svm

January 11, 2016

1 Multiclass Support Vector Machine exercise

Complete and hand in this completed worksheet (including its outputs and any supporting code outside of the worksheet) with your assignment submission. For more details see the assignments page on the course website.

In this exercise you will:

- implement a fully-vectorized **loss function** for the SVM
- implement the fully-vectorized expression for its **analytic gradient**
- **check your implementation** using numerical gradient
- use a validation set to **tune the learning rate and regularization** strength
- **optimize** the loss function with **SGD**
- **visualize** the final learned weights

In [38]:
```
# Run some setup code for this notebook.

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

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

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

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

1.1 CIFAR-10 Data Loading and Preprocessing

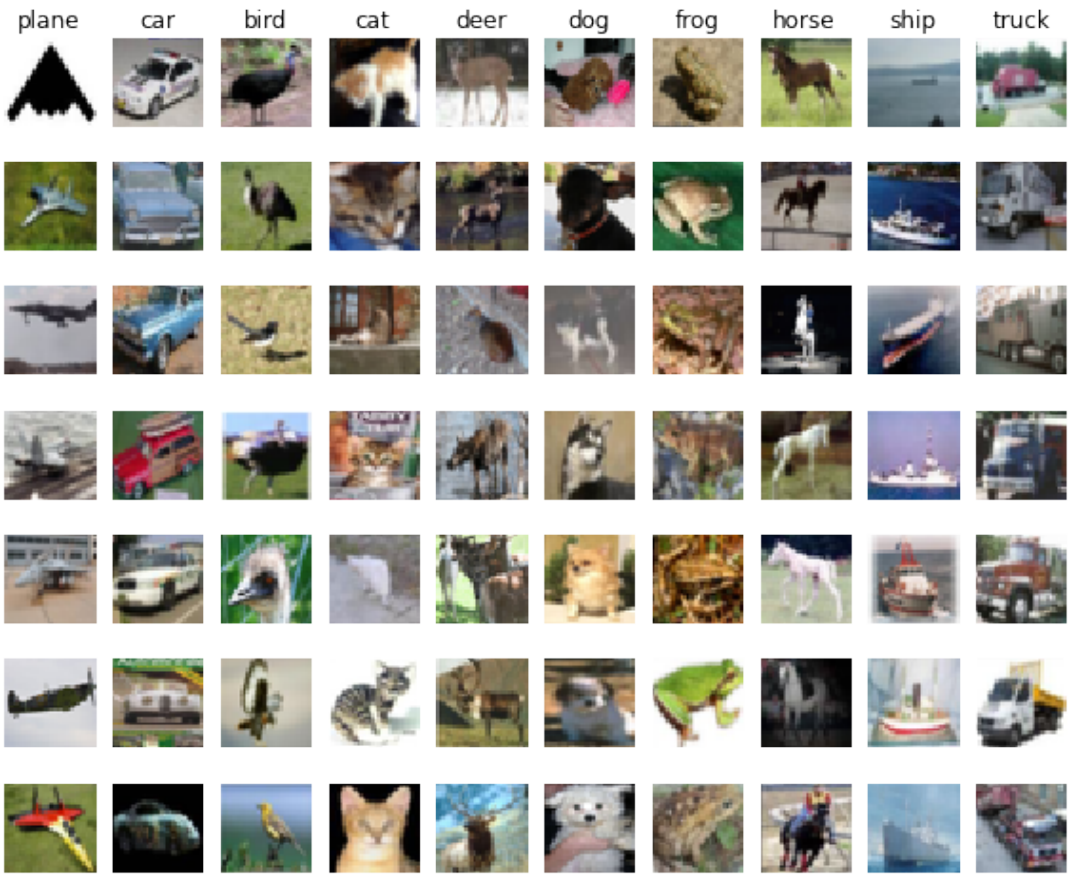
In [39]:
```
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'
X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

# 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 [40]:
```
# 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 [41]:
```
# Subsample the data for more efficient code execution in this exercise.
num_training = 49000
num_validation = 1000
num_test = 1000

