Multiclass Support Vector Machine exercise

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 [261]:
```
# Run some setup code for this notebook.
from __future__ import print_function

import random
import numpy as np
from cs175.data_utils import load_CIFAR10
import matplotlib.pyplot as plt



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

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

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

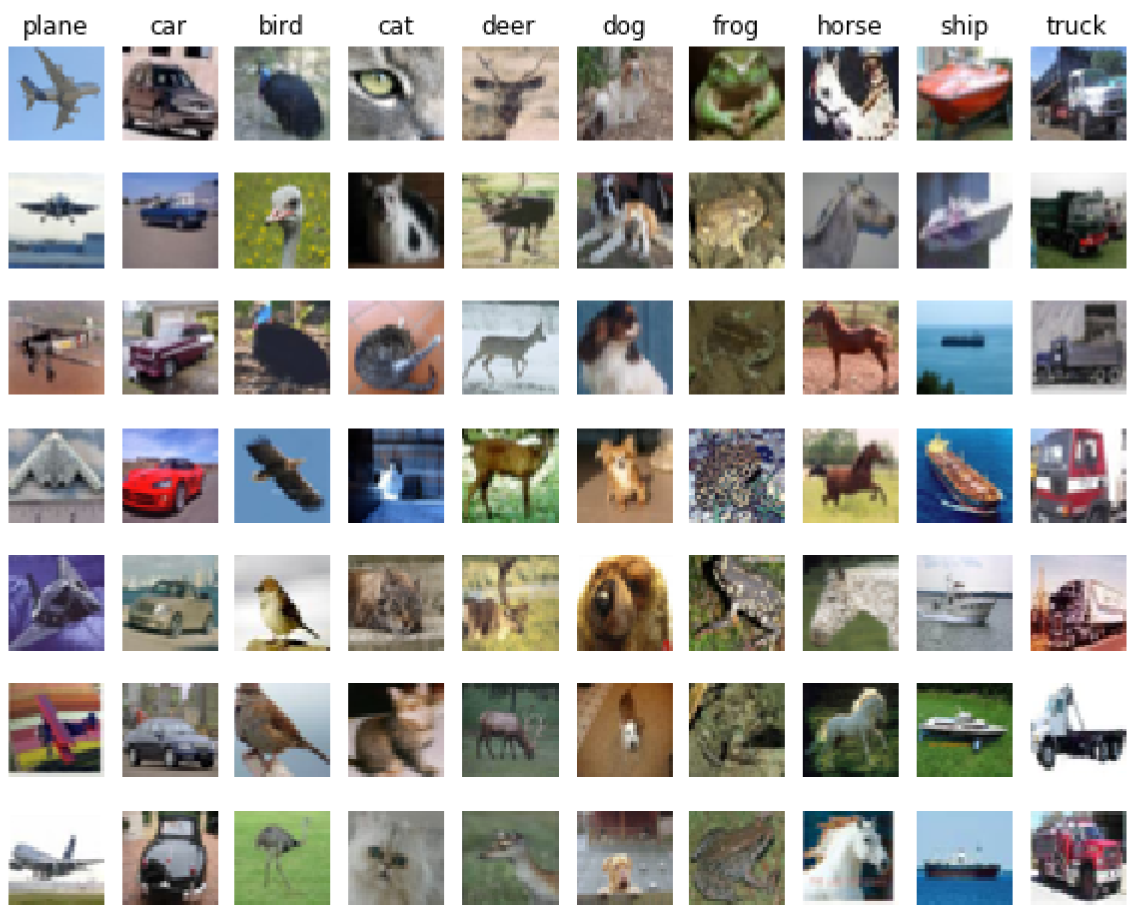
CIFAR-10 Data Loading and Preprocessing

In [262]:
```
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs175/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:  (50000L, 32L, 32L, 3L)
Training labels shape:  (50000L,)
Test data shape:  (10000L, 32L, 32L, 3L)
Test labels shape:  (10000L,)
```

In [263]:

```
# 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', 'shi
p', '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 [264]:
```
# 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)
```

