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 (https://compsci682-fa19.github.io/assignments2019/assignment1/)](https://compsci682-fa19.github.io/assignments2019/assignment1/) 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 [24]:
```
# Run some setup code for this notebook.
from __future__ import print_function
import random
import numpy as np
from cs682.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 pl
ots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

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

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

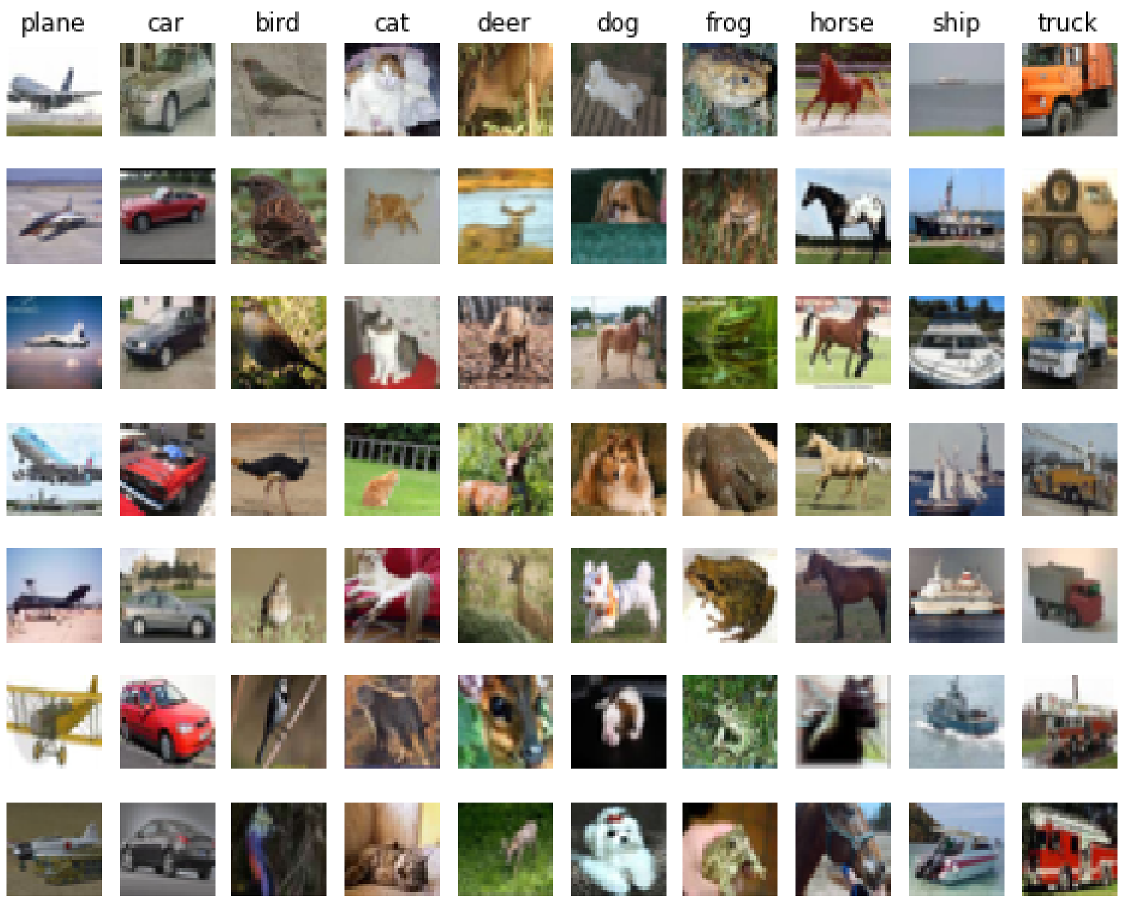
CIFAR-10 Data Loading and Preprocessing

In [25]:
```
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs682/datasets/cifar-10-batches-py'

