Erik Arriaga
SID: 015707183

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 [97]:
```
from __future__ import print_function

import random
import numpy as np
from cecs551.data_utils import load_CIFAR10
import matplotlib.pyplot as plt


# This is a bit of magic to make matplotlib figures appear inline in t
he
# notebook rather than in a new window.
%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plo
ts
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

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

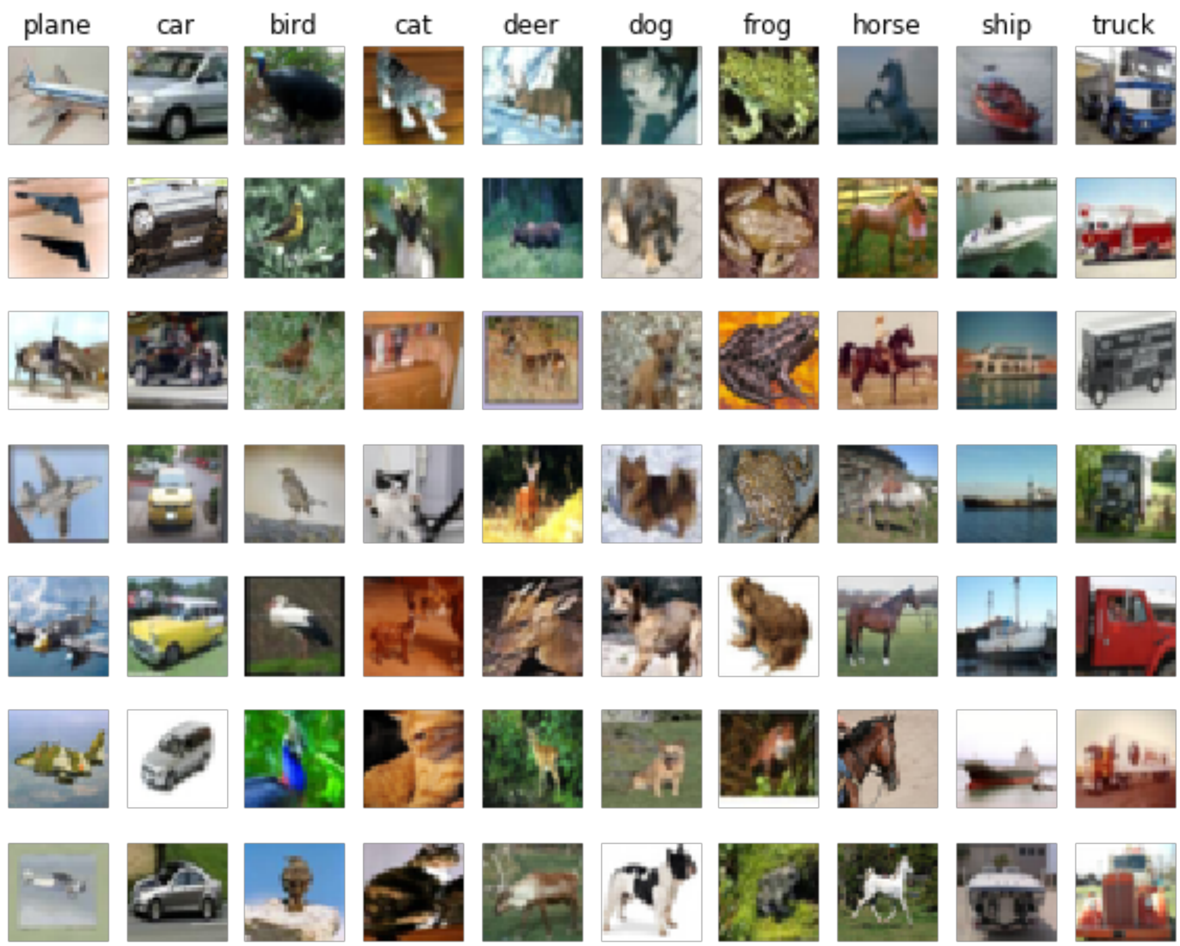
```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

CIFAR-10 Data Loading and Preprocessing

In [98]:
```
# Load the raw CIFAR-10 data.
cifar10_dir = 'cecs551/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 da
ta.
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 [99]:
```
# 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', 'hors
e', '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 [100]:
```
# 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 origina
l
# 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)
```