# 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 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 [42]:
```
# 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))

# 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
```

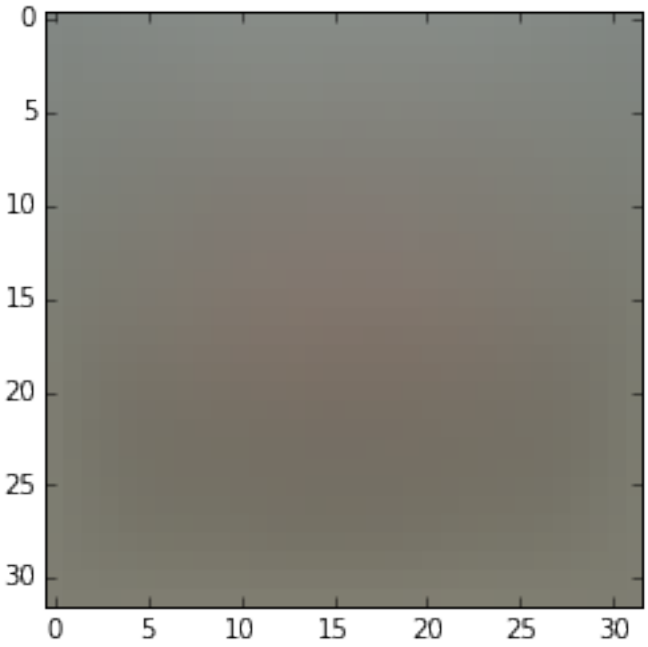
```
Training data shape:  (49000, 3072)
Validation data shape:  (1000, 3072)
Test data shape:  (1000, 3072)
```

In [43]:
```
# 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
```

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

Out[43]: <matplotlib.image.AxesImage at 0x107209190>



In [44]:
```
# second: subtract the mean image from train and test data
X_train -= mean_image
X_val -= mean_image
X_test -= mean_image
```

In [45]:
```
# 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.
# Also, lets transform both data matrices so that each image is a column.
X_train = np.hstack([X_train, np.ones((X_train.shape[0], 1))]).T
X_val = np.hstack([X_val, np.ones((X_val.shape[0], 1))]).T
X_test = np.hstack([X_test, np.ones((X_test.shape[0], 1))]).T

print X_train.shape, X_val.shape, X_test.shape
```

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

1.2 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 [46]:
```
# 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
np.random.seed(100) # ALLAN MODIFICATION
W = np.random.randn(10, 3073) * 0.0001
loss, grad = svm_loss_naive(W, X_train, y_train, 0.00001)
print 'loss: %f' % (loss, )
```

```
loss: 8.969550
```

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 [47]:
```
# 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_train, y_train, 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_train, y_train, 0.0)[0]
grad_numerical = grad_check_sparse(f, W, grad, 10)
```

```
numerical: 3.472757 analytic: 3.476673, relative error: 5.635208e-04
numerical: 0.405994 analytic: 0.410457, relative error: 5.466401e-03
numerical: 16.071068 analytic: 16.068990, relative error: 6.463916e-05
numerical: -7.456373 analytic: -7.455881, relative error: 3.294830e-05
numerical: -0.774292 analytic: -0.774434, relative error: 9.141859e-05
numerical: -0.568985 analytic: -0.567781, relative error: 1.059294e-03
numerical: 16.078500 analytic: 16.074582, relative error: 1.218388e-04
numerical: -3.200495 analytic: -3.201440, relative error: 1.476227e-04
numerical: -11.848588 analytic: -11.848210, relative error: 1.595628e-05
numerical: 3.734784 analytic: 3.733046, relative error: 2.327496e-04
```

1.2.1 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? *Hint: the SVM loss function is not strictly speaking differentiable*

Your Answer: *fill this in.*

In [48]:
```
# 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_train, y_train, 0.00001)
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_train, y_train, 0.00001)
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: 8.969550e+00 computed in 2.698734s
Vectorized loss: 8.969550e+00 computed in 0.368170s
difference: -0.000000
```

```
In [49]: # 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_train, y_train, 0.00001)
         toc = time.time()
         print 'Naive loss and gradient: computed in %fs' % (toc - tic)

         tic = time.time()
         _, grad_vectorized = svm_loss_vectorized(W, X_train, y_train, 0.00001)
         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 2.703949s
Vectorized loss and gradient: computed in 0.555753s
difference: 0.000000
```

1.2.2 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.

