified as this type.*

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-ipython
%load_ext autoreload
%autoreload 2
```

In [2]:
```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000, num_dev=500):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to 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
```

In [3]:
```
# 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
```

In [4]:
```
# 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 [5]:
```
# First implement the naive softmax loss function with nested loops.
# Open the file cs231n/classifiers/softmax.py and implement the
# softmax_loss_naive function.

from cs231n.classifiers.softmax import softmax_loss_naive
import time

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

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

```
loss: 2.405475
sanity check: 2.302585
```

Inline Question 1:

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

Your answer: *Because the initialization of w is random and the number of types is 10, so the probability of correct classification is 0.1. Therefore, we expect our loss to be close to -log(0.1).*

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

# As we did for the SVM, use numeric gradient checking as a debugging tool.
# The numeric gradient should be close to the analytic gradient.
from 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: 0.515617 analytic: 0.515617, relative error: 6.283933e-08
numerical: 1.662848 analytic: 1.662848, relative error: 4.184001e-08
numerical: 2.259963 analytic: 2.259963, relative error: 2.650463e-08
numerical: -0.073771 analytic: -0.073771, relative error: 7.692045e-07
numerical: 0.830560 analytic: 0.830560, relative error: 3.402198e-08
numerical: 3.678949 analytic: 3.678949, relative error: 1.018190e-08
numerical: -2.092716 analytic: -2.092716, relative error: 5.678137e-09
numerical: 2.715364 analytic: 2.715364, relative error: 6.194429e-09
numerical: -0.331929 analytic: -0.331929, relative error: 1.156010e-07
numerical: -0.495755 analytic: -0.495755, relative error: 2.141389e-07
numerical: -0.137596 analytic: -0.137596, relative error: 5.379353e-07
numerical: 3.251791 analytic: 3.251791, relative error: 1.046894e-08
numerical: 0.591519 analytic: 0.591518, relative error: 1.829066e-07
numerical: 0.840669 analytic: 0.840669, relative error: 6.491719e-08
numerical: 0.577753 analytic: 0.577753, relative error: 1.540781e-07
numerical: -4.225607 analytic: -4.225607, relative error: 1.050689e-08
numerical: -1.403294 analytic: -1.403294, relative error: 1.232897e-08
numerical: 3.400891 analytic: 3.400890, relative error: 2.898269e-08
numerical: 1.569563 analytic: 1.569563, relative error: 6.740954e-08
numerical: -3.675945 analytic: -3.675945, relative error: 1.039366e-08
```

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

from cs231n.classifiers.softmax import softmax_loss_vectorized
tic = time.time()
loss_vectorized, grad_vectorized = softmax_loss_vectorized(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('vectorized loss: %e computed in %fs' % (loss_vectorized, toc - tic))

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

```
naive loss: 2.405475e+00 computed in 0.166852s
vectorized loss: 2.405475e+00 computed in 0.378791s
Loss difference: 0.000000
Gradient difference: 0.000000
```

In [8]:
```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of over 0.35 on the validation set.
from cs231n.classifiers import Softmax
results = {}
best_val = -1
best_softmax = None
learning_rates = [1e-7, 5e-7]
regularization_strengths = [2.5e4, 5e4]

################################################################################
# TODO:                                                                        #
# Use the validation set to set the learning rate and regularization strength. #
# This should be identical to the validation that you did for the SVM; save    #
# the best trained softmax classifer in best_softmax.                          #
################################################################################
# Your code
num_iters = 1500
for lr in learning_rates:
    for reg in regularization_strengths:
        softmax = Softmax()
        loss_hist = softmax.train(X_train, y_train, learning_rate=lr, reg=reg,
                      num_iters=num_iters, verbose=True)
        y_train_pred = softmax.predict(X_train)
        y_val_pred = softmax.predict(X_val)
        y_train_acc = np.mean(y_train_pred == y_train)
        y_val_acc = np.mean(y_val_pred == y_val)
        results[(lr,reg)] = [y_train_acc, y_val_acc]
        if y_val_acc > best_val:
            best_val = y_val_acc
            best_softmax = softmax
################################################################################
#                              END OF YOUR CODE                                #
################################################################################

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

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

