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

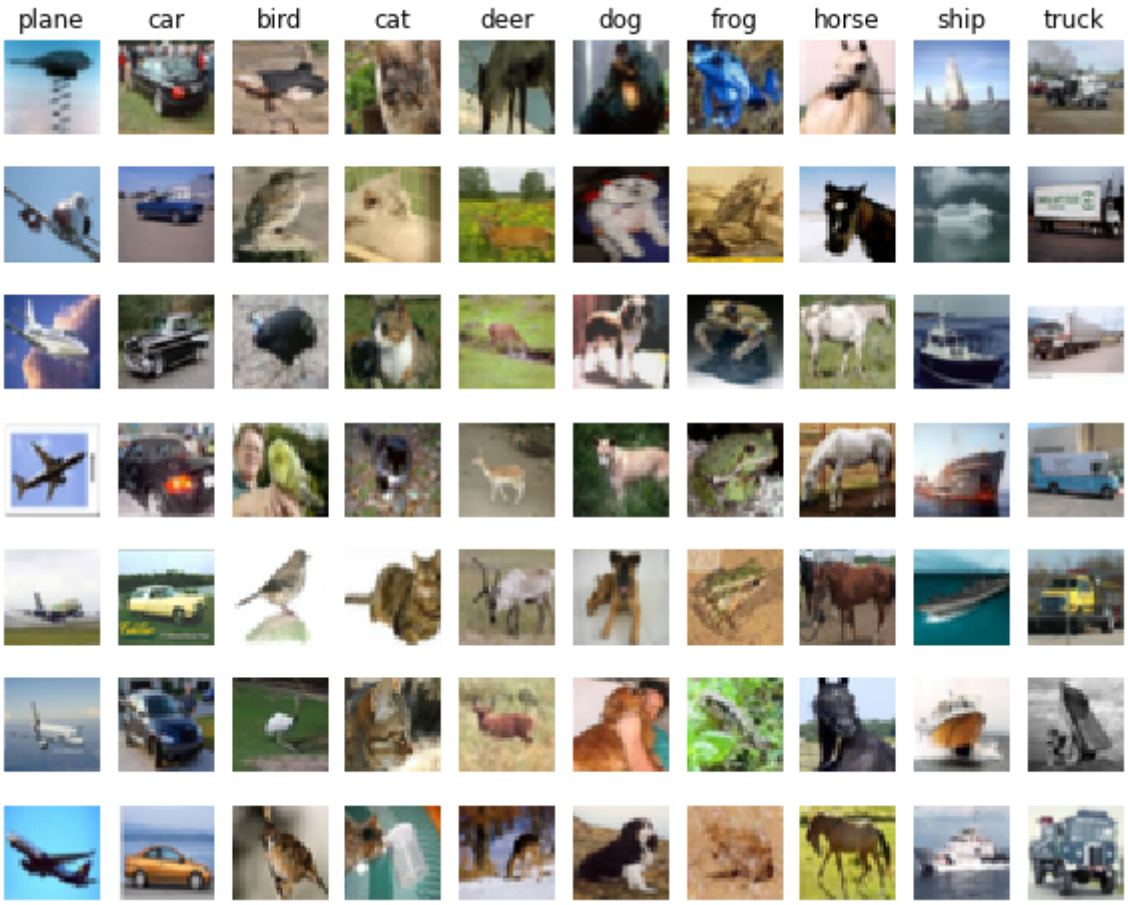
In [1]:
```
# Run some setup code for this notebook.
from __future__ import print_function
import random
import numpy as np
from cs682.data_utils import load_CIFAR10
import matplotlib.pyplot as plt


# 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 plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

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

CIFAR-10 Data Loading and Preprocessing

In [2]:
```
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs682/datasets/cifar-10-batches-py'

# Cleaning up variables to prevent loading data multiple times (which ma
y 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 dat
a.
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:  (50000, 32, 32, 3)
Training labels shape:  (50000,)
Test data shape:  (10000, 32, 32, 3)
Test labels shape:  (10000,)
```

In [3]:
```
# 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', '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 [4]:
```
# 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)
```