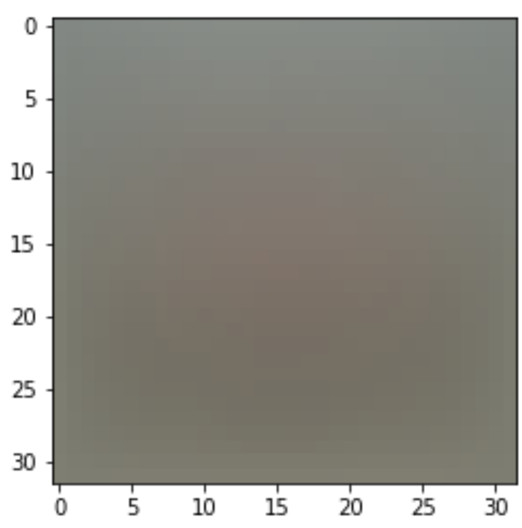
```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

In [101]:
```
# 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 [102]:
```
# 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 [103]:
```
# 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 [104]:
```
# third: append the bias dimension of ones (i.e. bias trick) so that o
ur 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 **cecs551/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 [105]:
```
# Evaluate the naive implementation of the loss we provided for you:
from cecs551.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: 8.485255
```

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 [106]:

```
# Once you've implemented the gradient, recompute it with the code bel
ow
# 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 dimen
sions, and
# compare them with your analytically computed gradient. The numbers s
hould match
# almost exactly along all dimensions.
from cecs551.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: 23.112341 analytic: 23.112341, relative error: 9.944593e-
13
numerical: 6.305958 analytic: 6.305958, relative error: 2.760470e-12
numerical: -9.772480 analytic: -9.772480, relative error: 1.283815e-
11
numerical: -13.438346 analytic: -13.438346, relative error: 3.961274
e-12
numerical: -7.283475 analytic: -7.283475, relative error: 3.090978e-
11
numerical: -8.745584 analytic: -8.745584, relative error: 2.514966e-
11
numerical: 16.489828 analytic: 16.489828, relative error: 1.346900e-
11
numerical: -27.769363 analytic: -27.769363, relative error: 1.200101
e-11
numerical: -11.557616 analytic: -11.557616, relative error: 5.111999
e-12
numerical: 2.740992 analytic: 2.750637, relative error: 1.756354e-03
numerical: -5.497730 analytic: -5.497730, relative error: 3.345098e-
11
numerical: 22.391418 analytic: 22.391418, relative error: 4.387062e-
14
numerical: 16.340837 analytic: 16.340837, relative error: 1.291663e-
11
numerical: -4.685618 analytic: -4.685618, relative error: 3.373116e-
12
numerical: 13.655848 analytic: 13.655848, relative error: 2.646620e-
11
numerical: -9.669880 analytic: -9.669880, relative error: 3.558664e-
11
numerical: -3.056508 analytic: -3.037875, relative error: 3.057409e-
03
numerical: 16.804458 analytic: 16.804458, relative error: 3.404732e-
11
numerical: 22.568309 analytic: 22.568309, relative error: 1.810060e-
11
numerical: 9.730973 analytic: 9.730973, relative error: 3.442767e-11
```

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: It is possible once in a while for a dimension in the gradcheck will not match because it may not be differentiable for example when max(x,y) function is not differentiable where x=y. But its not a commmon case so there is not need to really do anything about it.

In [107]:
```
# Next implement the function svm_loss_vectorized; for now only comput
e 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 cecs551.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: 8.485255e+00 computed in 0.099060s
Vectorized loss: 8.485255e+00 computed in 0.015369s
difference: 0.000000
```

In [108]:
```
# 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 0.097791s
Vectorized loss and gradient: computed in 0.003065s
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.