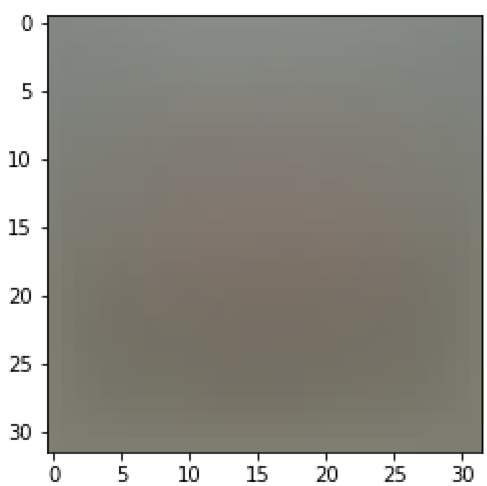
```
Train data shape:  (49000L, 32L, 32L, 3L)
Train labels shape:  (49000L,)
Validation data shape:  (1000L, 32L, 32L, 3L)
Validation labels shape:  (1000L,)
Test data shape:  (1000L, 32L, 32L, 3L)
Test labels shape:  (1000L,)
```

In [265]:
```
# 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:  (49000L, 3072L)
Validation data shape:  (1000L, 3072L)
Test data shape:  (1000L, 3072L)
dev data shape:  (500L, 3072L)
```

In [266]:
```
# 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 [267]:
```
# 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 [268]:
```
# 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)
```

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

SVM Classifier

Your code for this section will all be written inside **cs175/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 [269]:
```
# Evaluate the naive implementation of the loss we provided for you:
from cs175.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: 9.161092
```

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 [270]:
```
# Once you've implemented the gradient, recompute it with the code below
# and gradient check it with the function we provided for you

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

# Numerically compute the gradient along several randomly chosen dimensions, a
nd
# compare them with your analytically computed gradient. The numbers should ma
tch
# almost exactly along all dimensions.
from cs175.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: -0.281591 analytic: -0.281591, relative error: 6.088658e-10
numerical: 24.940809 analytic: 24.940809, relative error: 1.035980e-11
numerical: 47.877360 analytic: 47.877360, relative error: 8.068399e-12
numerical: 14.433459 analytic: 14.433459, relative error: 9.998566e-12
numerical: 6.211376 analytic: 6.211376, relative error: 6.326626e-11
numerical: 2.082350 analytic: 2.082350, relative error: 4.035531e-11
numerical: 2.389955 analytic: 2.389955, relative error: 7.416661e-11
numerical: -12.202079 analytic: -12.202079, relative error: 2.928553e-11
numerical: 8.999121 analytic: 8.999121, relative error: 9.961400e-12
numerical: 2.194905 analytic: 2.194905, relative error: 7.511593e-11
numerical: 8.176707 analytic: 8.184255, relative error: 4.613372e-04
numerical: -11.834654 analytic: -11.831532, relative error: 1.319047e-04
numerical: 2.350680 analytic: 2.343031, relative error: 1.629647e-03
numerical: -31.310090 analytic: -31.306958, relative error: 5.002476e-05
numerical: 10.426179 analytic: 10.416734, relative error: 4.531177e-04
numerical: -23.172681 analytic: -23.173354, relative error: 1.450881e-05
numerical: -20.853513 analytic: -20.863294, relative error: 2.344672e-04
numerical: 10.821806 analytic: 10.829965, relative error: 3.768187e-04
numerical: 3.389422 analytic: 3.389383, relative error: 5.765379e-06
numerical: -4.933609 analytic: -4.929806, relative error: 3.855615e-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? *Hint: the SVM loss function is not strictly speaking differentiable*

Your Answer: Hinge loss is not a differentiable function. This could cause issues when the gradient is not differentiable at the target point, for example just before the hinge. The chances of occurring directly at one of these points is very low.

In [271]:
```
# Next implement the function svm_loss_vectorized; for now only compute the lo
ss;
# we will implement the gradient in a moment.
tic = time.time()
loss_naive, grad_naive = svm_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Naive loss: %e computed in %fs' % (loss_naive, toc - tic))

from cs175.classifiers.linear_svm import svm_loss_vectorized
tic = time.time()
loss_vectorized, _ = svm_loss_vectorized(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Vectorized loss: %e computed in %fs' % (loss_vectorized, toc - tic))