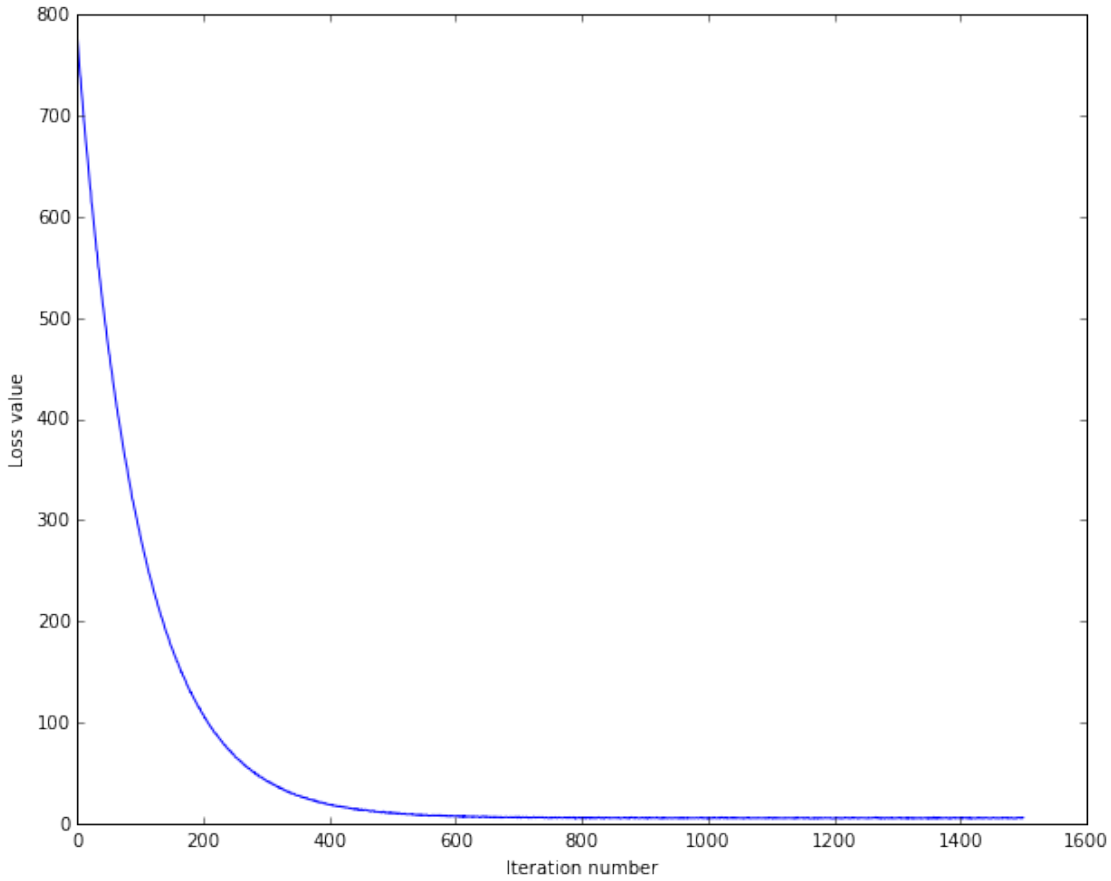
```
In [50]: # Now implement SGD in LinearSVM.train() function and run it with the code below
         from cs231n.classifiers import LinearSVM
         svm = LinearSVM()
         tic = time.time()
         loss_hist = svm.train(X_train, y_train, learning_rate=1e-7, reg=5e4,
                               num_iters=1500, verbose=True)
         toc = time.time()
         print 'That took %fs' % (toc - tic)
```

```
iteration 0 / 1500: loss 781.732139
iteration 100 / 1500: loss 285.290705
```

```
iteration 200 / 1500: loss 107.789131
iteration 300 / 1500: loss 42.544205
iteration 400 / 1500: loss 18.687729
iteration 500 / 1500: loss 10.375392
iteration 600 / 1500: loss 7.384117
iteration 700 / 1500: loss 6.062107
iteration 800 / 1500: loss 5.636144
iteration 900 / 1500: loss 5.276416
iteration 1000 / 1500: loss 5.724765
iteration 1100 / 1500: loss 5.473760
iteration 1200 / 1500: loss 5.107062
iteration 1300 / 1500: loss 5.470051
iteration 1400 / 1500: loss 5.964795
That took 5.326751s
```

