svm

May 8, 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.

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 [60]:
```
# Run some setup code for this notebook.
# Reference: https://github.com/Halfish/cs231n/tree/master/assignment1

import random
import numpy as np
from hmwk5_2.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
```

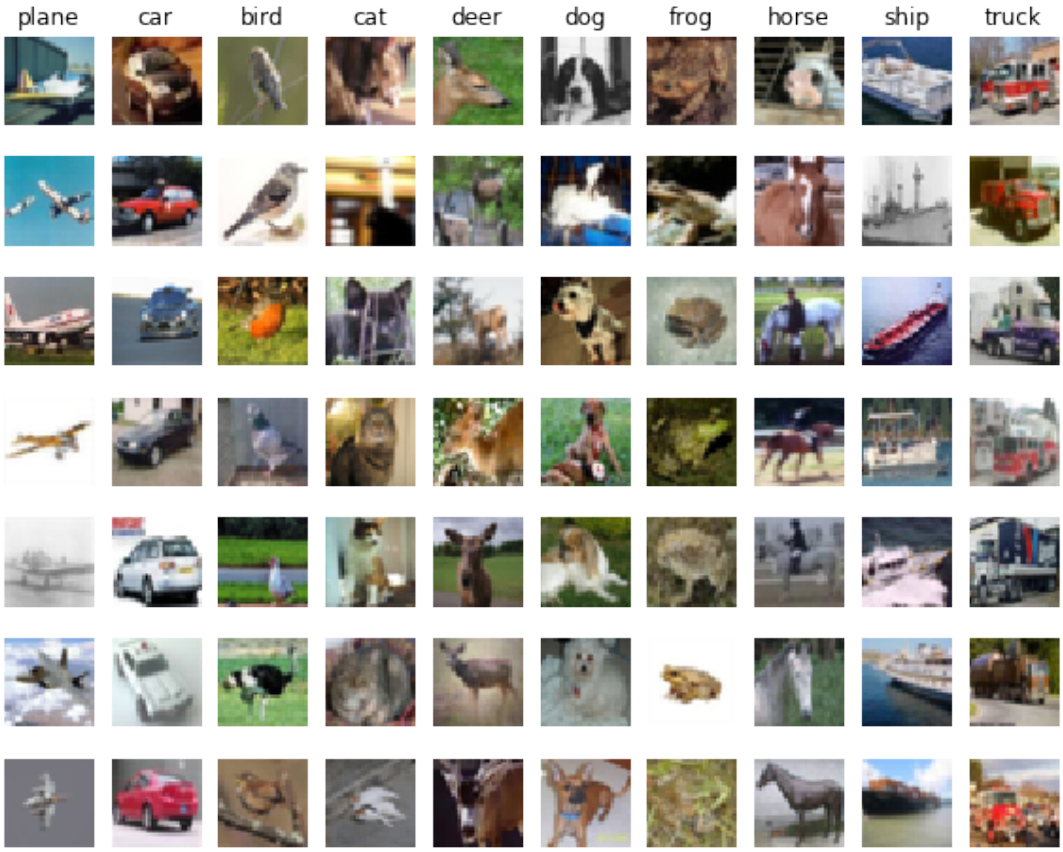
```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

1.1 CIFAR-10 Data Loading and Preprocessing

In [61]:
```
# Load the raw CIFAR-10 data.
cifar10_dir = 'hmwk5_2/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 [62]:
```
# 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 [63]:
```
# 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,)
```

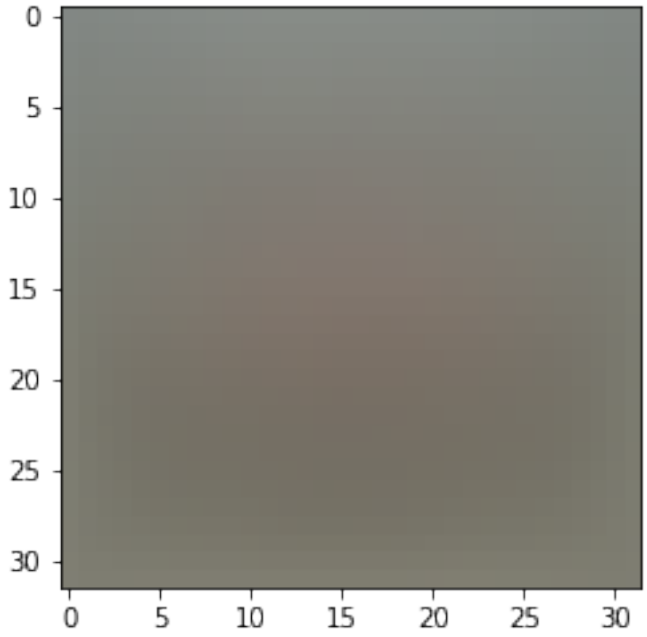
```
In [64]: # Preprocessing: reshape the image data into rows
         X_train = np.reshape(X_train, (X_train.shape[0], -1))
         X_val = np.reshape(X_val, (X_val.shape[0], -1))
         X_test = np.reshape(X_test, (X_test.shape[0], -1))
         X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