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

```
Naive loss: 9.161092e+00 computed in 0.059000s
Vectorized loss: 1.685215e+00 computed in 0.004000s
difference: 7.475877
```

In [272]:
```
# 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 0.059000s
Vectorized loss and gradient: computed in 0.005000s
difference: 2270.068900
```

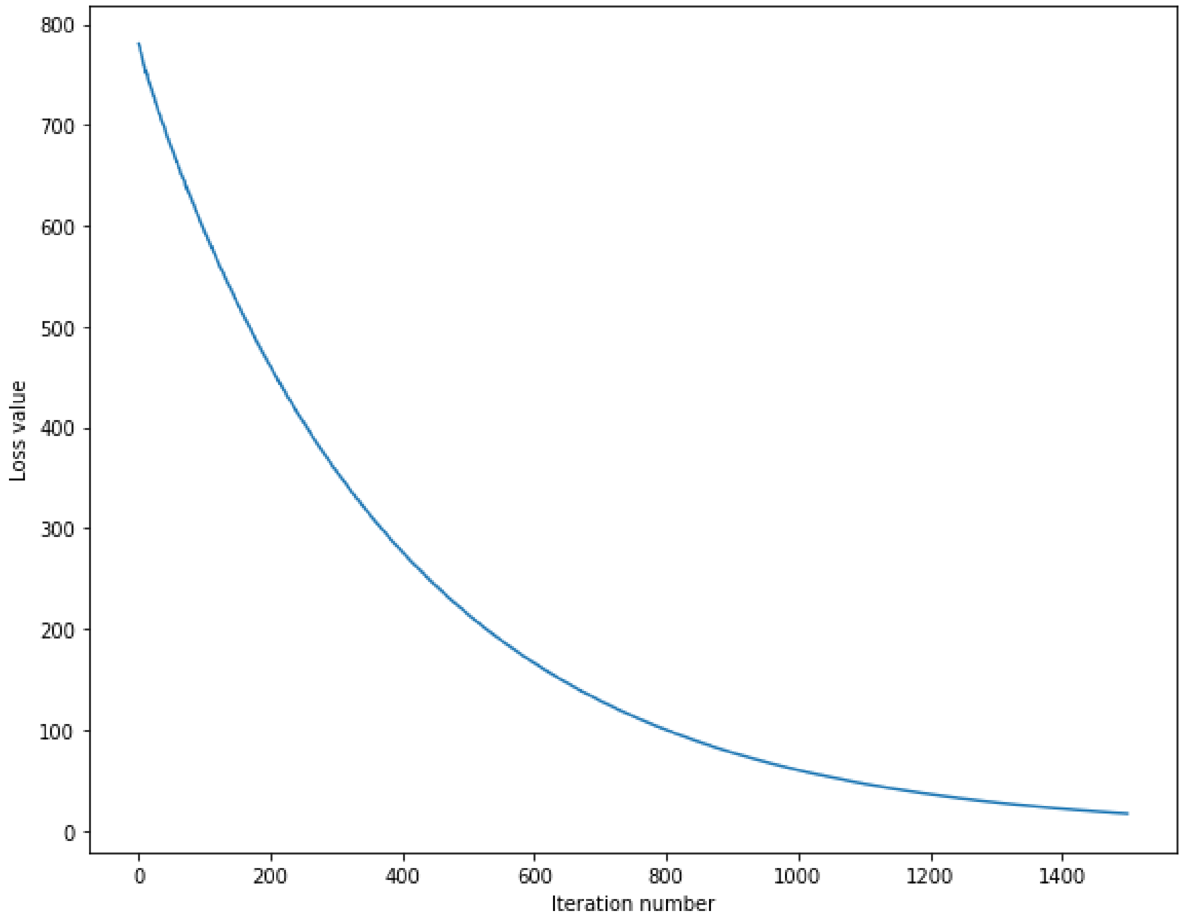
Stochastic Gradient Descent

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 [273]:
```
# In the file linear_classifier.py, implement SGD in the function
# LinearClassifier.train() and then run it with the code below.
from cs175.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 780.481247
iteration 100 / 1500: loss 593.456489
iteration 200 / 1500: loss 460.257223
iteration 300 / 1500: loss 357.276924
iteration 400 / 1500: loss 276.031261
iteration 500 / 1500: loss 214.224698
iteration 600 / 1500: loss 166.005736
iteration 700 / 1500: loss 128.806313