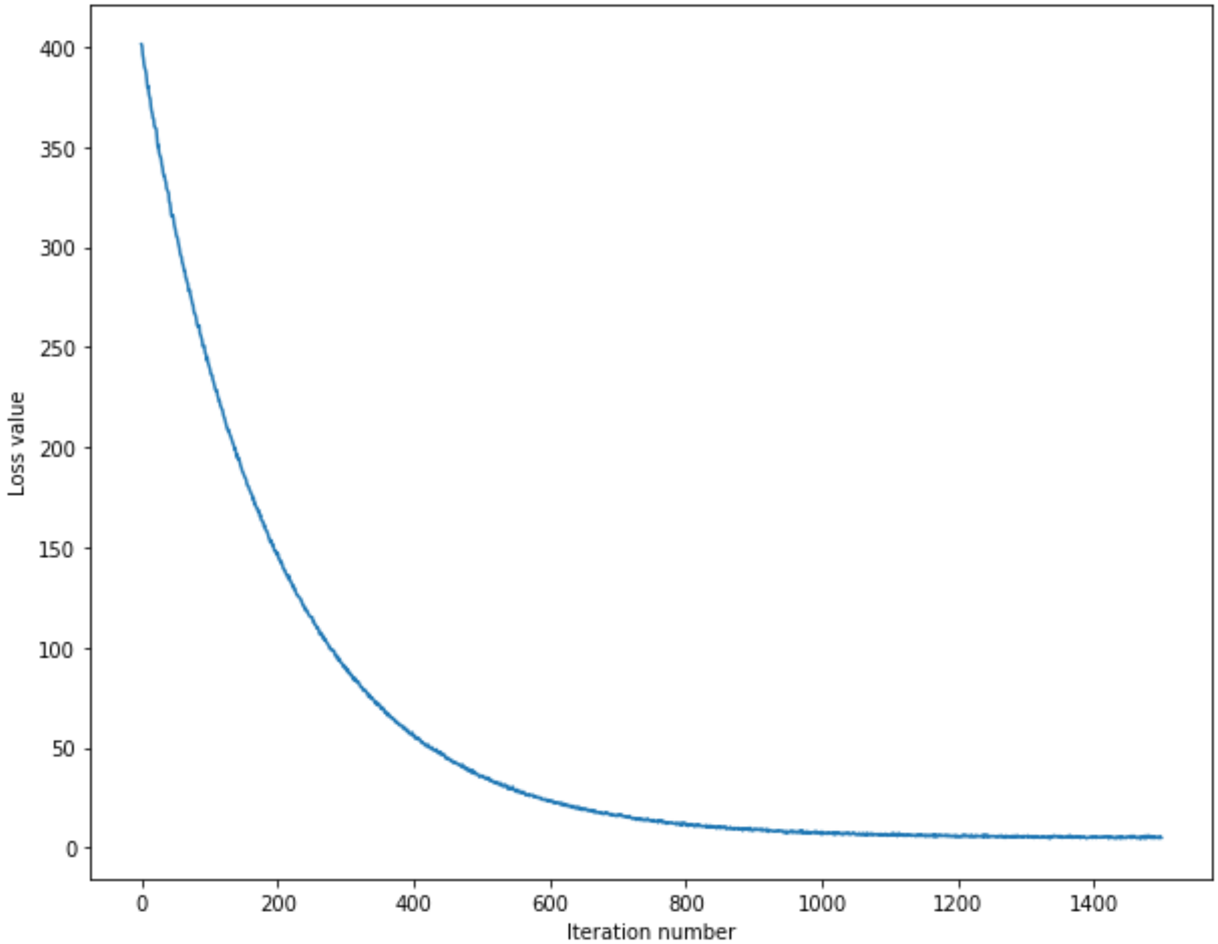
In [109]:

```
# In the file linear_classifier.py, implement SGD in the function
# LinearClassifier.train() and then run it with the code below.
from cecs551.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 401.256681
iteration 100 / 1500: loss 239.982612
iteration 200 / 1500: loss 147.237549
iteration 300 / 1500: loss 89.449265
iteration 400 / 1500: loss 55.740548
iteration 500 / 1500: loss 36.112024
iteration 600 / 1500: loss 23.184567
iteration 700 / 1500: loss 15.982779
iteration 800 / 1500: loss 11.801290
iteration 900 / 1500: loss 8.789944
iteration 1000 / 1500: loss 6.638772
iteration 1100 / 1500: loss 6.427840
iteration 1200 / 1500: loss 6.013772
iteration 1300 / 1500: loss 5.826273
iteration 1400 / 1500: loss 5.447535
That took 2.279000s
```