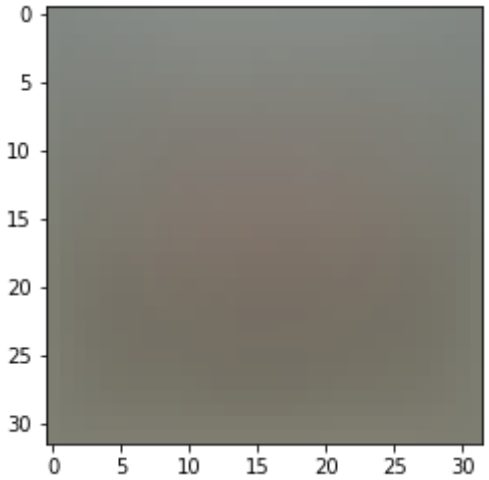
```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

In [5]:
```
# 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 [6]:
```
# 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 th
e 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 [7]:
```
# 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 [8]:
```
# 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 [9]:
```
# 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: 8.798825
```

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 [10]:
```
# 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 dimensi
ons, and
# compare them with your analytically computed gradient. The numbers sho
uld 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: 21.075643 analytic: 21.075643, relative error: 2.869140e-12
numerical: -23.347696 analytic: -23.347696, relative error: 6.095826e-1
2
numerical: 21.065230 analytic: 21.065230, relative error: 1.550595e-12
numerical: -1.627262 analytic: -1.662916, relative error: 1.083635e-02
numerical: 18.584493 analytic: 18.584493, relative error: 2.655212e-11
numerical: 17.996371 analytic: 17.996371, relative error: 1.598412e-11
numerical: 38.992193 analytic: 38.992193, relative error: 1.951559e-12
numerical: 5.380711 analytic: 5.380711, relative error: 8.776999e-11
numerical: 38.182320 analytic: 38.182320, relative error: 1.098502e-12
numerical: -17.859809 analytic: -17.859809, relative error: 3.995553e-1
2
numerical: -30.226370 analytic: -30.226370, relative error: 1.025274e-1
2
numerical: -46.757943 analytic: -46.757943, relative error: 2.821629e-1
2
numerical: 4.469970 analytic: 4.469970, relative error: 3.751814e-11
numerical: 0.501495 analytic: 0.501495, relative error: 2.158923e-10
numerical: -55.380953 analytic: -55.380953, relative error: 6.731372e-1
2
numerical: -5.721403 analytic: -5.721403, relative error: 7.292765e-12
numerical: -4.365053 analytic: -4.365053, relative error: 7.396147e-11
numerical: 46.799835 analytic: 46.799835, relative error: 1.716847e-12
numerical: 5.541011 analytic: 5.541011, relative error: 5.879854e-11
numerical: 3.597039 analytic: 3.597039, relative error: 4.434875e-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: *Yes, it is possible that once in a while a dimension in the gradcheck will not match exactly. It could be caused by the discontinuous nature of the SVM function; meaning it is not strictly differentiable. Such an example in one dimension would be x = 0.001. L(x) = max(0, x) = 0. So the analytical gradient would be 0, whereas, the numerical gradient would be non-zero.*

In [11]:
```
# 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 - ti
c))

# The losses should match but your vectorized implementation should be m
uch faster.
print('difference: %f' % (loss_naive - loss_vectorized))
```

```
Naive loss: 8.798825e+00 computed in 0.079120s
Vectorized loss: 8.798825e+00 computed in 0.004615s
difference: 0.000000
```

In [12]:
```
# Complete the implementation of svm_loss_vectorized, and compute the gr
adient
# of the loss function in a vectorized way.

# The naive implementation and the vectorized implementation should matc
h, 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 compu
ted
# by the two implementations. The gradient on the other hand is a matri
x, 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 0.084399s
Vectorized loss and gradient: computed in 0.005446s
difference: 0.000000
```