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

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

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

```
Clear previously loaded data.
Training data shape:  (50000, 32, 32, 3)
Training labels shape:  (50000,)
Test data shape:  (10000, 32, 32, 3)
Test labels shape:  (10000,)
```

In [26]:

```
# 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', 'hor
se', '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 [27]:

```
# 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 origin
al
# 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)
```

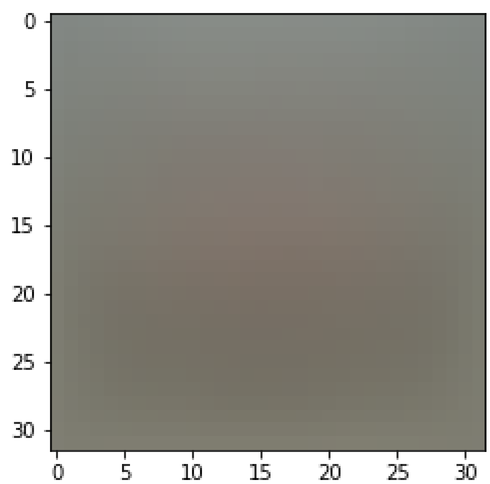
```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

In [28]:
```
# 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 [29]:
```
# 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 [30]:
```
# 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 [31]:
```
# 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 **cs682/classifiers/linear_svm.py**.

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

In [32]:
```
# Evaluate the naive implementation of the loss we provided for you:
from cs682.classifiers.linear_svm import svm_loss_naive
import time

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

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

```
loss: 8.589740
```

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 [33]:
```
# Once you've implemented the gradient, recompute it with the code be
low
# and gradient check it with the function we provided for you

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

# Numerically compute the gradient along several randomly chosen dime
nsions, and
# compare them with your analytically computed gradient. The numbers
 should match
# almost exactly along all dimensions.
from cs682.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: 27.485750 analytic: 27.485750, relative error: 1.388409e-1
2
numerical: 26.531183 analytic: 26.531183, relative error: 1.209201e-1
1
numerical: 9.232358 analytic: 9.232358, relative error: 6.625100e-12
numerical: 14.437954 analytic: 14.437954, relative error: 5.025255e-1
2
numerical: 22.083533 analytic: 22.083533, relative error: 1.631244e-1
1
numerical: -24.383923 analytic: -24.383923, relative error: 1.403322e
-11
numerical: 9.799457 analytic: 9.799457, relative error: 1.767147e-11
numerical: 14.759554 analytic: 14.759554, relative error: 6.750962e-1
2
numerical: 28.077529 analytic: 28.077529, relative error: 2.172559e-1
3
numerical: -1.233368 analytic: -1.233368, relative error: 6.199210e-1
1
numerical: 14.927535 analytic: 14.927535, relative error: 1.604054e-1
1
numerical: 2.365012 analytic: 2.365012, relative error: 3.384741e-11
numerical: -2.073142 analytic: -2.073142, relative error: 5.095891e-1
1
numerical: 7.855398 analytic: 7.855398, relative error: 3.405390e-11
numerical: -28.289928 analytic: -28.289928, relative error: 8.268965e
-13
numerical: 2.439470 analytic: 2.439470, relative error: 1.376263e-10
numerical: 23.936934 analytic: 23.936934, relative error: 7.175204e-1
2
numerical: -14.462851 analytic: -14.462851, relative error: 2.068563e
-11
numerical: 27.549003 analytic: 27.549003, relative error: 7.266884e-1
3
numerical: -5.732948 analytic: -5.732948, relative error: 5.745093e-1
2
```

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 loss function will not be differentiable at the point where the loss is zero, since it is max(loss,0). This is not a cause of concern as it happens rarely. For example, consider the one-dimenstional example of X = 1. Assume W=(0,-1). Now, loss is max(-x + 0 + 1, 0). For little margins above and below X = 1, we will have very different gradients, i.e. 0 and -1 respectively. However, numerial computation will have very little difference at these points. This is the reason grad check not matching exactly once in a while.

In [34]:
```
# Next implement the function svm_loss_vectorized; for now only compu
te 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 cs682.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 b
e much faster.
print('difference: %f' % (loss_naive - loss_vectorized))
```