In [51]:
```
# 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')
```

Out[51]: <matplotlib.text.Text at 0x10b416410>



In [52]:
```
# 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: 0.364163
validation accuracy: 0.377000
```

In [53]:
```
# 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 = [5e-8, 5e-7]
regularization_strengths = [5e4, 1e5]

# 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.                                      #
################################################################################
import itertools as it
import copy

for l_r, reg in it.product(learning_rates, regularization_strengths):
    loss_hist = svm.train(X_train, y_train, learning_rate=l_r, reg=reg,
                          num_iters=1500, verbose=True)
    y_train_pred = svm.predict(X_train)
    y_val_pred = svm.predict(X_val)

    train_acc = np.mean(y_train == y_train_pred)
    val_acc = np.mean(y_val == y_val_pred)

    if val_acc > best_val:
        best_val = val_acc
        best_svm = copy.copy(svm)
```

```
            results[(l_r, reg)] = (train_acc, val_acc)

        ################################################################################
        #                                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 5.277755
iteration 100 / 1500: loss 4.934534
iteration 200 / 1500: loss 5.269886
iteration 300 / 1500: loss 5.883307
iteration 400 / 1500: loss 4.839533
iteration 500 / 1500: loss 5.383470
iteration 600 / 1500: loss 5.499386
iteration 700 / 1500: loss 4.973608
iteration 800 / 1500: loss 5.627622
iteration 900 / 1500: loss 5.586436
iteration 1000 / 1500: loss 5.274968
iteration 1100 / 1500: loss 4.980725
iteration 1200 / 1500: loss 5.306225
iteration 1300 / 1500: loss 5.374312
iteration 1400 / 1500: loss 5.011241
iteration 0 / 1500: loss 5.693182
iteration 100 / 1500: loss 5.064019
iteration 200 / 1500: loss 5.845920
iteration 300 / 1500: loss 5.376172
iteration 400 / 1500: loss 5.437568
iteration 500 / 1500: loss 5.580521
iteration 600 / 1500: loss 5.601341
iteration 700 / 1500: loss 5.626216
iteration 800 / 1500: loss 5.428513
iteration 900 / 1500: loss 5.800692
iteration 1000 / 1500: loss 5.637468
iteration 1100 / 1500: loss 5.716574
iteration 1200 / 1500: loss 6.145699
iteration 1300 / 1500: loss 5.645870
iteration 1400 / 1500: loss 5.603244
iteration 0 / 1500: loss 5.615385
iteration 100 / 1500: loss 5.955003
iteration 200 / 1500: loss 5.628470
iteration 300 / 1500: loss 6.226592
iteration 400 / 1500: loss 5.884577
iteration 500 / 1500: loss 5.499377
iteration 600 / 1500: loss 5.437213
iteration 700 / 1500: loss 6.360736
iteration 800 / 1500: loss 5.924417
iteration 900 / 1500: loss 6.220029
```