iteration 800 / 1500: loss 99.892242
iteration 900 / 1500: loss 77.483965
iteration 1000 / 1500: loss 60.056017
iteration 1100 / 1500: loss 46.565064
iteration 1200 / 1500: loss 36.202661
iteration 1300 / 1500: loss 28.003894
iteration 1400 / 1500: loss 21.766643
That took 6.762000s
```

In [274]:
```
# 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 [275]:
```
# Write the LinearSVM.predict function and evaluate the performance on both th
e
# training and validation set
y_train_pred = svm.predict(X_train)
print('training accuracy: %f' % (np.mean(y_train == y_train_pred), ))
y_val_pred = svm.predict(X_val)
print('validation accuracy: %f' % (np.mean(y_val == y_val_pred), ))
```

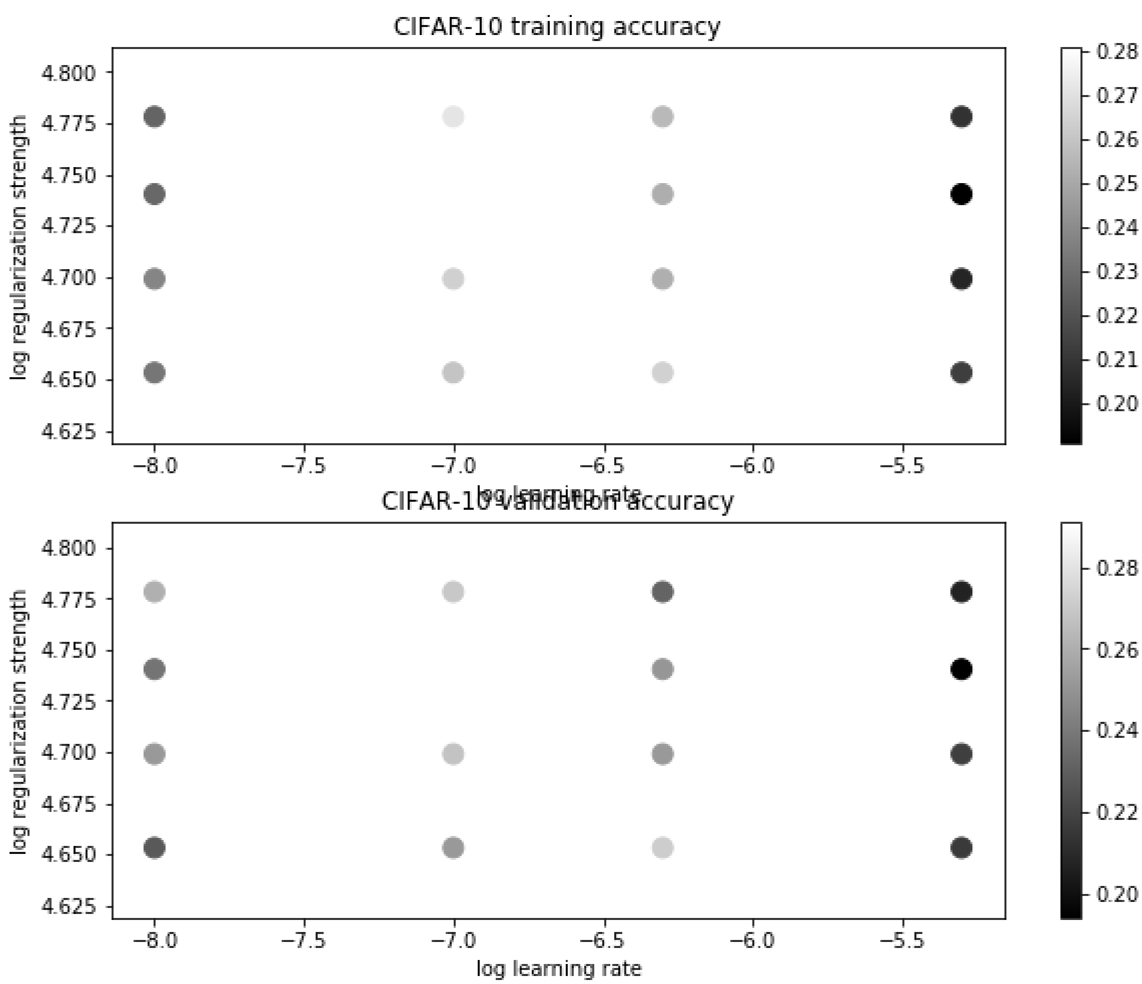
```
training accuracy: 0.251510
validation accuracy: 0.231000
```

In [284]:
```
# 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-6, 1e-7, 5e-7, 1e-8]
regularization_strengths = [4.5e4, 5e4, 5.5e4, 6e4]