```
Naive loss: 8.589740e+00 computed in 0.066196s
Vectorized loss: 8.589740e+00 computed in 0.002009s
difference: 0.000000
```

In [35]:
```
# 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 m
atch, 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 co
mputed
# by the two implementations. The gradient on the other hand is a mat
rix, 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 0.065558s
Vectorized loss and gradient: computed in 0.002451s
difference: 0.000000
```

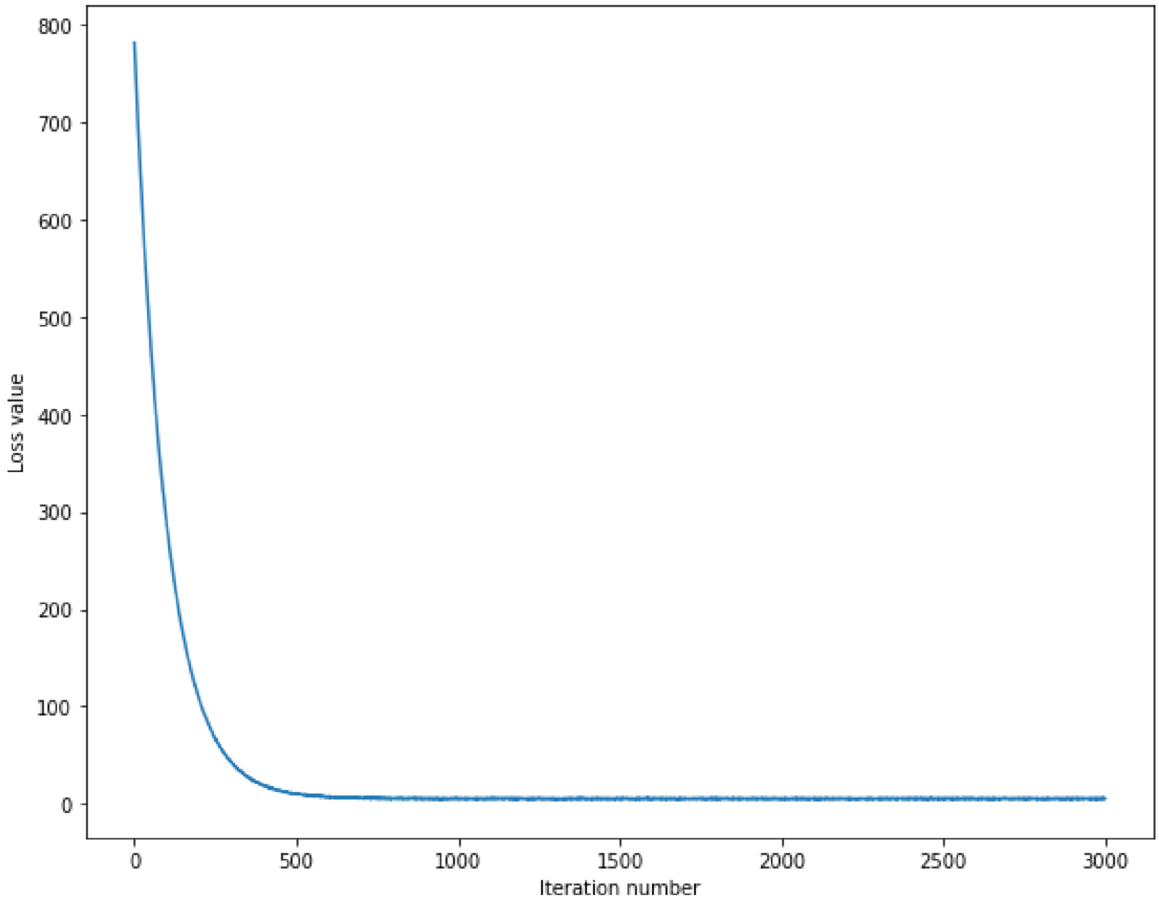
Stochastic Gradient Descent

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 [36]:
```
# In the file linear_classifier.py, implement SGD in the function
# LinearClassifier.train() and then run it with the code below.
from cs682.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=3000, verbose=True)
toc = time.time()
print('That took %fs' % (toc - tic))
```