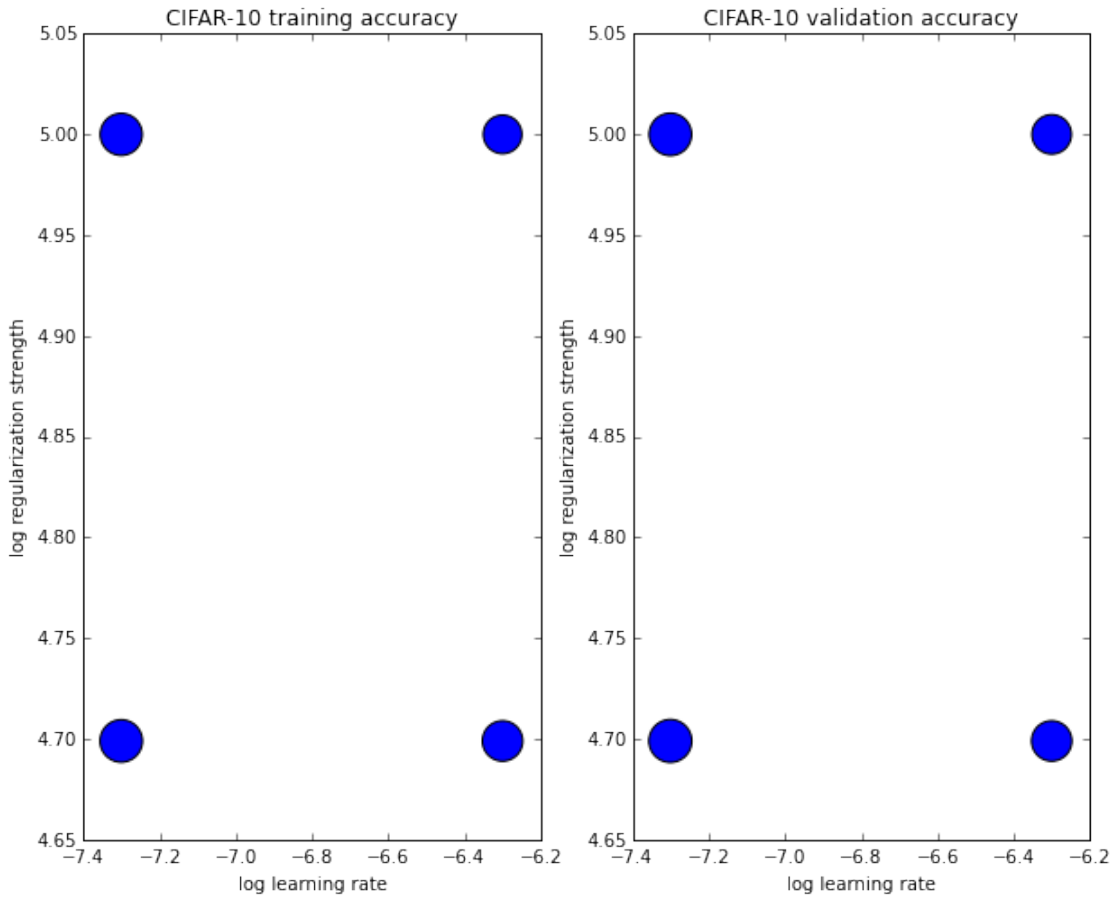
```
iteration 1000 / 1500: loss 4.887896
iteration 1100 / 1500: loss 5.846160
iteration 1200 / 1500: loss 6.144114
iteration 1300 / 1500: loss 5.808904
iteration 1400 / 1500: loss 5.517840
iteration 0 / 1500: loss 6.393931
iteration 100 / 1500: loss 5.501673
iteration 200 / 1500: loss 6.039688
iteration 300 / 1500: loss 5.752922
iteration 400 / 1500: loss 5.806513
iteration 500 / 1500: loss 6.251013
iteration 600 / 1500: loss 6.109306
iteration 700 / 1500: loss 6.258964
iteration 800 / 1500: loss 6.284725
iteration 900 / 1500: loss 5.446260
iteration 1000 / 1500: loss 5.987198
iteration 1100 / 1500: loss 6.677108
iteration 1200 / 1500: loss 5.948872
iteration 1300 / 1500: loss 5.899338
iteration 1400 / 1500: loss 6.208671
lr 5.000000e-08 reg 5.000000e+04 train accuracy: 0.369306 val accuracy: 0.377000
lr 5.000000e-08 reg 1.000000e+05 train accuracy: 0.361980 val accuracy: 0.370000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.333633 val accuracy: 0.333000
lr 5.000000e-07 reg 1.000000e+05 train accuracy: 0.305082 val accuracy: 0.319000
best validation accuracy achieved during cross-validation: 0.377000
```