         # 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 [65]: # 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 [66]:
```
        # 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 [67]:
```
        # 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 **hmwk5_2/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 [68]: # Evaluate the naive implementation of the loss we provided for you:
         from hmwk5_2.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: 8.483553
```

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 [69]: # 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 hmwk5_2.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: 25.855713 analytic: 25.815622, relative error: 7.758919e-04
numerical: 3.907512 analytic: 3.907512, relative error: 2.427447e-11
numerical: -17.879713 analytic: -17.879713, relative error: 7.587391e-12
numerical: -27.041109 analytic: -27.155068, relative error: 2.102712e-03
```

```
numerical: -39.878786 analytic: -39.878786, relative error: 3.305603e-12
numerical: -35.669623 analytic: -35.669623, relative error: 2.049580e-12
numerical: -18.814247 analytic: -18.913386, relative error: 2.627766e-03
numerical: 19.171200 analytic: 19.171200, relative error: 3.150358e-15
numerical: -2.099352 analytic: -2.099352, relative error: 5.713892e-11
numerical: -34.032650 analytic: -34.039644, relative error: 1.027424e-04
numerical: -0.056620 analytic: -0.056620, relative error: 4.534219e-09
numerical: -3.435647 analytic: -3.347384, relative error: 1.301231e-02
numerical: 5.443648 analytic: 5.443648, relative error: 1.131612e-11
numerical: -39.734890 analytic: -39.754235, relative error: 2.433716e-04
numerical: 8.020023 analytic: 8.020023, relative error: 3.957477e-12
numerical: -8.392084 analytic: -8.418687, relative error: 1.582489e-03
numerical: -11.785246 analytic: -11.695452, relative error: 3.824171e-03
numerical: 27.871778 analytic: 27.871778, relative error: 1.173143e-11
numerical: -8.505469 analytic: -8.505469, relative error: 1.776739e-11
numerical: 19.194893 analytic: 19.194893, relative error: 8.537110e-12
```

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: The disrepancy is caused by where the cost function is not differentiable. For example, the SVM has the term max(0,x), when x is 0, the gradient is undefined.

In [70]:
```
# Next implement the function svm_loss_vectorized; for now only compute the loss;
# we will implement the gradient in a moment.
import time
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 hmwk5_2.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: 8.483553e+00 computed in 0.126018s
Vectorized loss: 8.483553e+00 computed in 0.005173s
difference: -0.000000
```

In [71]:
```
# 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 0.132670s
Vectorized loss and gradient: computed in 0.004662s
difference: 0.000000
```