```
iteration 0 / 3000: loss 781.744439
iteration 100 / 3000: loss 284.328956
iteration 200 / 3000: loss 106.640172
iteration 300 / 3000: loss 41.868259
iteration 400 / 3000: loss 19.077488
iteration 500 / 3000: loss 10.456769
iteration 600 / 3000: loss 6.864472
iteration 700 / 3000: loss 6.292737
iteration 800 / 3000: loss 5.695862
iteration 900 / 3000: loss 5.705327
iteration 1000 / 3000: loss 4.971040
iteration 1100 / 3000: loss 5.027295
iteration 1200 / 3000: loss 5.336624
iteration 1300 / 3000: loss 5.030771
iteration 1400 / 3000: loss 5.618442
iteration 1500 / 3000: loss 5.526525
iteration 1600 / 3000: loss 5.549422
iteration 1700 / 3000: loss 5.443943
iteration 1800 / 3000: loss 5.853622
iteration 1900 / 3000: loss 5.060834
iteration 2000 / 3000: loss 5.019086
iteration 2100 / 3000: loss 4.903166
iteration 2200 / 3000: loss 5.165892
iteration 2300 / 3000: loss 5.042505
iteration 2400 / 3000: loss 6.020268
iteration 2500 / 3000: loss 4.640616
iteration 2600 / 3000: loss 4.889599
iteration 2700 / 3000: loss 5.374185
iteration 2800 / 3000: loss 5.169209
iteration 2900 / 3000: loss 5.492799
That took 4.533447s
```

In [37]:
```
# 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 [38]:
```
# Write the LinearSVM.predict function and evaluate the performance o
n 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: 0.371327
validation accuracy: 0.384000
```

In [39]:
```
# Use the validation set to tune hyperparameters (regularization stre
ngth and
# learning rate). You should experiment with different ranges for the
learning
# rates and regularization strengths; if you are careful you should b
e able to
# get a classification accuracy of about 0.4 on the validation set.
learning_rates = [1e-8, 5e-6]
regularization_strengths = [2e3, 5e4]

# 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 th
e fraction
# of data points that are correctly classified.
results = {}
best_val = -1   # The highest validation accuracy that we have seen s
o far.
best_svm = None # The LinearSVM object that achieved the highest vali
dation rate.

####################################################################
###########
# TODO:
#
# Write code that chooses the best hyperparameters by tuning on the v
alidation #
# set. For each combination of hyperparameters, train a linear SVM on
the      #
# training set, compute its accuracy on the training and validation s
ets, and  #
# store these numbers in the results dictionary. In addition, store t
he best   #
# validation accuracy in best_val and the LinearSVM object that achie
ves this  #
# accuracy in best_svm.
#
#
#
# Hint: You should use a small value for num_iters as you develop you
r         #
# validation code so that the SVMs don't take much time to train; onc
e you are #
# confident that your validation code works, you should rerun the val
idation   #
# code with a larger value for num_iters.
#
####################################################################
###########
# Your code
# learning_rate = 2e-8
# reg = 4e3
num_iters = 10
for it in range(num_iters):
    for jt in range(num_iters):
        svm = LinearSVM()
```

```
        learning_rate = learning_rates[0] + it * ((learning_rates[1]
- learning_rates[0]) / num_iters)
        reg = regularization_strengths[0] + jt * ((regularization_str
engths[1] - regularization_strengths[0])/ num_iters)
        loss_hist = svm.train(X_train, y_train, learning_rate=learnin
g_rate, reg=reg,
                          num_iters=3000, verbose=False)
        y_train_pred = svm.predict(X_train)
        y_val_pred = svm.predict(X_val)
        training_accuracy = np.mean(y_train == y_train_pred)
        validation_accuracy = np.mean(y_val == y_val_pred)
        results[(learning_rate, reg)] = (training_accuracy, validatio
n_accuracy)
#         reg = reg + 0.05e3
        if validation_accuracy > best_val:
            best_val = validation_accuracy
            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)
```

