svm

September 28, 2018

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 [10]:
```
# Run some setup code for this notebook.

import random
import numpy as np
from cs682.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
```

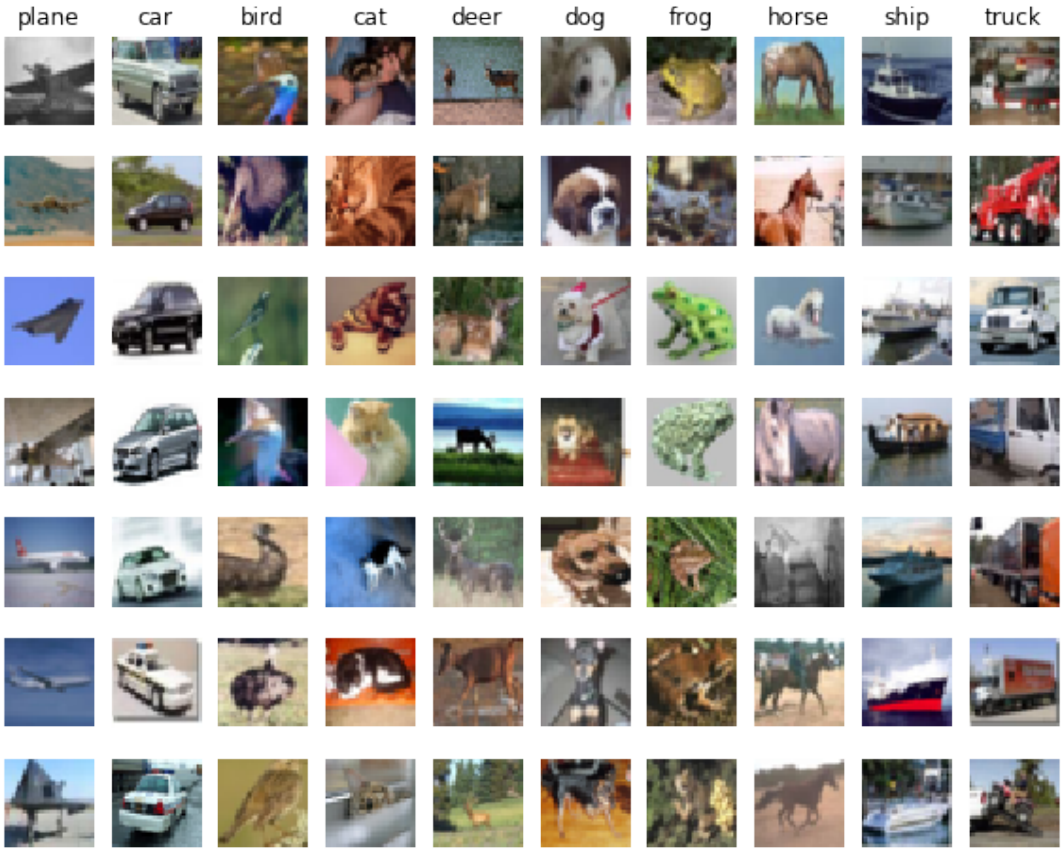
```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

1.1 CIFAR-10 Data Loading and Preprocessing

In [11]:
```
# 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 memor
try:
   del X_train, y_train
   del X_test, y_test
   print('Clear previously loaded data.')
except:
   pass

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

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

```
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 [12]:
```
# 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', 'tru
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 [13]:
```
# 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,)
```

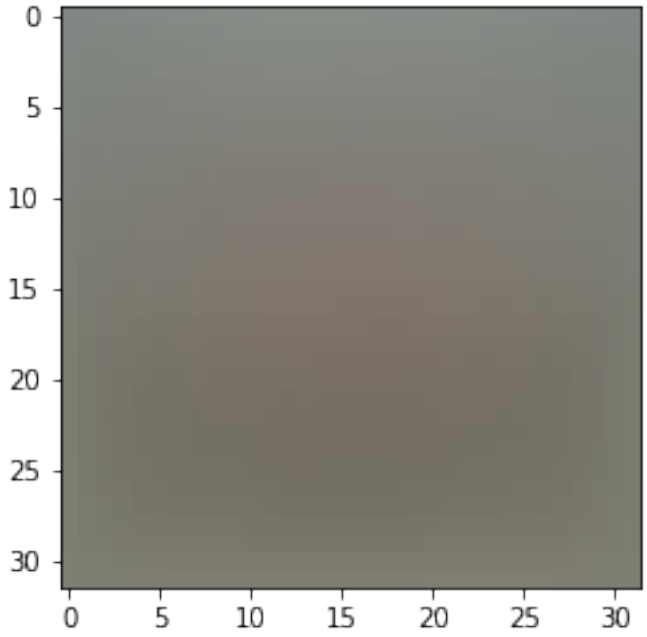
```
In [14]: # Preprocessing: reshape the image data into rows
         X_train = np.reshape(X_train, (X_train.shape[0], -1))
         X_val = np.reshape(X_val, (X_val.shape[0], -1))
         X_test = np.reshape(X_test, (X_test.shape[0], -1))
         X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

         # 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 [15]: # 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 [16]:
```
# 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 [17]:
```
# 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)
```

1.2 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 [18]: # 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: 8.990574
```

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 [19]: # 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 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: -24.108611 analytic: -24.108611, relative error: 2.704705e-11
numerical: 7.813560 analytic: 7.813560, relative error: 9.103830e-12
numerical: -12.520158 analytic: -12.520158, relative error: 8.184262e-12
numerical: -2.031423 analytic: -2.031423, relative error: 1.282149e-10
```

```
numerical: 4.496740 analytic: 4.496740, relative error: 5.679604e-11
numerical: -24.770518 analytic: -24.770518, relative error: 1.904929e-11
numerical: -3.636857 analytic: -3.636857, relative error: 3.462981e-11
numerical: 4.101868 analytic: 4.101868, relative error: 2.708284e-11
numerical: 6.854217 analytic: 6.854217, relative error: 1.483525e-11
numerical: 8.042819 analytic: 8.042819, relative error: 1.071404e-12
numerical: -15.222171 analytic: -15.222443, relative error: 8.958126e-06
numerical: 14.872230 analytic: 14.876024, relative error: 1.275436e-04
numerical: -0.993121 analytic: -0.990314, relative error: 1.415678e-03
numerical: 0.150445 analytic: 0.153391, relative error: 9.696562e-03
numerical: -2.875891 analytic: -2.876576, relative error: 1.191767e-04
numerical: -3.042409 analytic: -3.037801, relative error: 7.578284e-04
numerical: 18.864060 analytic: 18.870379, relative error: 1.674558e-04
numerical: 7.001854 analytic: 7.008647, relative error: 4.848467e-04
numerical: 2.096751 analytic: 2.098283, relative error: 3.652597e-04
numerical: 13.471014 analytic: 13.468233, relative error: 1.032337e-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? How would change the margin affect of the frequency of this happening? *Hint: the SVM loss function is not strictly speaking differentiable*

Your Answer: Yeah, it is possible. The discrepancy is can happen because the level of accuracy is not small enough. This can happen with with all non differenciable functions. Example, for the mod function, which is not differenciable at the origin, hence the grad check might not match for positive or negative levels of accuracies. It will not have a huge effect on the margin.

In [32]:
```
# 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 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 be much faster.
print('difference: %f' % (loss_naive - loss_vectorized))
```