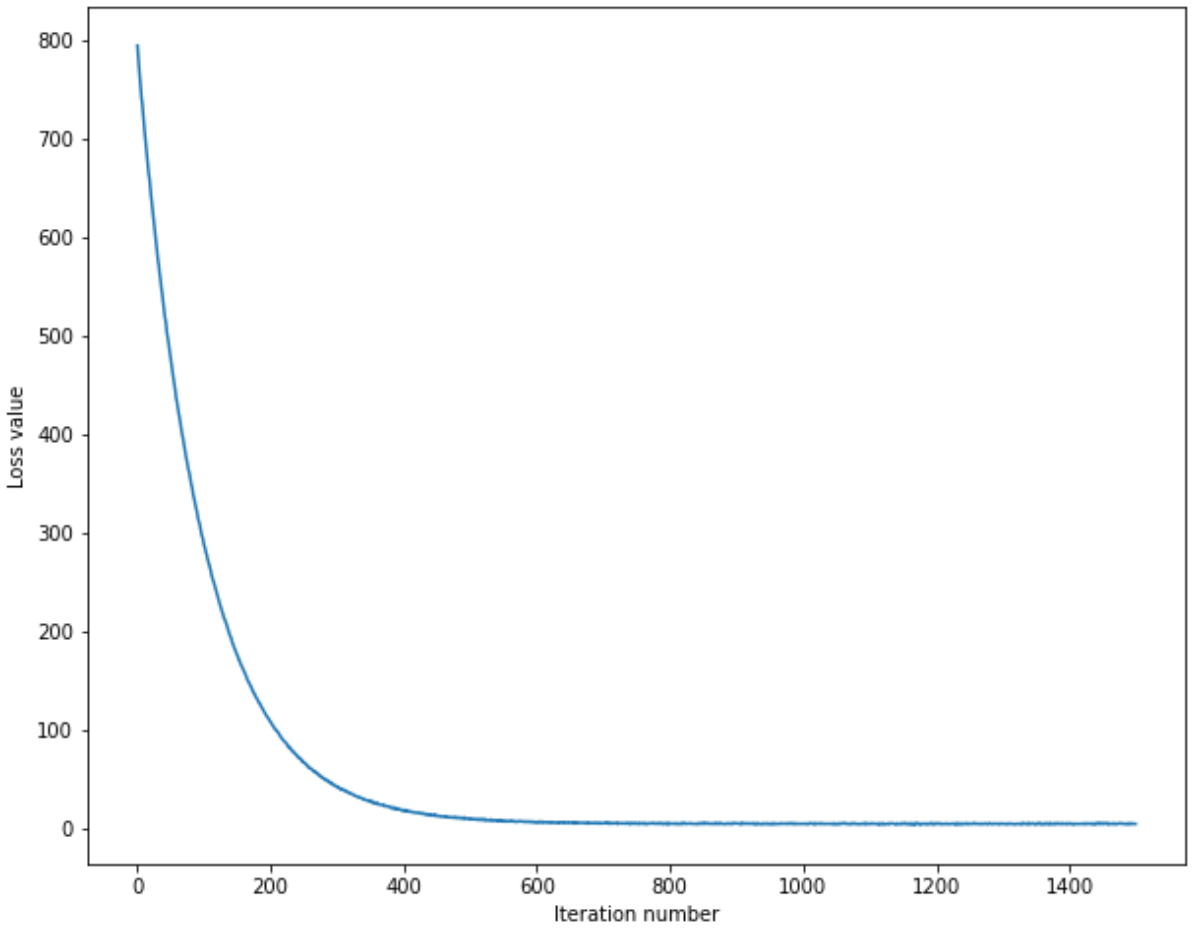
Stochastic Gradient Descent

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 [13]:
```
# 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 794.912996
iteration 100 / 1500: loss 287.932956
iteration 200 / 1500: loss 108.324655
iteration 300 / 1500: loss 42.736626
iteration 400 / 1500: loss 19.395020
iteration 500 / 1500: loss 9.680554
iteration 600 / 1500: loss 7.062760
iteration 700 / 1500: loss 5.893017
iteration 800 / 1500: loss 5.411213
iteration 900 / 1500: loss 5.645085
iteration 1000 / 1500: loss 5.285874
iteration 1100 / 1500: loss 4.944847
iteration 1200 / 1500: loss 5.103733
iteration 1300 / 1500: loss 5.317000
iteration 1400 / 1500: loss 5.734901
That took 3.601004s
```