```
iteration 0 / 1500: loss 389.235007
iteration 100 / 1500: loss 234.868950
iteration 200 / 1500: loss 142.625381
iteration 300 / 1500: loss 87.051621
iteration 400 / 1500: loss 53.422824
iteration 500 / 1500: loss 33.176412
iteration 600 / 1500: loss 20.782209
iteration 700 / 1500: loss 13.349236
iteration 800 / 1500: loss 8.851855
iteration 900 / 1500: loss 6.195012
iteration 1000 / 1500: loss 4.536064
iteration 1100 / 1500: loss 3.549924
iteration 1200 / 1500: loss 2.984868
iteration 1300 / 1500: loss 2.550700
iteration 1400 / 1500: loss 2.312766
iteration 0 / 1500: loss 771.331778
iteration 100 / 1500: loss 283.508390
iteration 200 / 1500: loss 104.973731
iteration 300 / 1500: loss 39.849412
iteration 400 / 1500: loss 15.853963
iteration 500 / 1500: loss 7.090063
iteration 600 / 1500: loss 3.924109
iteration 700 / 1500: loss 2.747495
iteration 800 / 1500: loss 2.328893
iteration 900 / 1500: loss 2.146455
iteration 1000 / 1500: loss 2.134046
iteration 1100 / 1500: loss 2.153098
iteration 1200 / 1500: loss 2.040051
iteration 1300 / 1500: loss 2.078911
iteration 1400 / 1500: loss 2.089975
iteration 0 / 1500: loss 388.421974
iteration 100 / 1500: loss 32.771709
iteration 200 / 1500: loss 4.419469
iteration 300 / 1500: loss 2.248381
iteration 400 / 1500: loss 2.029577
iteration 500 / 1500: loss 2.100767
iteration 600 / 1500: loss 2.057887
iteration 700 / 1500: loss 2.059342
iteration 800 / 1500: loss 2.057983
iteration 900 / 1500: loss 2.017836
iteration 1000 / 1500: loss 1.978115
iteration 1100 / 1500: loss 2.081727
iteration 1200 / 1500: loss 2.032726
iteration 1300 / 1500: loss 2.064390
iteration 1400 / 1500: loss 2.051917
iteration 0 / 1500: loss 771.898613
iteration 100 / 1500: loss 6.865982
iteration 200 / 1500: loss 2.104260
iteration 300 / 1500: loss 2.124455
iteration 400 / 1500: loss 2.128282
iteration 500 / 1500: loss 2.099258
iteration 600 / 1500: loss 2.102733
iteration 700 / 1500: loss 2.104299
iteration 800 / 1500: loss 2.090088
iteration 900 / 1500: loss 2.105925
iteration 1000 / 1500: loss 2.081717
iteration 1100 / 1500: loss 2.061542
iteration 1200 / 1500: loss 2.107355
iteration 1300 / 1500: loss 2.063412
iteration 1400 / 1500: loss 2.112968
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.348673 val accuracy: 0.357000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.330796 val accuracy: 0.345000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.334857 val accuracy: 0.350000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.330939 val accuracy: 0.335000
best validation accuracy achieved during cross-validation: 0.357000
```

In [9]:
```
# evaluate on test set
# Evaluate the best softmax on test set
y_test_pred = best_softmax.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('softmax on raw pixels final test set accuracy: %f' % (test_accuracy, ))
```

```
softmax on raw pixels final test set accuracy: 0.362000
```

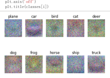
Inline Question - *True or False*

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

Your answer: True.

Your explanation: As for SVM, if the newly added test pictures are classified correctly, the loss is 0. As for Softmax, whether the classification is correct, the loss will always exist.

In [10]:
```
# 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])
```



Implementing a Neural Network

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

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

import numpy as np
import matplotlib.pyplot as plt

from cs231n.classifiers.neural_net import TwoLayerNet

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-ipython
%load_ext autoreload
%autoreload 2

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

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

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

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

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

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

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

Forward pass: compute scores

Open the file `cs231n/classifiers/neural_net.py` and look at the method `TwoLayerNet.loss`. This function is very similar to the loss functions you have written for the SVM and Softmax exercises: It takes the data and weights and computes the class scores, the loss, and the gradients on the parameters.

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

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

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

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

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

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

Forward pass: compute loss

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

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

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

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

Backward pass

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

In [5]:
```
from cs231n.gradient_check import eval_numerical_gradient