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

################################################################################
##
# TODO:
 #
# Write code that chooses the best hyperparameters by tuning on the validation
 #
# set. For each combination of hyperparameters, train a linear SVM on the
 #
# training set, compute its accuracy on the training and validation sets, and
 #
# store these numbers in the results dictionary. In addition, store the best
 #
# validation accuracy in best_val and the LinearSVM object that achieves this
 #
# accuracy in best_svm.
 #
#
 #
# Hint: You should use a small value for num_iters as you develop your
 #
# validation code so that the SVMs don't take much time to train; once you are
 #
# confident that your validation code works, you should rerun the validation
 #
# code with a larger value for num_iters.
 #
################################################################################
##
for learn in learning_rates:
    for reg in regularization_strengths:
        svm = LinearSVM()
        svm.train(X_train, y_train, learning_rate=learn, reg=reg,
                      num_iters=3000)
        p_train = svm.predict(X_train)
        p_val   = svm.predict(X_val)

        train_acc = np.mean(p_train == y_train)
        val_acc   = np.mean(p_val == y_val)
```

```
        if val_acc > best_val:
            best_val = val_acc
            best_svm = svm

        results[(learn, 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_v
al)
```

```
lr 1.000000e-08 reg 4.500000e+04 train accuracy: 0.232592 val accuracy: 0.228
000
lr 1.000000e-08 reg 5.000000e+04 train accuracy: 0.237898 val accuracy: 0.252
000
lr 1.000000e-08 reg 5.500000e+04 train accuracy: 0.227878 val accuracy: 0.238
000
lr 1.000000e-08 reg 6.000000e+04 train accuracy: 0.226429 val accuracy: 0.261
000
lr 1.000000e-07 reg 4.500000e+04 train accuracy: 0.259939 val accuracy: 0.252
000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.264429 val accuracy: 0.268
000
lr 1.000000e-07 reg 5.500000e+04 train accuracy: 0.280959 val accuracy: 0.291