1.2.2 Stochastic Gradient Descent

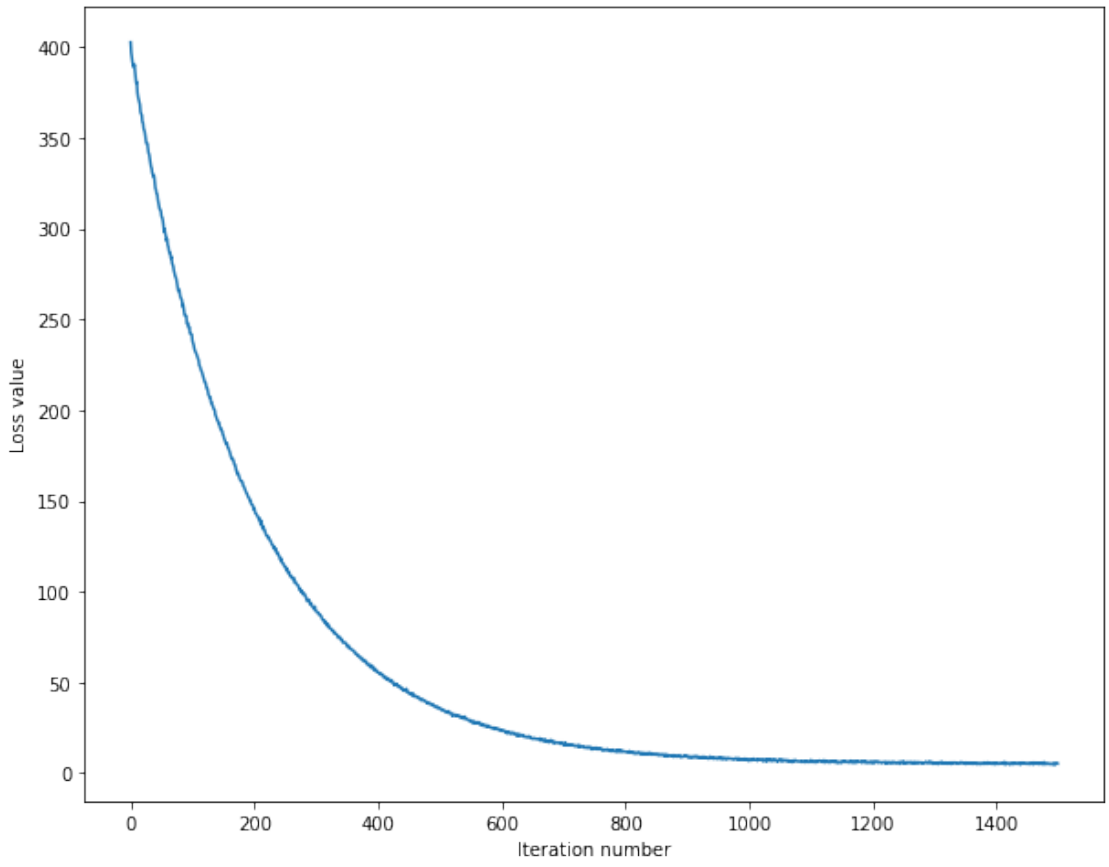
We now have vectorized and efficient expressions for the loss, the gradient and our gradient matches the numerical gradient. We are therefore ready to do SGD to minimize the loss.

In [72]:
```
# In the file linear_classifier.py, implement SGD in the function
# LinearClassifier.train() and then run it with the code below.
from hmwk5_2.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 402.754269
iteration 100 / 1500: loss 238.144304
iteration 200 / 1500: loss 145.899293
iteration 300 / 1500: loss 89.100576
iteration 400 / 1500: loss 55.703209
iteration 500 / 1500: loss 35.720911
iteration 600 / 1500: loss 23.287144
iteration 700 / 1500: loss 15.334827
```

```
iteration 800 / 1500: loss 11.587156
iteration 900 / 1500: loss 9.004518
iteration 1000 / 1500: loss 7.156552
iteration 1100 / 1500: loss 7.182535
iteration 1200 / 1500: loss 5.944234
iteration 1300 / 1500: loss 5.644012
iteration 1400 / 1500: loss 5.260238
That took 4.692531s
```

In [73]:
```
# 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 [74]:
```
# 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: 0.380857
validation accuracy: 0.393000
```

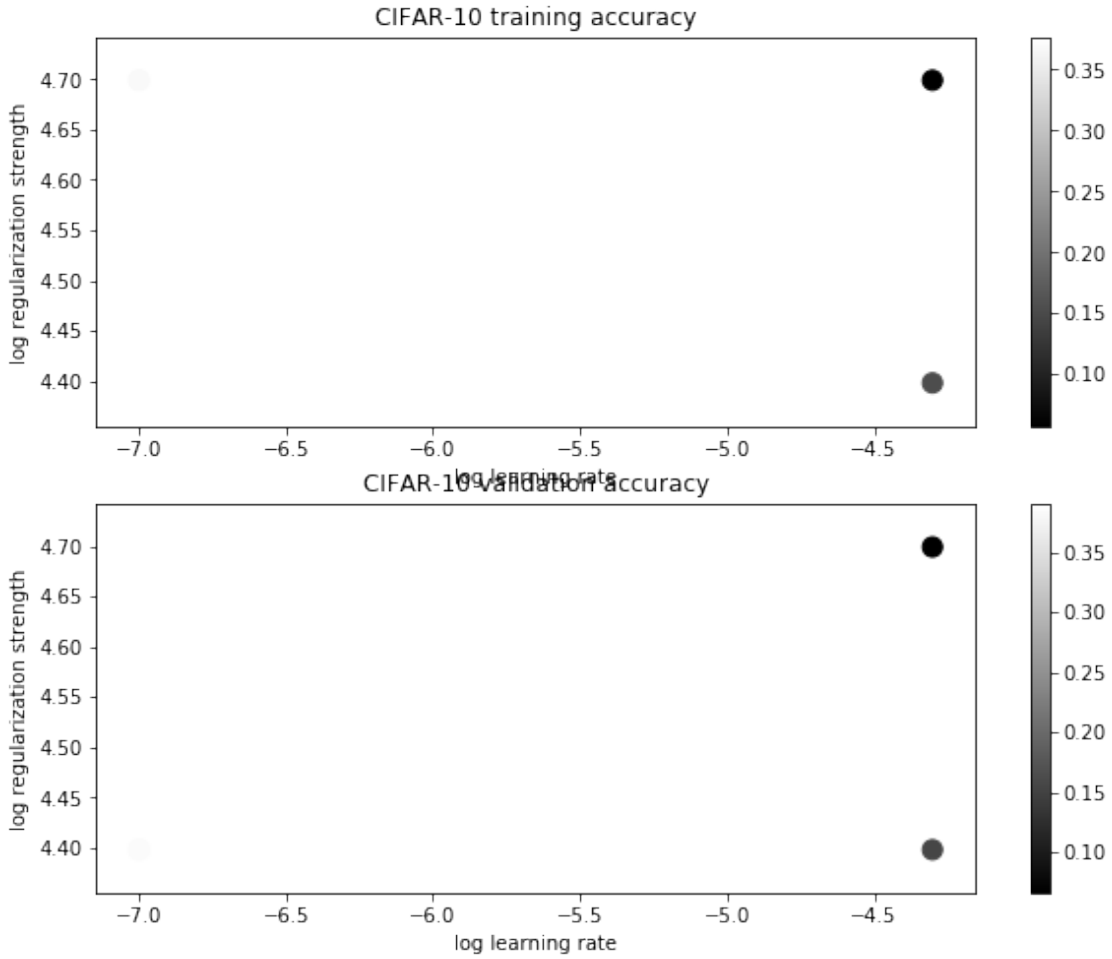
In [75]:
```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of about 0.4 on the validation set.
learning_rates = [1e-7, 5e-5]
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.
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.                                      #
################################################################################
for lr in learning_rates:
    for rs in regularization_strengths:
        svm = LinearSVM()
        loss_hist = svm.train(X_train, y_train, learning_rate=lr, reg=rs,
                              num_iters=1500, verbose=True)