In [14]:
```
# 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 [15]:
```
# Write the LinearSVM.predict function and evaluate the performance on b
oth 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: 0.372898
validation accuracy: 0.375000
```

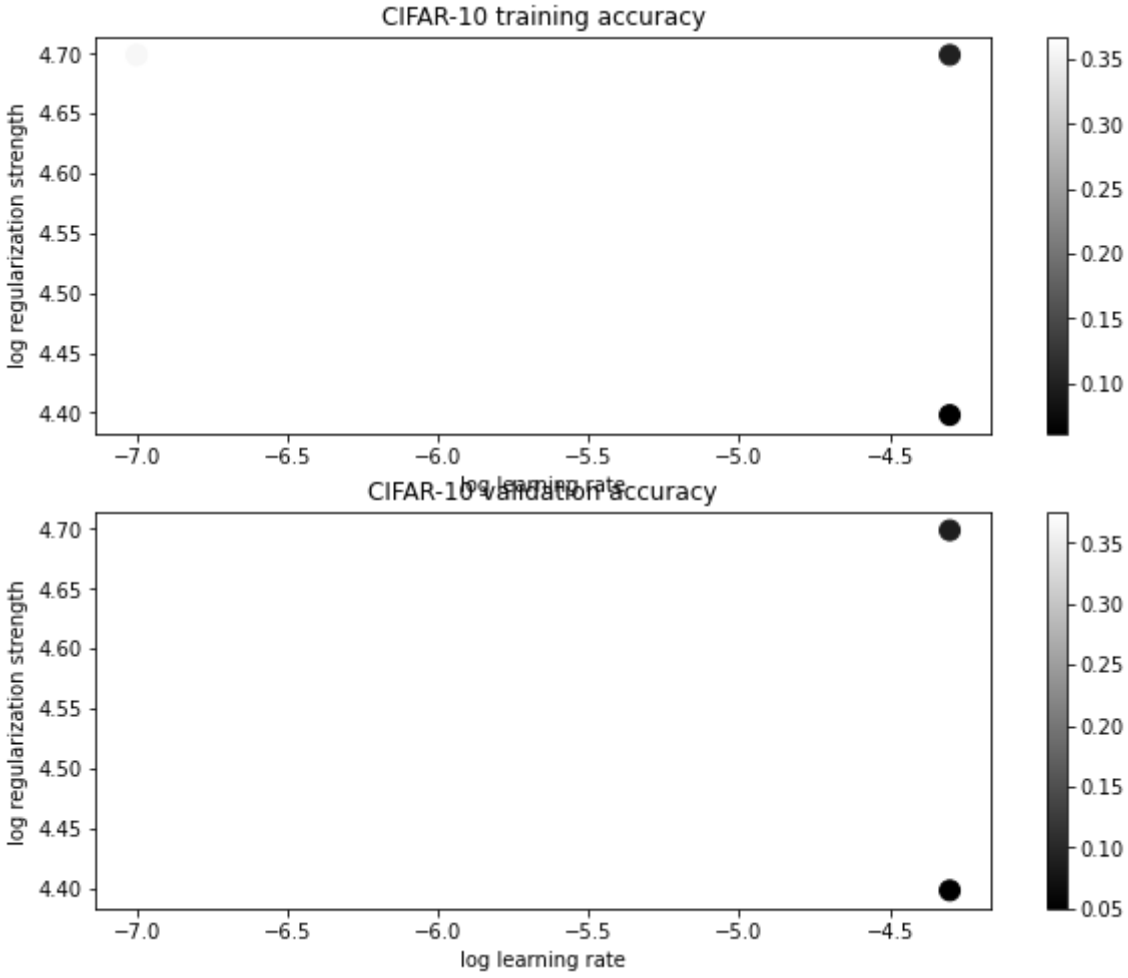
In [16]:
```
# Use the validation set to tune hyperparameters (regularization strengt
h and
# learning rate). You should experiment with different ranges for the le
arning
# rates and regularization strengths; if you are careful you should be a
ble to
# get a classification accuracy of about 0.4 on the 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 f
raction
# of data points that are correctly classified.
results = {}
best_val = -1   # The highest validation accuracy that we have seen so f
ar.
best_svm = None # The LinearSVM object that achieved the highest validat
ion rate.

################################################################################
########
# TODO:
#
# Write code that chooses the best hyperparameters by tuning on the vali
dation #
# set. For each combination of hyperparameters, train a linear SVM on th
e      #
# training set, compute its accuracy on the training and validation set
s, 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 y
ou are #
# confident that your validation code works, you should rerun the valida
tion   #
# code with a larger value for num_iters.
#
################################################################################
########
for learning_rate in learning_rates:
    for regularization_strength in regularization_strengths:
        svm2 = LinearSVM()
        lost_hist2 = svm2.train(X_train, y_train, learning_rate=learning
_rate, reg=regularization_strength,
                      num_iters=1500, verbose=True)
        y_train_pred2 = svm2.predict(X_train)
```