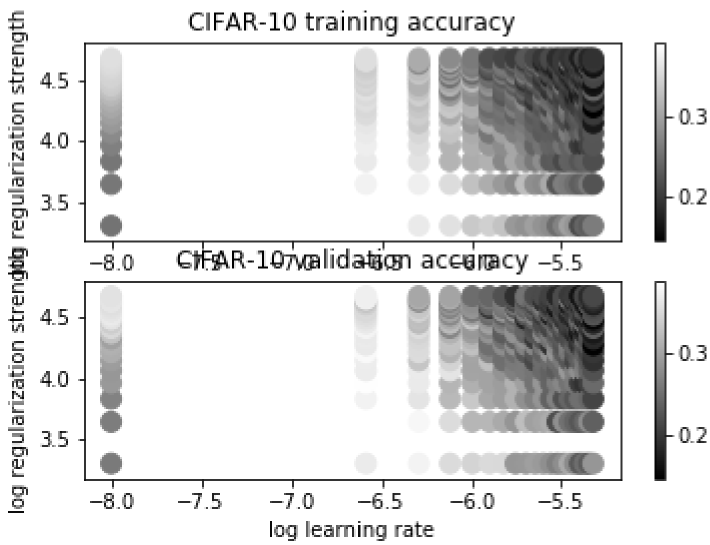
```
lr 1.000000e-08 reg 2.000000e+03 train accuracy: 0.252061 val accurac
y: 0.261000
lr 1.000000e-08 reg 6.800000e+03 train accuracy: 0.260714 val accurac
y: 0.253000
lr 1.000000e-08 reg 1.160000e+04 train accuracy: 0.283347 val accurac
y: 0.291000
lr 1.000000e-08 reg 1.640000e+04 train accuracy: 0.295592 val accurac
y: 0.294000
lr 1.000000e-08 reg 2.120000e+04 train accuracy: 0.319633 val accurac
y: 0.333000
lr 1.000000e-08 reg 2.600000e+04 train accuracy: 0.327694 val accurac
y: 0.336000
lr 1.000000e-08 reg 3.080000e+04 train accuracy: 0.341837 val accurac
y: 0.365000
lr 1.000000e-08 reg 3.560000e+04 train accuracy: 0.350143 val accurac
y: 0.362000
lr 1.000000e-08 reg 4.040000e+04 train accuracy: 0.356755 val accurac
y: 0.368000
lr 1.000000e-08 reg 4.520000e+04 train accuracy: 0.358449 val accurac
y: 0.374000
lr 5.090000e-07 reg 2.000000e+03 train accuracy: 0.378980 val accurac
y: 0.385000
lr 5.090000e-07 reg 6.800000e+03 train accuracy: 0.369755 val accurac
y: 0.374000
lr 5.090000e-07 reg 1.160000e+04 train accuracy: 0.333367 val accurac
y: 0.328000
lr 5.090000e-07 reg 1.640000e+04 train accuracy: 0.338143 val accurac
y: 0.344000
lr 5.090000e-07 reg 2.120000e+04 train accuracy: 0.343633 val accurac
y: 0.357000
lr 5.090000e-07 reg 2.600000e+04 train accuracy: 0.336755 val accurac
y: 0.343000
lr 5.090000e-07 reg 3.080000e+04 train accuracy: 0.335857 val accurac
y: 0.334000
lr 5.090000e-07 reg 3.560000e+04 train accuracy: 0.330980 val accurac
y: 0.339000
lr 5.090000e-07 reg 4.040000e+04 train accuracy: 0.320184 val accurac
y: 0.325000
lr 5.090000e-07 reg 4.520000e+04 train accuracy: 0.326184 val accurac
y: 0.349000
lr 1.008000e-06 reg 2.000000e+03 train accuracy: 0.369388 val accurac
y: 0.391000
lr 1.008000e-06 reg 6.800000e+03 train accuracy: 0.353061 val accurac
y: 0.357000
lr 1.008000e-06 reg 1.160000e+04 train accuracy: 0.312163 val accurac
y: 0.330000
lr 1.008000e-06 reg 1.640000e+04 train accuracy: 0.304776 val accurac
y: 0.299000
lr 1.008000e-06 reg 2.120000e+04 train accuracy: 0.277265 val accurac
y: 0.284000
lr 1.008000e-06 reg 2.600000e+04 train accuracy: 0.252163 val accurac
y: 0.259000
lr 1.008000e-06 reg 3.080000e+04 train accuracy: 0.265633 val accurac
y: 0.283000
lr 1.008000e-06 reg 3.560000e+04 train accuracy: 0.285837 val accurac
y: 0.287000
lr 1.008000e-06 reg 4.040000e+04 train accuracy: 0.292122 val accurac
```

```
y: 0.313000
lr 1.008000e-06 reg 4.520000e+04 train accuracy: 0.261306 val accurac
y: 0.260000
lr 1.507000e-06 reg 2.000000e+03 train accuracy: 0.308102 val accurac
y: 0.300000
lr 1.507000e-06 reg 6.800000e+03 train accuracy: 0.305184 val accurac
y: 0.310000
lr 1.507000e-06 reg 1.160000e+04 train accuracy: 0.252102 val accurac
y: 0.257000
lr 1.507000e-06 reg 1.640000e+04 train accuracy: 0.267469 val accurac
y: 0.297000
lr 1.507000e-06 reg 2.120000e+04 train accuracy: 0.240633 val accurac
y: 0.236000
lr 1.507000e-06 reg 2.600000e+04 train accuracy: 0.261510 val accurac
y: 0.270000
lr 1.507000e-06 reg 3.080000e+04 train accuracy: 0.183633 val accurac
y: 0.186000
lr 1.507000e-06 reg 3.560000e+04 train accuracy: 0.273449 val accurac
y: 0.267000
lr 1.507000e-06 reg 4.040000e+04 train accuracy: 0.232918 val accurac
y: 0.223000
lr 1.507000e-06 reg 4.520000e+04 train accuracy: 0.229755 val accurac
y: 0.234000
lr 2.006000e-06 reg 2.000000e+03 train accuracy: 0.303245 val accurac
y: 0.302000
lr 2.006000e-06 reg 6.800000e+03 train accuracy: 0.293755 val accurac
y: 0.302000
lr 2.006000e-06 reg 1.160000e+04 train accuracy: 0.255776 val accurac
y: 0.238000
lr 2.006000e-06 reg 1.640000e+04 train accuracy: 0.263959 val accurac
y: 0.269000
lr 2.006000e-06 reg 2.120000e+04 train accuracy: 0.244286 val accurac
y: 0.255000
lr 2.006000e-06 reg 2.600000e+04 train accuracy: 0.245204 val accurac
y: 0.253000
lr 2.006000e-06 reg 3.080000e+04 train accuracy: 0.263735 val accurac
y: 0.263000
lr 2.006000e-06 reg 3.560000e+04 train accuracy: 0.222755 val accurac
y: 0.251000
lr 2.006000e-06 reg 4.040000e+04 train accuracy: 0.242347 val accurac
y: 0.259000
lr 2.006000e-06 reg 4.520000e+04 train accuracy: 0.247551 val accurac
y: 0.248000
lr 2.505000e-06 reg 2.000000e+03 train accuracy: 0.302408 val accurac
y: 0.304000
lr 2.505000e-06 reg 6.800000e+03 train accuracy: 0.235204 val accurac
y: 0.239000
lr 2.505000e-06 reg 1.160000e+04 train accuracy: 0.293918 val accurac
y: 0.310000
lr 2.505000e-06 reg 1.640000e+04 train accuracy: 0.222327 val accurac
y: 0.208000
lr 2.505000e-06 reg 2.120000e+04 train accuracy: 0.201347 val accurac
y: 0.209000
lr 2.505000e-06 reg 2.600000e+04 train accuracy: 0.269878 val accurac
y: 0.270000
lr 2.505000e-06 reg 3.080000e+04 train accuracy: 0.232673 val accurac
y: 0.234000
```

```
lr 2.505000e-06 reg 3.560000e+04 train accuracy: 0.193245 val accurac
y: 0.186000
lr 2.505000e-06 reg 4.040000e+04 train accuracy: 0.239265 val accurac
y: 0.249000
lr 2.505000e-06 reg 4.520000e+04 train accuracy: 0.174816 val accurac
y: 0.171000
lr 3.004000e-06 reg 2.000000e+03 train accuracy: 0.288163 val accurac
y: 0.286000
lr 3.004000e-06 reg 6.800000e+03 train accuracy: 0.246122 val accurac
y: 0.257000
lr 3.004000e-06 reg 1.160000e+04 train accuracy: 0.224510 val accurac
y: 0.237000
lr 3.004000e-06 reg 1.640000e+04 train accuracy: 0.230449 val accurac
y: 0.238000
lr 3.004000e-06 reg 2.120000e+04 train accuracy: 0.251122 val accurac
y: 0.241000
lr 3.004000e-06 reg 2.600000e+04 train accuracy: 0.253878 val accurac
y: 0.246000
lr 3.004000e-06 reg 3.080000e+04 train accuracy: 0.195265 val accurac
y: 0.221000
lr 3.004000e-06 reg 3.560000e+04 train accuracy: 0.193633 val accurac
y: 0.189000
lr 3.004000e-06 reg 4.040000e+04 train accuracy: 0.210163 val accurac
y: 0.218000
lr 3.004000e-06 reg 4.520000e+04 train accuracy: 0.199816 val accurac
y: 0.201000
lr 3.503000e-06 reg 2.000000e+03 train accuracy: 0.240204 val accurac
y: 0.248000
lr 3.503000e-06 reg 6.800000e+03 train accuracy: 0.218694 val accurac
y: 0.217000
lr 3.503000e-06 reg 1.160000e+04 train accuracy: 0.242878 val accurac
y: 0.257000
lr 3.503000e-06 reg 1.640000e+04 train accuracy: 0.202816 val accurac
y: 0.221000
lr 3.503000e-06 reg 2.120000e+04 train accuracy: 0.218429 val accurac
y: 0.228000
lr 3.503000e-06 reg 2.600000e+04 train accuracy: 0.205531 val accurac
y: 0.206000
lr 3.503000e-06 reg 3.080000e+04 train accuracy: 0.213694 val accurac
y: 0.223000
lr 3.503000e-06 reg 3.560000e+04 train accuracy: 0.211980 val accurac
y: 0.213000
lr 3.503000e-06 reg 4.040000e+04 train accuracy: 0.184449 val accurac
y: 0.192000
lr 3.503000e-06 reg 4.520000e+04 train accuracy: 0.210510 val accurac
y: 0.200000
lr 4.002000e-06 reg 2.000000e+03 train accuracy: 0.255531 val accurac
y: 0.280000
lr 4.002000e-06 reg 6.800000e+03 train accuracy: 0.231959 val accurac
y: 0.227000
lr 4.002000e-06 reg 1.160000e+04 train accuracy: 0.206878 val accurac
y: 0.193000
lr 4.002000e-06 reg 1.640000e+04 train accuracy: 0.175184 val accurac
y: 0.190000
lr 4.002000e-06 reg 2.120000e+04 train accuracy: 0.210082 val accurac
y: 0.211000
lr 4.002000e-06 reg 2.600000e+04 train accuracy: 0.164531 val accurac
```

```
y: 0.157000
lr 4.002000e-06 reg 3.080000e+04 train accuracy: 0.184265 val accurac
y: 0.172000
lr 4.002000e-06 reg 3.560000e+04 train accuracy: 0.196918 val accurac
y: 0.198000
lr 4.002000e-06 reg 4.040000e+04 train accuracy: 0.178347 val accurac
y: 0.188000
lr 4.002000e-06 reg 4.520000e+04 train accuracy: 0.188816 val accurac
y: 0.189000
lr 4.501000e-06 reg 2.000000e+03 train accuracy: 0.255449 val accurac
y: 0.243000
lr 4.501000e-06 reg 6.800000e+03 train accuracy: 0.279224 val accurac
y: 0.283000
lr 4.501000e-06 reg 1.160000e+04 train accuracy: 0.223102 val accurac
y: 0.260000
lr 4.501000e-06 reg 1.640000e+04 train accuracy: 0.179531 val accurac
y: 0.187000
lr 4.501000e-06 reg 2.120000e+04 train accuracy: 0.153020 val accurac
y: 0.157000
lr 4.501000e-06 reg 2.600000e+04 train accuracy: 0.191367 val accurac
y: 0.184000
lr 4.501000e-06 reg 3.080000e+04 train accuracy: 0.172020 val accurac
y: 0.182000
lr 4.501000e-06 reg 3.560000e+04 train accuracy: 0.209143 val accurac
y: 0.196000
lr 4.501000e-06 reg 4.040000e+04 train accuracy: 0.191327 val accurac
y: 0.193000
lr 4.501000e-06 reg 4.520000e+04 train accuracy: 0.157776 val accurac
y: 0.168000
best validation accuracy achieved during cross-validation: 0.391000
```