        y_train_pred = svm.predict(X_train)
        y_val_pred = svm.predict(X_val)

        train_accuracy = np.mean(y_train == y_train_pred)
        val_accuracy = np.mean(y_val == y_val_pred)
```

```
            results[(lr, rs)] = (train_accuracy, val_accuracy)

            if val_accuracy > best_val:
                best_val = val_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)
```

```
iteration 0 / 1500: loss 399.749119
iteration 100 / 1500: loss 237.844294
iteration 200 / 1500: loss 144.924033
iteration 300 / 1500: loss 89.676472
iteration 400 / 1500: loss 56.088792
iteration 500 / 1500: loss 35.643393
iteration 600 / 1500: loss 23.258578
iteration 700 / 1500: loss 15.828474
iteration 800 / 1500: loss 11.723438
iteration 900 / 1500: loss 9.032353
iteration 1000 / 1500: loss 7.683780
iteration 1100 / 1500: loss 6.238153
iteration 1200 / 1500: loss 5.754255
iteration 1300 / 1500: loss 5.309717
iteration 1400 / 1500: loss 5.827387
iteration 0 / 1500: loss 787.293539
iteration 100 / 1500: loss 286.062313
iteration 200 / 1500: loss 107.591264
iteration 300 / 1500: loss 41.880970
iteration 400 / 1500: loss 18.498705
iteration 500 / 1500: loss 10.183269
iteration 600 / 1500: loss 6.891912
iteration 700 / 1500: loss 5.732447
iteration 800 / 1500: loss 6.209047
iteration 900 / 1500: loss 5.478543
iteration 1000 / 1500: loss 5.015930
iteration 1100 / 1500: loss 4.946343
iteration 1200 / 1500: loss 4.895653
iteration 1300 / 1500: loss 5.201358
iteration 1400 / 1500: loss 5.676970
```

```
iteration 0 / 1500: loss 408.460519
iteration 100 / 1500: loss 970.195048
iteration 200 / 1500: loss 851.775430
iteration 300 / 1500: loss 915.448736
iteration 400 / 1500: loss 1090.370445
iteration 500 / 1500: loss 1142.593551
iteration 600 / 1500: loss 1077.340001
iteration 700 / 1500: loss 1169.346690
iteration 800 / 1500: loss 1135.015821
iteration 900 / 1500: loss 933.427939
iteration 1000 / 1500: loss 1088.277089
iteration 1100 / 1500: loss 1089.163699
iteration 1200 / 1500: loss 1068.901479
iteration 1300 / 1500: loss 1055.981213
iteration 1400 / 1500: loss 1026.029886
iteration 0 / 1500: loss 786.963088
iteration 100 / 1500: loss 4026488911277294842472453680339531857920.000000
iteration 200 / 1500: loss 665546369201827971575780582020098785191622346676872968580417108735
iteration 300 / 1500: loss 110009484520651915061027183888921196077337217522939561880278245276
iteration 400 / 1500: loss 181836867339735612762554145199538857171920794273020160211271923250
iteration 500 / 1500: loss 300561778541207833458027145162341197791825380738005716049011663337
iteration 600 / 1500: loss 496804548172686274136513038250447901482422705347393399164886572319
iteration 700 / 1500: loss 821178129444785886596705835072550713392541882539897019564789639509
iteration 800 / 1500: loss 135734168046312443781705963403785118147535581318694619061446830214