In [54]:
```
# 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
sz = [results[x][0]*1500 for x in results] # default size of markers is 20
plt.subplot(1,2,1)
plt.scatter(x_scatter, y_scatter, sz)
plt.xlabel('log learning rate')
plt.ylabel('log regularization strength')
plt.title('CIFAR-10 training accuracy')

# plot validation accuracy
sz = [results[x][1]*1500 for x in results] # default size of markers is 20
plt.subplot(1,2,2)
plt.scatter(x_scatter, y_scatter, sz)
plt.xlabel('log learning rate')
plt.ylabel('log regularization strength')
plt.title('CIFAR-10 validation accuracy')
```

Out[54]:
```
<matplotlib.text.Text at 0x10eafbd90>
```

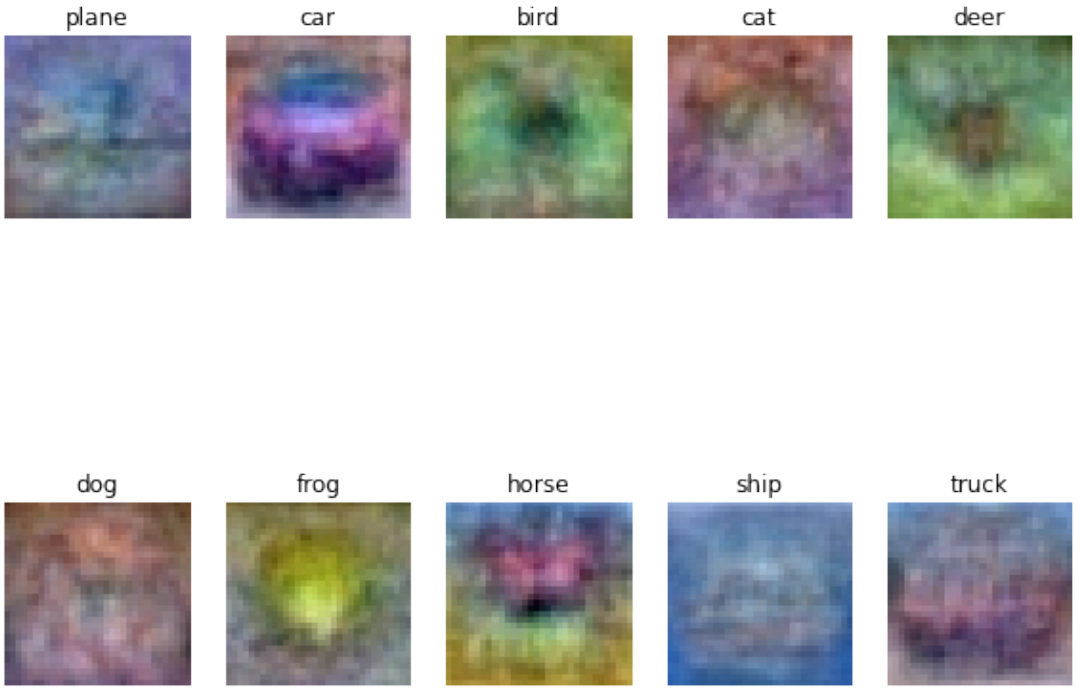



```
In [55]: # Evaluate the best svm on test set
         y_test_pred = best_svm.predict(X_test)
         test_accuracy = np.mean(y_test == y_test_pred)
         print 'linear SVM on raw pixels final test set accuracy: %f' % test_accuracy

linear SVM on raw pixels final test set accuracy: 0.295000

In [56]: # 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(10, 32, 32, 3)
         w_min, w_max = np.min(w), np.max(w)
         classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
         for i in xrange(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])
```



1.2.3 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: *fill this in*

```
In [ ]:
```