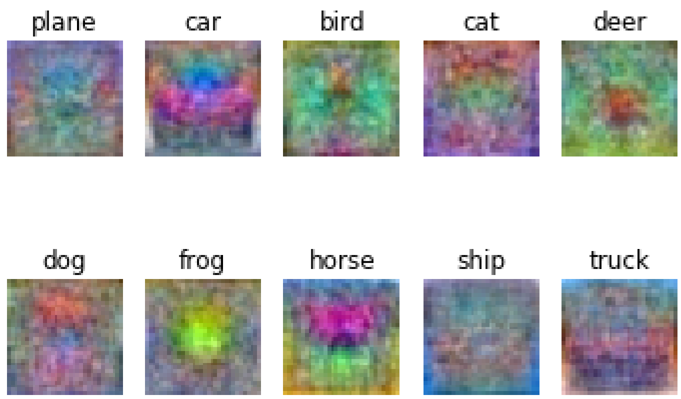
In [20]:
```
# Visualize the cross-validation results
import math
x_scatter = [math.log10(x[0]) for x in results]
y_scatter = [math.log10(x[1]) for x in results]

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

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



In [21]:
```
# Evaluate the best svm on test set
y_test_pred = best_svm.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('linear SVM on raw pixels final test set accuracy: %f' % test_a
ccuracy)
```

```
linear SVM on raw pixels final test set accuracy: 0.368000
```

In [22]:
```
# Visualize the learned weights for each class.
# Depending on your choice of learning rate and regularization streng
th, 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', 'hor
se', 'ship', 'truck']
for i in range(10):
    plt.subplot(2, 5, i + 1)

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



Inline question 2:

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

Your answer: *The visualized SVM weights look like the average of all the images in that category. This is because we are effectively doing a cos function of two vectors, one being the visualized SVM and the other being test. To get best possible result for cos, they need to be as close to each other as possible, which means the visualized SVM should give maximum result when we do a cos function with that category images. An average of the images in this category fits this category reasonably well.*