000
lr 1.000000e-07 reg 6.000000e+04 train accuracy: 0.271714 val accuracy: 0.270
000
lr 5.000000e-07 reg 4.500000e+04 train accuracy: 0.264612 val accuracy: 0.272
000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.252653 val accuracy: 0.252
000
lr 5.000000e-07 reg 5.500000e+04 train accuracy: 0.252327 val accuracy: 0.251
000
lr 5.000000e-07 reg 6.000000e+04 train accuracy: 0.256265 val accuracy: 0.232
000
lr 5.000000e-06 reg 4.500000e+04 train accuracy: 0.213122 val accuracy: 0.216
000
lr 5.000000e-06 reg 5.000000e+04 train accuracy: 0.204286 val accuracy: 0.218
000
lr 5.000000e-06 reg 5.500000e+04 train accuracy: 0.191082 val accuracy: 0.194
000
lr 5.000000e-06 reg 6.000000e+04 train accuracy: 0.208816 val accuracy: 0.207
000
best validation accuracy achieved during cross-validation: 0.291000
```

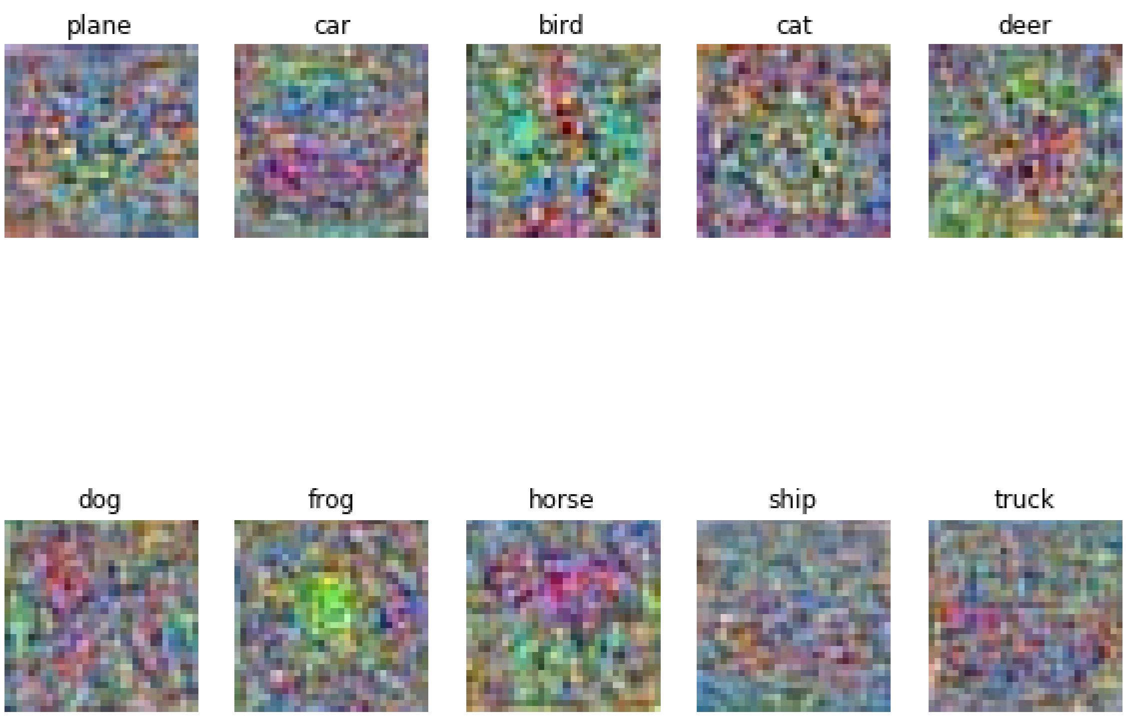
In [287]:
```
# Visualize the cross-validation results
import math
x_scatter = [math.log10(x[0]) for x in results]
y_scatter = [math.log10(x[1]) for x in results]

# 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 [288]:
```
# 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: 0.273000
```

In [290]:
```
# Visualize the learned weights for each class.
# Depending on your choice of learning rate and regularization strength, these
 may
# or may not be nice to look at.
w = best_svm.W[:-1,:] # strip out the bias
w = w.reshape(32, 32, 3, 10)
w_min, w_max = np.min(w), np.max(w)
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'shi
p', 'truck']
for i in range(10):
    plt.subplot(2, 5, i + 1)

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



Inline question 2:

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

Your answer: These images appear to be many versions of themselves layered over. The SVM weights represent these bright and dark colors as they are the maximizing the margins between pixel intensity and choosing the largest margins. The pictures with most consistent background colors do show dominance in the colors overlayed, such as blue in ships and green in frog.