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

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

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

```
W2 max relative error: 3.440708e-09
b2 max relative error: 3.865091e-11
W1 max relative error: 3.561318e-09
b1 max relative error: 1.555470e-09
```

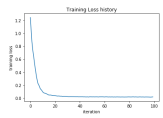
Train the network

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 [6]:
```
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 [7]:
```
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
    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
```

In [8]:
```
# 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
```

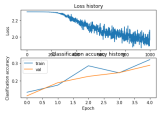
Debug the training

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('Classification 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 [12]:
```
best_net = None # store the best model into this

#################################################################################
# TODO: Tune hyperparameters using the validation set. Store your best trained  #
# model in best_net.                                                            #
#                                                                               #
# To help debug your network, it may help to use visualizations similar to the  #
# ones we used above; these visualizations will have significant qualitative    #
# differences from the ones we saw above for the poorly tuned network.          #
#                                                                               #
# Tweaking hyperparameters by hand can be fun, but you might find it useful to  #
# write code to sweep through possible combinations of hyperparameters          #
# automatically like we did on the previous exercises.                          #
#################################################################################
# Your code
best_val = -1
best_stats = None
learning_rates = [1e-3, 1e-4]
regularization_strengths = [0.4, 0.5, 0.6, 0.7]
results = {}
iters = 2000
for lr in learning_rates:
    for rs in regularization_strengths:
        net = TwoLayerNet(input_size, hidden_size, num_classes)

        # Train the network
        stats = net.train(X_train, y_train, X_val, y_val,
                    num_iters=iters, batch_size=200,
                    learning_rate=lr, learning_rate_decay=0.95,
                    reg=rs)

        y_train_pred = net.predict(X_train)
        acc_train = np.mean(y_train == y_train_pred)
        y_val_pred = net.predict(X_val)
        acc_val = np.mean(y_val == y_val_pred)

        results[(lr, rs)] = (acc_train, acc_val)

        if best_val < acc_val:
            best_stats = stats
            best_val = acc_val
            best_net = net

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

print ('best validation accuracy achieved during cross-validation: ', best_val)
#################################################################################
#                               END OF YOUR CODE                                #
#################################################################################
```

```
C:\Users\Administrator\AI_Assignment\cs231n\classifiers\neural_net.py:102: RuntimeWarning: divide by zero encountered in log
  loss = -np.log(scores[np.arange(N), y]).sum()
C:\Users\Administrator\AI_Assignment\cs231n\classifiers\neural_net.py:99: RuntimeWarning: overflow encountered in subtract
  scores -= np.max(scores, axis=1, keepdims=True)  # 数值稳定性
C:\Users\Administrator\AI_Assignment\cs231n\classifiers\neural_net.py:99: RuntimeWarning: invalid value encountered in subtract
  scores -= np.max(scores, axis=1, keepdims=True)  # 数值稳定性
C:\Users\Administrator\AI_Assignment\cs231n\classifiers\neural_net.py:80: RuntimeWarning: invalid value encountered in greater
  ol_act = (al > 0) * al
C:\Users\Administrator\AI_Assignment\cs231n\classifiers\neural_net.py:125: RuntimeWarning: invalid value encountered in greater
  dol = (al > 0) * dol

lr  0.001 reg  0.4 train accuracy:  0.5195714285714429 val accuracy:  0.471
lr  0.001 reg  0.5 train accuracy:  0.5117142857142857 val accuracy:  0.485
lr  0.001 reg  0.6 train accuracy:  0.509 val accuracy:  0.488
lr  0.001 reg  0.7 train accuracy:  0.4988975918367347 val accuracy:  0.48
lr  0.005 reg  0.4 train accuracy:  0.10028539612244898 val accuracy:  0.087
lr  0.005 reg  0.5 train accuracy:  0.10028539612244898 val accuracy:  0.087
lr  0.005 reg  0.6 train accuracy:  0.10028539612244898 val accuracy:  0.087
lr  0.005 reg  0.7 train accuracy:  0.10028539612244898 val accuracy:  0.087
best validation accuracy achieved during cross-validation:  0.488
```

In [13]:
```
# 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 [14]:
```
test_acc = (best_net.predict(X_test) == y_test).mean()
print('Test accuracy: ', test_acc)
```

```
Test accuracy:  0.494
```

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: Training on a larger dataset and increasing the regularization strength can both improve the generalization ability of the model, but adding more hidden units makes the model more overfitting.