/Users/hanwenzhao/Dropbox/Matlab/ComputerVision/HW5_2/hmwk5/hmwk5_2/classifiers/linear_svm.py:
  loss += 0.5 * reg * np.sum(W * W)
/anaconda3/envs/hmwk5/lib/python3.6/site-packages/numpy/core/_methods.py:32: RuntimeWarning: o
  return umr_sum(a, axis, dtype, out, keepdims)
/Users/hanwenzhao/Dropbox/Matlab/ComputerVision/HW5_2/hmwk5/hmwk5_2/classifiers/linear_svm.py:
  loss += 0.5 * reg * np.sum(W * W)


iteration 900 / 1500: loss inf
iteration 1000 / 1500: loss inf
iteration 1100 / 1500: loss inf
iteration 1200 / 1500: loss inf
iteration 1300 / 1500: loss inf
iteration 1400 / 1500: loss inf
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.376612 val accuracy: 0.384000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.368776 val accuracy: 0.390000
lr 5.000000e-05 reg 2.500000e+04 train accuracy: 0.154490 val accuracy: 0.155000
lr 5.000000e-05 reg 5.000000e+04 train accuracy: 0.057265 val accuracy: 0.066000
best validation accuracy achieved during cross-validation: 0.390000


In [76]: # 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()
```

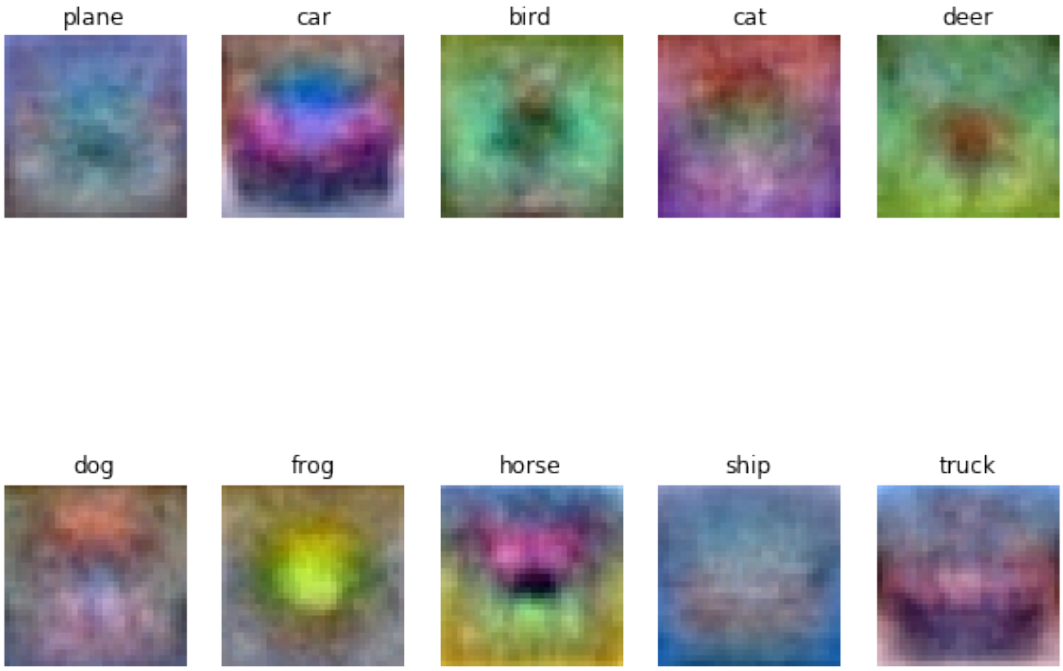


In [77]:
```
# Evaluate the best svm on test set
y_test_pred = best_svm.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
```

```
linear SVM on raw pixels final test set accuracy: 0.371000
```

In [78]:
```
# 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: The SVM weights are templetes for its corresponding object. The ship contains many blue pixels because ship usually floats on ocean.