In [110]:

```
# 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 [111]:
```
# 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: 0.382265
validation accuracy: 0.379000
```

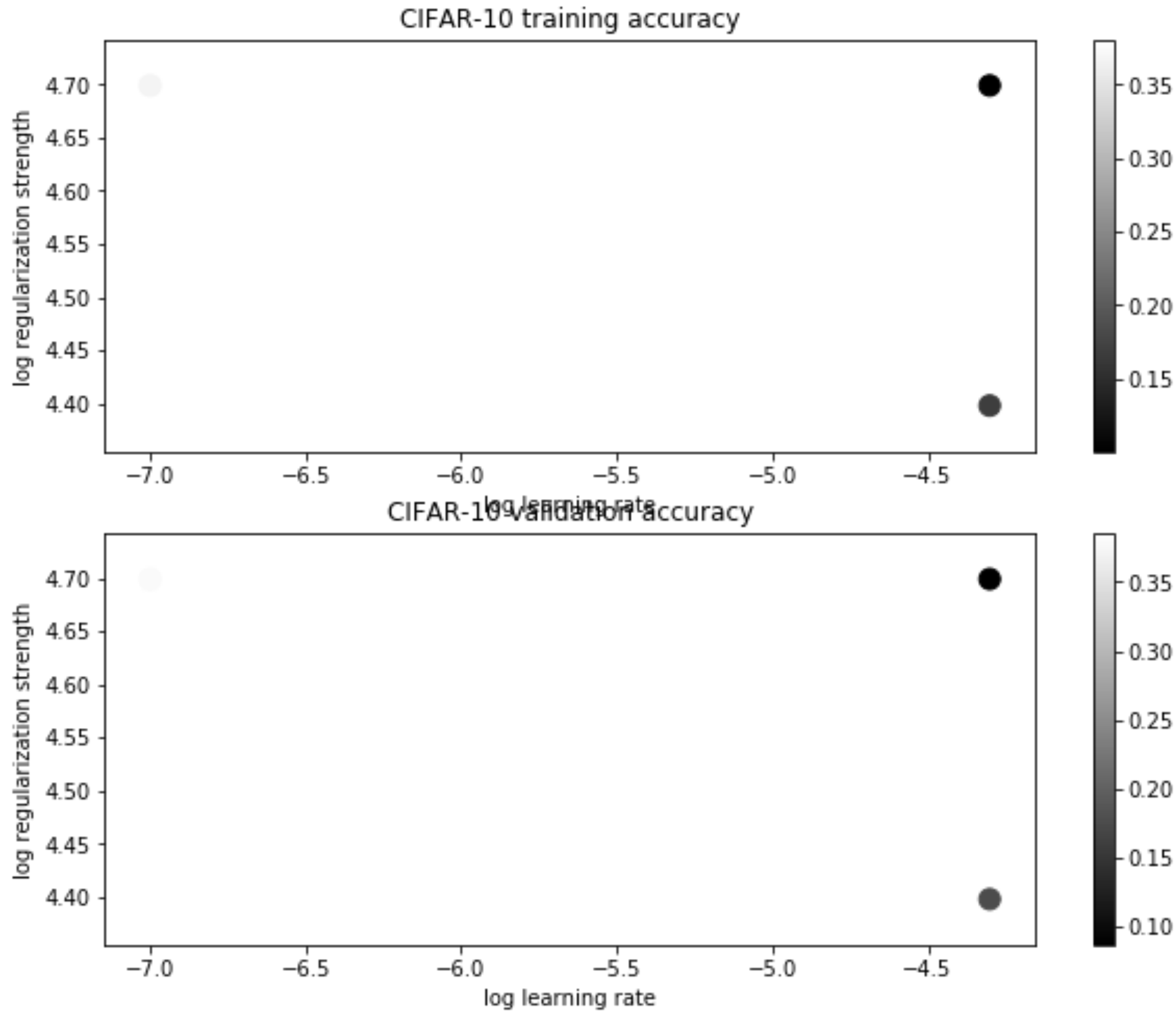
In [112]:
```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of about 0.4 on the 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 valid
ation rate.

#######################################################################
##########
# TODO:
#
# Write code that chooses the best hyperparameters by tuning on the va
lidation #
# set. For each combination of hyperparameters, train a linear SVM on
the      #
# training set, compute its accuracy on the training and validation se
ts, and  #
# store these numbers in the results dictionary. In addition, store th
e best   #
# validation accuracy in best_val and the LinearSVM object that achiev
es 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 vali
dation   #
# code with a larger value for num_iters.
#
#######################################################################
##########
for i in learning_rates:
    for j in regularization_strengths:
        svm=LinearSVM()
        svm.train(X_train, y_train, learning_rate=i, reg=j,num_iters=2
000, verbose=False)
        y_train_pred=svm.predict(X_train)
        train_accuracy = np.mean(y_train_pred == y_train)
        y_val_pred=svm.predict(X_val)
        val_accuracy = np.mean(y_val_pred == y_val)
        results[(i,j)] = (train_accuracy, val_accuracy)
```