```
Naive loss: 8.990574e+00 computed in 0.124924s
Vectorized loss: 8.990574e+00 computed in 0.003596s
difference: -0.000000
```

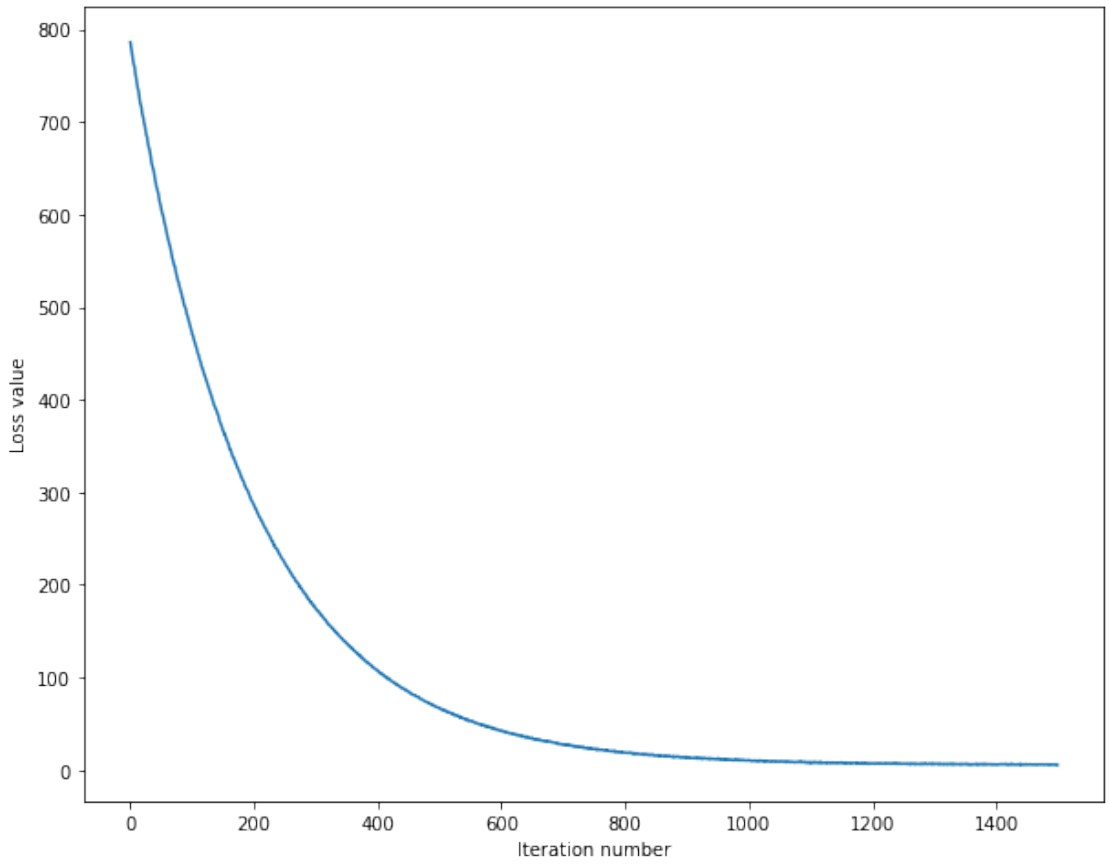
1.2.2 Stochastic Gradient Descent

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 [21]:
```
# 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=1500, verbose=True)
toc = time.time()
print('That took %fs' % (toc - tic))
```

```
iteration 0 / 1500: loss 786.205506
iteration 100 / 1500: loss 470.545603
iteration 200 / 1500: loss 285.992351
iteration 300 / 1500: loss 174.378581
iteration 400 / 1500: loss 107.180093
iteration 500 / 1500: loss 66.982101
iteration 600 / 1500: loss 42.480333
iteration 700 / 1500: loss 27.227735
iteration 800 / 1500: loss 18.838914
iteration 900 / 1500: loss 13.679232
iteration 1000 / 1500: loss 10.638101
iteration 1100 / 1500: loss 7.845622
iteration 1200 / 1500: loss 6.987930
iteration 1300 / 1500: loss 6.277957
iteration 1400 / 1500: loss 5.582518
That took 12.756051s
```

In [22]:
```
# 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 [14]:
```
# 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: 0.378714
validation accuracy: 0.385000
```

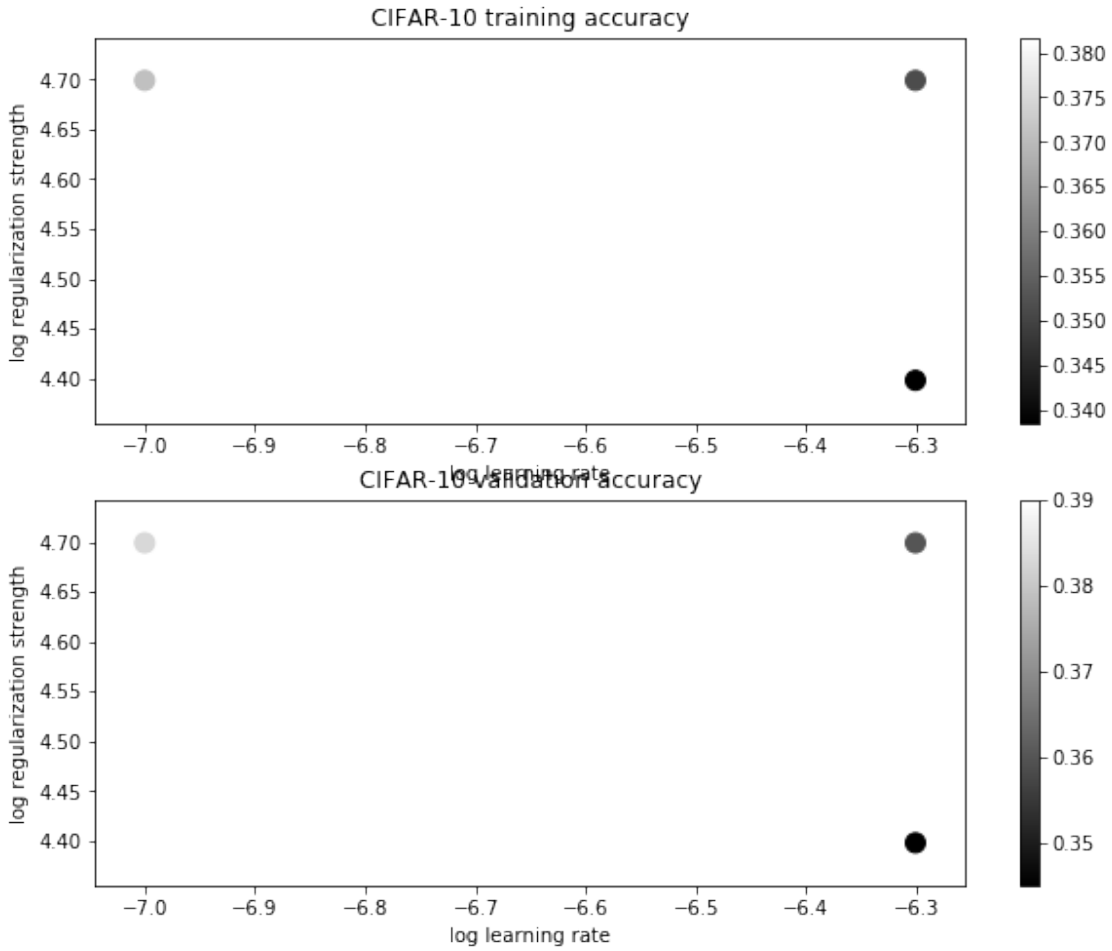
In [28]:
```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of about 0.4 on the validation set.
learning_rates = [1e-7, 5e-7]
regularization_strengths = [2.5e4, 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 the fraction
# of data points that are correctly classified.
test_accuracies =[]
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.
svm1 = LinearSVM()
for i in learning_rates:
    for j in regularization_strengths:
        loss_history = svm1.train(X_train, y_train, learning_rate=i, reg=j, num_iters=
        y_train_prediction = svm1.predict(X_train)
        accuracy_train = np.mean(y_train == y_train_prediction)
        y_val_prediction = svm1.predict(X_val)
        accuracy_validation = np.mean(y_val == y_val_prediction)
        y_test_pred = svm1.predict(X_test)
        #test_accuracy = np.mean(y_test == y_test_pred)
        #test_accuracies.append(test_accuracy)
        #print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
        results[(i,j)] = accuracy_train, accuracy_validation
        if accuracy_validation > best_val:
           best_val = accuracy_validation
           best_svm = svm1
################################################################################
# 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
################################################################################
#                              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 789.757246
iteration 100 / 1500: loss 469.809985
iteration 200 / 1500: loss 284.606484
iteration 300 / 1500: loss 173.803452
iteration 400 / 1500: loss 106.274001
iteration 500 / 1500: loss 65.857728
iteration 600 / 1500: loss 42.266374
iteration 700 / 1500: loss 27.464876
iteration 800 / 1500: loss 18.968869
iteration 900 / 1500: loss 12.858708
iteration 1000 / 1500: loss 10.190389
iteration 1100 / 1500: loss 8.758876
iteration 1200 / 1500: loss 7.240618
iteration 1300 / 1500: loss 6.274803
iteration 1400 / 1500: loss 6.261473
iteration 0 / 1500: loss 7.158331
iteration 100 / 1500: loss 6.205398
iteration 200 / 1500: loss 6.264308
iteration 300 / 1500: loss 5.280626
iteration 400 / 1500: loss 6.188082
iteration 500 / 1500: loss 5.715437
iteration 600 / 1500: loss 6.238145
iteration 700 / 1500: loss 5.628909
iteration 800 / 1500: loss 5.712633
iteration 900 / 1500: loss 5.561339
iteration 1000 / 1500: loss 5.819825
iteration 1100 / 1500: loss 5.666501
iteration 1200 / 1500: loss 5.946238
iteration 1300 / 1500: loss 6.116036
iteration 1400 / 1500: loss 5.623897
iteration 0 / 1500: loss 5.564203
iteration 100 / 1500: loss 6.003337
iteration 200 / 1500: loss 5.799800
iteration 300 / 1500: loss 6.032289
iteration 400 / 1500: loss 6.281438
iteration 500 / 1500: loss 5.563329
iteration 600 / 1500: loss 5.651315
iteration 700 / 1500: loss 6.033787
iteration 800 / 1500: loss 5.599360
iteration 900 / 1500: loss 5.710713
iteration 1000 / 1500: loss 5.737837
iteration 1100 / 1500: loss 6.367185
iteration 1200 / 1500: loss 6.165881
iteration 1300 / 1500: loss 6.173567
iteration 1400 / 1500: loss 5.797589
iteration 0 / 1500: loss 7.223745
iteration 100 / 1500: loss 6.066232
iteration 200 / 1500: loss 6.169951
```

```
iteration 300 / 1500: loss 6.311852
iteration 400 / 1500: loss 6.533727
iteration 500 / 1500: loss 5.604897
iteration 600 / 1500: loss 6.530148
iteration 700 / 1500: loss 6.613336
iteration 800 / 1500: loss 5.763047
iteration 900 / 1500: loss 6.905548
iteration 1000 / 1500: loss 6.369873
iteration 1100 / 1500: loss 6.338782
iteration 1200 / 1500: loss 6.237365
iteration 1300 / 1500: loss 6.724698
iteration 1400 / 1500: loss 6.159615
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.381673 val accuracy: 0.390000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.370898 val accuracy: 0.383000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.338469 val accuracy: 0.345000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.351327 val accuracy: 0.360000
best validation accuracy achieved during cross-validation: 0.390000
```

In [29]:
```
# 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()
```

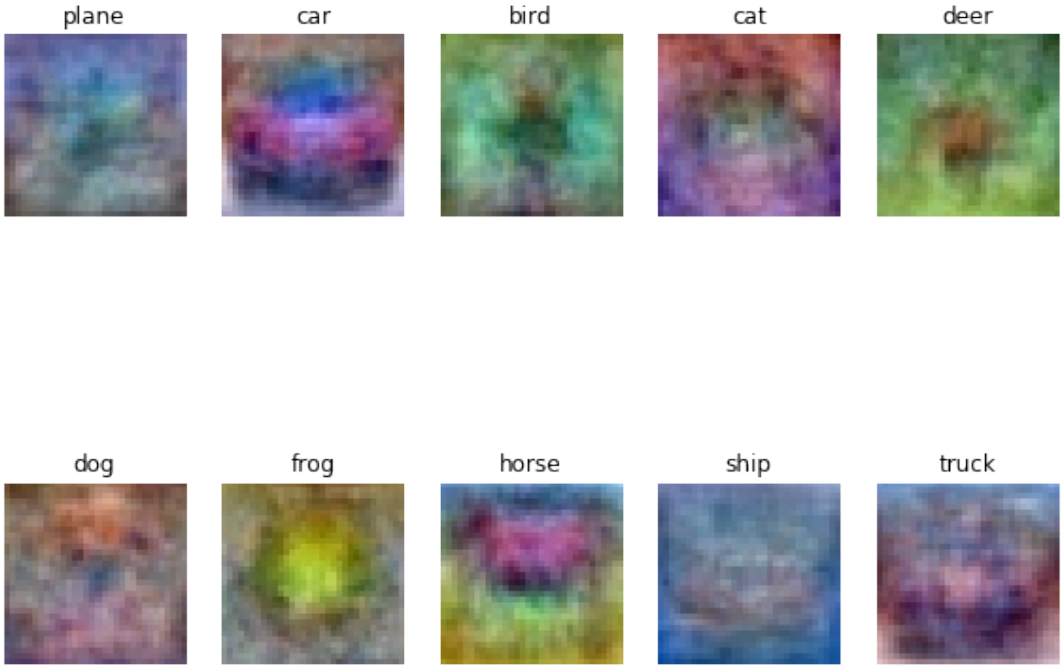


In [30]:
```
# Evaluate the best svm on test set

y_test_pred = best_svm.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
#svm2 = LinearSVM()

print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
```

```
linear SVM on raw pixels final test set accuracy: 0.360000
```

In [31]:
```
# 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', 'tru
```

```
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])
```



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: Theses svm weights are an average of the data looks. The plane has a light blue background because skies are blue and are a common background for planes. The structure of a car can be noticed for the car class. And a two headed horse structure for the horse class.