```
        y_val_pred2 = svm2.predict(X_val)
        validation_accuracy = np.mean(y_val == y_val_pred2)
        training_acccuracy = np.mean(y_train == y_train_pred2)
        results[(learning_rate, regularization_strength)] = (training_ac
ccuracy, validation_accuracy)
        if validation_accuracy>best_val:
            best_val = validation_accuracy
            best_svm = svm2
################################################################################
########
#                               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 794.525860
iteration 100 / 1500: loss 289.448893
iteration 200 / 1500: loss 107.915433
iteration 300 / 1500: loss 42.616255
iteration 400 / 1500: loss 19.276410
iteration 500 / 1500: loss 10.236430
iteration 600 / 1500: loss 7.015447
iteration 700 / 1500: loss 5.950552
iteration 800 / 1500: loss 5.631508
iteration 900 / 1500: loss 5.585682
iteration 1000 / 1500: loss 5.119921
iteration 1100 / 1500: loss 5.361187
iteration 1200 / 1500: loss 5.215365
iteration 1300 / 1500: loss 5.499823
iteration 1400 / 1500: loss 5.671152
iteration 0 / 1500: loss 1570.540102
iteration 100 / 1500: loss 212.354226
iteration 200 / 1500: loss 33.104767
iteration 300 / 1500: loss 9.943550
iteration 400 / 1500: loss 5.874009
iteration 500 / 1500: loss 5.715917
iteration 600 / 1500: loss 5.589037
iteration 700 / 1500: loss 5.794695
iteration 800 / 1500: loss 5.506864
iteration 900 / 1500: loss 5.919477
iteration 1000 / 1500: loss 5.819220
iteration 1100 / 1500: loss 5.399934
iteration 1200 / 1500: loss 5.977091
iteration 1300 / 1500: loss 6.408548
iteration 1400 / 1500: loss 5.415373
iteration 0 / 1500: loss 799.463349
iteration 100 / 1500: loss 433509728708405248137338445838499708928.0000
00
iteration 200 / 1500: loss 716556862102529005932445901738682908500826680
802921243348620202263036559360.000000
iteration 300 / 1500: loss 11844111045812094621393973869415942368696529
271784867037869299840465783138549060030395126052266749057793261568.0000
00
iteration 400 / 1500: loss 19577366973209663426658079951151586942833567
702447953547459667089052752410939354003877541262732199251967425383444400
714479524819931898838922559488.000000
iteration 500 / 1500: loss 32359819670826257586542777876592242801516119
771928473771585729977665590838198477464096849067224774840690854377459767
729109158791552078407699339230187367298683354827231810182709248 0.00000
0
iteration 600 / 1500: loss 53488190243425491400860378387082556521331775
142127960180820150392550022022903222855021192282408278168399959848049207
725863265176083178704007680172383595155689368642920966998961669261003826
844697355195777870645428224.000000
iteration 700 / 1500: loss 884116946453869555532449646919866864464827986
338070806609396810003704318824222547489504140865923549394355686202526591
282930193577842755381742555118957884207258003755050436324202364714361828
427098720713328448820513359827345473842656217793320326266880.000000
iteration 800 / 1500: loss 14613745042588961831555877286922697441850823309
946533045248341415018122753056001024522294701379603354232747237837512206
486447142211534481195657995591339141173049253068066650128902620709 8
```

```
98339255360254316316273833124891094497537349146307247949709619428703354
05779815153175703433487843328.000000

/Users/satwikgoyal/Desktop/CS 682/assignment1/cs682/classifiers/linear_
svm.py:84: RuntimeWarning: overflow encountered in double_scalars
  loss += reg * np.sum(W * W)
/opt/anaconda3/lib/python3.8/site-packages/numpy/core/fromnumeric.py:8
7: RuntimeWarning: overflow encountered in reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
/Users/satwikgoyal/Desktop/CS 682/assignment1/cs682/classifiers/linear_
svm.py:84: RuntimeWarning: overflow encountered in multiply
  loss += 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
iteration 0 / 1500: loss 1572.281940
iteration 100 / 1500: loss 44009254299085813569993141164184559579897168
82416770705616686983438730970659142655293640178815401255049191924344882
590646272.000000
iteration 200 / 1500: loss 11364289154851009982558777891966387111999946
89216821240878160709415023508621816701207097528446449522575474948169921
27441528071880855694889170614873263511352817129763945355163945076175909
12139647251500538970445149595853182263275112694191879094272.000000
iteration 300 / 1500: loss inf
iteration 400 / 1500: loss inf
iteration 500 / 1500: loss inf