```
        if best_val < val_accuracy:
            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)
```

```
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.380571 val accura
cy: 0.385000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.367673 val accura
cy: 0.379000
lr 5.000000e-05 reg 2.500000e+04 train accuracy: 0.169102 val accura
cy: 0.176000
lr 5.000000e-05 reg 5.000000e+04 train accuracy: 0.100265 val accura
cy: 0.087000
best validation accuracy achieved during cross-validation: 0.385000
```

In [113]:
```
# 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()
```

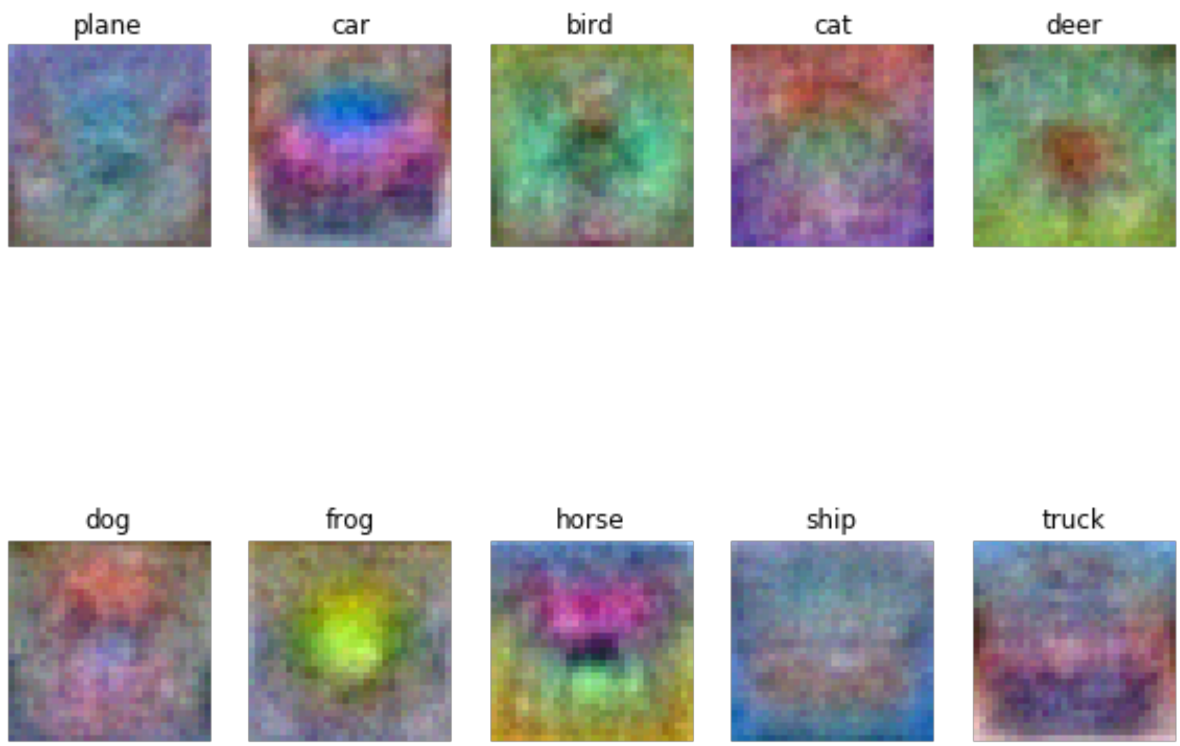


In [114]:

```
# 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_ac
curacy)
```

```
linear SVM on raw pixels final test set accuracy: 0.375000
```

In [115]:
```
# Visualize the learned weights for each class.
# Depending on your choice of learning rate and regularization strengt
h, 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', 'hors
e', '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 they are a mixture of parts from training images and it takes different aspects from them and combine it to one label (eg. plane, car, dog, etc). In this case horse, since its easier to explain, appears to have one long horse with two heads which can mean that pictures could have either the two horses, features of one hourse facing the left or one horse facing right but since it is combined then it may look like a horse with 2 heads composed by the differnt different images. It is blurry because of the low acurract obtained.

```
In [ ]:
```