/Users/satwikgoyal/Desktop/CS 682/assignment1/cs682/classifiers/linear_
svm.py:105: RuntimeWarning: overflow encountered in multiply
  dW += 2*reg*W
/Users/satwikgoyal/Desktop/CS 682/assignment1/cs682/classifiers/linear_
classifier.py:70: RuntimeWarning: invalid value encountered in subtract
  self.W -= learning_rate*grad

iteration 600 / 1500: loss nan
iteration 700 / 1500: loss nan
iteration 800 / 1500: loss nan
iteration 900 / 1500: loss nan
iteration 1000 / 1500: loss nan
iteration 1100 / 1500: loss nan
iteration 1200 / 1500: loss nan
iteration 1300 / 1500: loss nan
iteration 1400 / 1500: loss nan
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.367388 val accuracy:
0.375000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.357408 val accuracy:
0.375000
lr 5.000000e-05 reg 2.500000e+04 train accuracy: 0.061184 val accuracy:
0.050000
lr 5.000000e-05 reg 5.000000e+04 train accuracy: 0.100265 val accuracy:
0.087000
best validation accuracy achieved during cross-validation: 0.375000
```

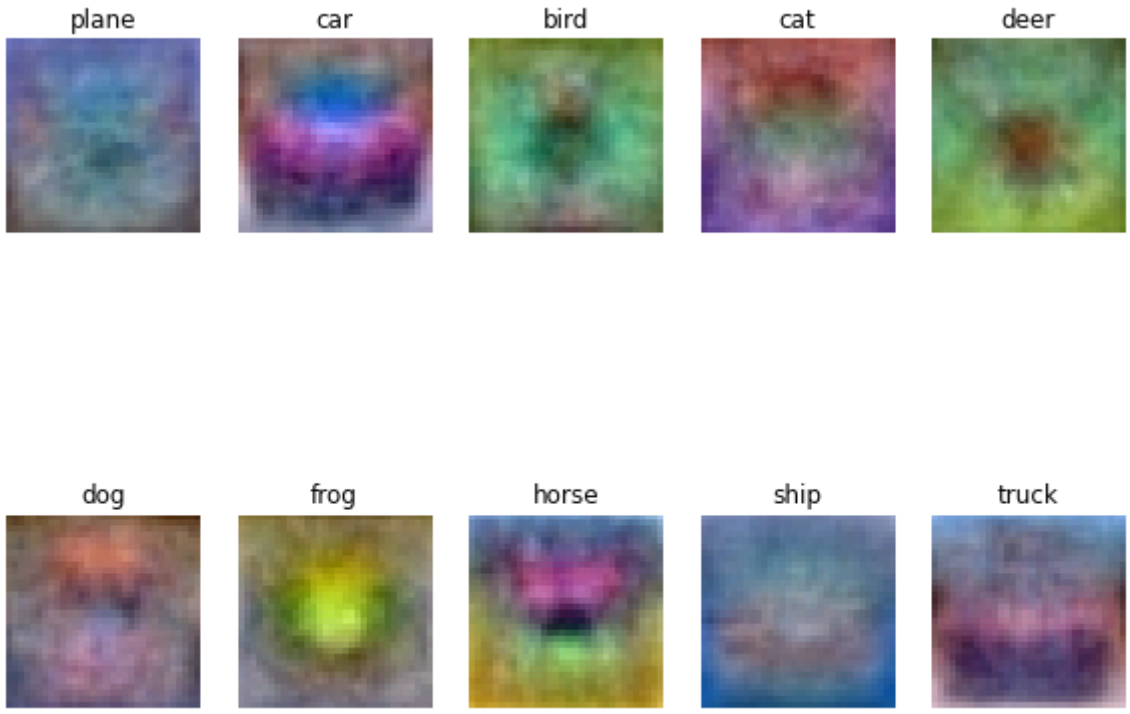
In [17]:
```
# Visualize the cross-validation results
import math
x_scatter = [math.log10(x[0]) for x in results]
y_scatter = [math.log10(x[1]) for x in results]

# 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 2
0
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 [18]:
```
# 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_accu
racy)
```

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

In [19]:
```
# 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', '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: *Each visualized SVM weight is the looks like an average over all the images in a class. It is because each SVM weight is computed using the training data in each class, and so images are mapped over each other to give the correct class' maximum weight. For example, the horse image looks like an image with 2 horse heads probably because there were many pictures in the class with the horse's head facing right and many pictures with the horse's head facing the left. Siminaly, in the car class we can sort of see a car's pink/red front with a blueish windshield. Similarly, for the deer class we can see a brown coloured deer in the center on the image with green background which is most probably the greenery around it.*

In [ ]: