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 [1]:

```
# Run some setup code for this notebook.
from __future__ import print_function
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt


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

# Some more magic so that the notebook will reload external python modules;
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
print ("setup done\n")
```

```
setup done
```

CIFAR-10 Data Loading and Preprocessing

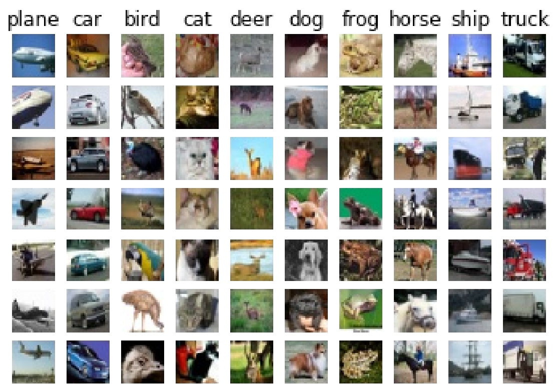
In [2]:

```
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs231n/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 data.
print('Training data shape: ', X_train.shape)
print('Training labels shape: ', y_train.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

```
Training data shape:  (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]:

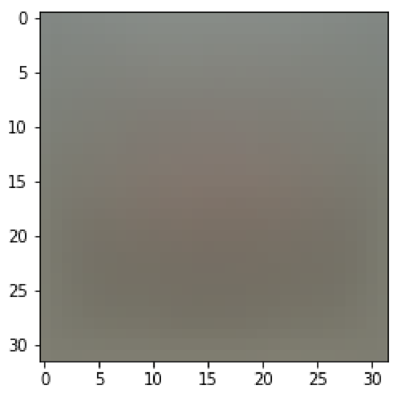
```
# Preprocessing: reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_val = np.reshape(X_val, (X_val.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))
print ("done reshaping\n")
# As a sanity check, 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)
```

```
done reshaping

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 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 [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
print ("done with subtractions\n")
# why do we do this? => Stack overflow :  serves to "center" the data. Additionally, you ideally w
ould like to
# divide by the sttdev of that feature or pixel as well if you want to normalize each feature valu
e to a z-score
```

```
done with subtractions
```

In [8]:

```python
# 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 **cs231n/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 [9]:

```python
# Evaluate the naive implementation of the loss we provided for you:
from cs231n.classifiers.linear_svm import svm_loss_naive
import time

# generate a random SVM weight matrix of small numbers
W = np.random.randn(3073, 10) * 0.0001

loss, grad = svm_loss_naive(W, X_dev, y_dev, 0.000005)
print('loss: %f' % (loss, ))
```

```
loss: 9.111736
```

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]:

```python
# 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 cs231n.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: -6.082993 analytic: -6.082993, relative error: 5.451360e-11
numerical: -14.411476 analytic: -14.411476, relative error: 2.029289e-11
numerical: 6.004580 analytic: 6.004580, relative error: 1.467054e-11
```

```
numerical: 32.999353 analytic: 32.999353, relative error: 2.152233e-12
numerical: 15.765745 analytic: 15.765745, relative error: 2.137386e-11
numerical: -16.528952 analytic: -16.590635, relative error: 1.862436e-03
numerical: 13.620796 analytic: 13.620796, relative error: 5.650428e-12
numerical: 8.930517 analytic: 8.930517, relative error: 1.616461e-11
numerical: -1.832288 analytic: -1.832288, relative error: 2.885587e-11
numerical: 8.354073 analytic: 8.354073, relative error: 1.373879e-11
numerical: 7.024489 analytic: 7.024489, relative error: 3.145639e-11
numerical: 3.358550 analytic: 3.358550, relative error: 2.447294e-11
numerical: 20.798334 analytic: 20.798334, relative error: 9.277681e-12
numerical: -6.027563 analytic: -6.027563, relative error: 5.163169e-12
numerical: 40.565396 analytic: 40.565396, relative error: 4.276439e-12
numerical: -54.503318 analytic: -54.440156, relative error: 5.797735e-04
numerical: 11.681741 analytic: 11.681741, relative error: 1.957853e-11
numerical: 11.129533 analytic: 11.129533, relative error: 1.489537e-12
numerical: -34.766760 analytic: -34.797493, relative error: 4.418034e-04
numerical: 5.985280 analytic: 5.985835, relative error: 4.631813e-05
```

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 SVM loss function is not strictly speaking differentiable. The point is at the very hinge of the loss function. For [;f(x) = max(-x,0);] at x=0, there is no real gradient. **source of help: reddit**

In [11]:

```python
# 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 cs231n.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: 9.111736e+00 computed in 0.142968s
Vectorized loss: 9.111736e+00 computed in 0.065855s
difference: 0.000000
```

In [12]:

```python
# 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 0.122224s
Vectorized loss and gradient: computed in 0.008875s
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.

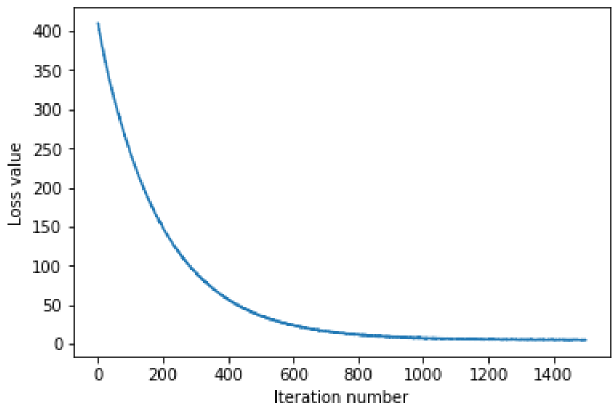
In [13]:

```
# In the file linear_classifier.py, implement SGD in the function
# LinearClassifier.train() and then run it with the code below.
from cs231n.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 409.809867
iteration 100 / 1500: loss 243.497321
iteration 200 / 1500: loss 148.842409
iteration 300 / 1500: loss 91.741212
iteration 400 / 1500: loss 57.079103
iteration 500 / 1500: loss 36.245337
iteration 600 / 1500: loss 24.165671
iteration 700 / 1500: loss 15.839547
iteration 800 / 1500: loss 11.512384
iteration 900 / 1500: loss 8.952027
iteration 1000 / 1500: loss 7.585314
iteration 1100 / 1500: loss 6.316617
iteration 1200 / 1500: loss 6.010636
iteration 1300 / 1500: loss 5.223708
iteration 1400 / 1500: loss 5.053512
That took 55.332310s
```

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]:

```
# 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]
```

```python
# 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.
best_lr = None
best_reg = None

################################################################################
# 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.                                      #
################################################################################

range_lr = np.linspace(learning_rates[0],learning_rates[1],5)
range_reg = np.linspace(regularization_strengths[0],regularization_strengths[1],5)

print ("strating loop ! This may take a while\n\n")
# loop through all the combinations
for cur_lr in range_lr: #go over the learning rates
    for cur_reg in range_reg:#go over the regularization strength
        # initiate linear classifier with hyperparameters
        svm = LinearSVM()
        svm.train(X_train, y_train, learning_rate=cur_lr, reg=cur_reg,num_iters=1600, verbose=True)

        # Training
        y_pred = svm.predict(X_train)
        train_accuracy = np.mean(np.equal(y_train, y_pred, dtype=float))


        # Validation
        y_pred = svm.predict(X_val)
        val_accuracy = np.mean(np.equal(y_val, y_pred, dtype=float))

        results[(cur_lr, cur_reg)] = (train_accuracy, val_accuracy)

        if val_accuracy > best_val:
            best_val = val_accuracy
            best_svm = svm
            best_lr = cur_lr
            best_reg = cur_reg

# best results



# 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_accura
cy))

print('best validation accuracy achieved during cross-validation: %f' % best_val)
```

```
strating loop ! This may take a while


iteration 0 / 1600: loss 401.836315
iteration 100 / 1600: loss 239.776630
iteration 200 / 1600: loss 145.831922
iteration 300 / 1600: loss 89.607330
iteration 400 / 1600: loss 56.353273
iteration 500 / 1600: loss 36.116630
iteration 600 / 1600: loss 24.513271
iteration 700 / 1600: loss 16.352047
iteration 800 / 1600: loss 11.533623
iteration 900 / 1600: loss 8.731652
iteration 1000 / 1600: loss 7.601978
iteration 1100 / 1600: loss 6.671071
iteration 1200 / 1600: loss 6.382279
iteration 1300 / 1600: loss 5.494139
iteration 1400 / 1600: loss 5.262041
iteration 1500 / 1600: loss 5.637816
iteration 0 / 1600: loss 503.367777
iteration 100 / 1600: loss 263.066988
iteration 200 / 1600: loss 142.368742
iteration 300 / 1600: loss 77.739737
iteration 400 / 1600: loss 43.858984
iteration 500 / 1600: loss 25.262528
iteration 600 / 1600: loss 16.045703
iteration 700 / 1600: loss 10.985047
iteration 800 / 1600: loss 8.169537
iteration 900 / 1600: loss 6.638457
iteration 1000 / 1600: loss 5.938618
iteration 1100 / 1600: loss 5.786087
iteration 1200 / 1600: loss 5.353577
iteration 1300 / 1600: loss 4.934477
iteration 1400 / 1600: loss 4.897986
iteration 1500 / 1600: loss 5.008100
iteration 0 / 1600: loss 600.675263
iteration 100 / 1600: loss 278.694523
iteration 200 / 1600: loss 133.290576
iteration 300 / 1600: loss 64.443730
iteration 400 / 1600: loss 32.788222
iteration 500 / 1600: loss 18.123590
iteration 600 / 1600: loss 11.398718
iteration 700 / 1600: loss 7.933774
iteration 800 / 1600: loss 6.271571
iteration 900 / 1600: loss 5.444358
iteration 1000 / 1600: loss 5.144270
iteration 1100 / 1600: loss 5.215012
iteration 1200 / 1600: loss 4.543057
iteration 1300 / 1600: loss 5.375985
iteration 1400 / 1600: loss 5.070968
iteration 1500 / 1600: loss 4.406947
iteration 0 / 1600: loss 698.579658
iteration 100 / 1600: loss 286.523018
iteration 200 / 1600: loss 121.878573
iteration 300 / 1600: loss 53.453330
iteration 400 / 1600: loss 25.190756
iteration 500 / 1600: loss 13.351961
iteration 600 / 1600: loss 8.709801
iteration 700 / 1600: loss 6.991149
iteration 800 / 1600: loss 6.081343
iteration 900 / 1600: loss 5.178135
iteration 1000 / 1600: loss 5.473010
iteration 1100 / 1600: loss 5.057360
iteration 1200 / 1600: loss 5.411360
iteration 1300 / 1600: loss 4.893508
iteration 1400 / 1600: loss 5.181273
iteration 1500 / 1600: loss 5.278916
iteration 0 / 1600: loss 791.797469
iteration 100 / 1600: loss 289.004391
iteration 200 / 1600: loss 108.474657
iteration 300 / 1600: loss 42.909983
iteration 400 / 1600: loss 19.154178
iteration 500 / 1600: loss 10.038297
iteration 600 / 1600: loss 6.961636
iteration 700 / 1600: loss 6.327182
iteration 800 / 1600: loss 5.620939
iteration 900 / 1600: loss 4.695515
iteration 1000 / 1600: loss 5.287873
iteration 1100 / 1600: loss 5.410183
iteration 1200 / 1600: loss 5.603413
iteration 1300 / 1600: loss 5.218436
iteration 1400 / 1600: loss 5.353349
iteration 1500 / 1600: loss 5.227169
iteration 0 / 1600: loss 407.162180
iteration 100 / 1600: loss 41.777023
iteration 200 / 1600: loss 51.189676
iteration 300 / 1600: loss 79.272648
iteration 400 / 1600: loss 61.839193
iteration 500 / 1600: loss 58.256400
iteration 600 / 1600: loss 50.520371
iteration 700 / 1600: loss 49.388951
iteration 800 / 1600: loss 48.190353
iteration 900 / 1600: loss 49.727617
iteration 1000 / 1600: loss 69.812725
iteration 1100 / 1600: loss 46.289338
iteration 1200 / 1600: loss 62.349763
iteration 1300 / 1600: loss 70.210827
iteration 1400 / 1600: loss 65.946359
iteration 1500 / 1600: loss 85.602400
iteration 0 / 1600: loss 504.053635
iteration 100 / 1600: loss 43.960169
iteration 200 / 1600: loss 48.934399
iteration 300 / 1600: loss 58.543076
iteration 400 / 1600: loss 77.888359
iteration 500 / 1600: loss 76.289567
iteration 600 / 1600: loss 56.567966
iteration 700 / 1600: loss 72.521855
iteration 800 / 1600: loss 67.012406
iteration 900 / 1600: loss 47.752688
iteration 1000 / 1600: loss 45.240314
iteration 1100 / 1600: loss 60.407286
iteration 1200 / 1600: loss 75.944764
iteration 1300 / 1600: loss 54.735258
iteration 1400 / 1600: loss 45.400015
iteration 1500 / 1600: loss 69.206268
iteration 0 / 1600: loss 597.323468
iteration 100 / 1600: loss 76.389958
iteration 200 / 1600: loss 54.863677
iteration 300 / 1600: loss 67.637297
iteration 400 / 1600: loss 54.480531
iteration 500 / 1600: loss 63.868945
iteration 600 / 1600: loss 81.945904
iteration 700 / 1600: loss 56.811417
iteration 800 / 1600: loss 59.331608
iteration 900 / 1600: loss 79.985412
iteration 1000 / 1600: loss 38.130718
iteration 1100 / 1600: loss 95.372689
iteration 1200 / 1600: loss 60.730795
iteration 1300 / 1600: loss 53.132119
iteration 1400 / 1600: loss 45.764033
iteration 1500 / 1600: loss 88.818195
iteration 0 / 1600: loss 703.607402
iteration 100 / 1600: loss 89.923413
iteration 200 / 1600: loss 84.666589
iteration 300 / 1600: loss 63.199613
iteration 400 / 1600: loss 86.758295
iteration 500 / 1600: loss 67.365669
iteration 600 / 1600: loss 67.913422
iteration 700 / 1600: loss 61.784822
iteration 800 / 1600: loss 65.547743
iteration 900 / 1600: loss 70.636050
iteration 1000 / 1600: loss 131.470401
iteration 1100 / 1600: loss 63.397759
iteration 1200 / 1600: loss 48.547083
iteration 1300 / 1600: loss 63.239399
iteration 1400 / 1600: loss 74.564878
iteration 1500 / 1600: loss 73.415555
iteration 0 / 1600: loss 793.696295
iteration 100 / 1600: loss 94.543887
iteration 200 / 1600: loss 97.799155
iteration 300 / 1600: loss 89.967224
iteration 400 / 1600: loss 76.951500
iteration 500 / 1600: loss 66.505118
iteration 600 / 1600: loss 96.008640
iteration 700 / 1600: loss 82.382012
iteration 800 / 1600: loss 111.040278
iteration 900 / 1600: loss 88.760370
iteration 1000 / 1600: loss 75.000467
iteration 1100 / 1600: loss 104.742929
iteration 1200 / 1600: loss 87.357908
iteration 1300 / 1600: loss 94.646580
iteration 1400 / 1600: loss 90.374852
iteration 1500 / 1600: loss 70.971877
iteration 0 / 1600: loss 407.980660
iteration 100 / 1600: loss 235.916653
iteration 200 / 1600: loss 194.201798
iteration 200 / 1600: loss [illegible]
iteration 300 / 1600: loss 212.638677
iteration 400 / 1600: loss 182.491794
iteration 500 / 1600: loss 179.368747
iteration 600 / 1600: loss 195.602903
iteration 700 / 1600: loss 144.009615
iteration 800 / 1600: loss 99.563585
iteration 900 / 1600: loss 179.824108
iteration 1000 / 1600: loss 148.491581
iteration 1100 / 1600: loss 202.121628
iteration 1200 / 1600: loss 175.010310
iteration 1300 / 1600: loss 151.420483
iteration 1400 / 1600: loss 168.241103
iteration 1500 / 1600: loss 172.002407
iteration 0 / 1600: loss 502.913190
iteration 100 / 1600: loss 274.669141
iteration 200 / 1600: loss 195.273985
iteration 300 / 1600: loss 153.828493
iteration 400 / 1600: loss 191.272835
iteration 500 / 1600: loss 195.536805
iteration 600 / 1600: loss 219.368340
iteration 700 / 1600: loss 188.961563
iteration 800 / 1600: loss 147.584668
iteration 900 / 1600: loss 242.499088
iteration 1000 / 1600: loss 224.264415
iteration 1100 / 1600: loss 148.304011
iteration 1200 / 1600: loss 197.654777
iteration 1300 / 1600: loss 251.248286
iteration 1400 / 1600: loss 186.701563
iteration 1500 / 1600: loss 225.703595
iteration 0 / 1600: loss 602.416580
iteration 100 / 1600: loss 218.536230
iteration 200 / 1600: loss 272.227376
iteration 300 / 1600: loss 256.318927
iteration 400 / 1600: loss 205.972079
iteration 500 / 1600: loss 305.939837
iteration 600 / 1600: loss 322.034776
iteration 700 / 1600: loss 339.381031
iteration 800 / 1600: loss 260.634849
iteration 900 / 1600: loss 258.728049
iteration 1000 / 1600: loss 272.643081
iteration 1100 / 1600: loss 239.304546
iteration 1200 / 1600: loss 276.582645
iteration 1300 / 1600: loss 346.831876
iteration 1400 / 1600: loss 323.510209
iteration 1500 / 1600: loss 260.494181
iteration 0 / 1600: loss 680.241978
iteration 100 / 1600: loss 325.360839
iteration 200 / 1600: loss 324.245896
iteration 300 / 1600: loss 390.974722
iteration 400 / 1600: loss 287.558916
iteration 500 / 1600: loss 355.895475
iteration 600 / 1600: loss 398.136123
iteration 700 / 1600: loss 401.248775
iteration 800 / 1600: loss 325.505194
iteration 900 / 1600: loss 444.359912
iteration 1000 / 1600: loss 360.617108
iteration 1100 / 1600: loss 431.631409
iteration 1200 / 1600: loss 373.538761
iteration 1300 / 1600: loss 353.374128
iteration 1400 / 1600: loss 325.531067
iteration 1500 / 1600: loss 310.885105
iteration 0 / 1600: loss 790.376187
iteration 100 / 1600: loss 409.640102
iteration 200 / 1600: loss 490.448874
iteration 300 / 1600: loss 495.202984
iteration 400 / 1600: loss 479.213971
iteration 500 / 1600: loss 565.809302
iteration 600 / 1600: loss 551.557373
iteration 700 / 1600: loss 481.921439
iteration 800 / 1600: loss 445.822005
iteration 900 / 1600: loss 597.311689
iteration 1000 / 1600: loss 565.111693
iteration 1100 / 1600: loss 508.985320
iteration 1200 / 1600: loss 491.311612
iteration 1300 / 1600: loss 582.682967
iteration 1400 / 1600: loss 558.742317
iteration 1500 / 1600: loss 508.775503
iteration 1500 / 1600: loss [illegible]
iteration 0 / 1600: loss 408.562090
iteration 100 / 1600: loss 450.052184
iteration 200 / 1600: loss 366.310912
iteration 300 / 1600: loss 313.143052
iteration 400 / 1600: loss 349.160451
iteration 500 / 1600: loss 442.431342
iteration 600 / 1600: loss 400.999722
iteration 700 / 1600: loss 356.696412
iteration 800 / 1600: loss 424.049112
iteration 900 / 1600: loss 462.108814
iteration 1000 / 1600: loss 407.927830
iteration 1100 / 1600: loss 401.454546
iteration 1200 / 1600: loss 361.511713
iteration 1300 / 1600: loss 424.661944
iteration 1400 / 1600: loss 333.354558
iteration 1500 / 1600: loss 364.630595
iteration 0 / 1600: loss 502.445537
iteration 100 / 1600: loss 747.966008
iteration 200 / 1600: loss 627.389526
iteration 300 / 1600: loss 628.909647
iteration 400 / 1600: loss 621.147299
iteration 500 / 1600: loss 663.805894
iteration 600 / 1600: loss 553.157856
iteration 700 / 1600: loss 723.554393
iteration 800 / 1600: loss 618.631001
iteration 900 / 1600: loss 577.883847
iteration 1000 / 1600: loss 467.549848
iteration 1100 / 1600: loss 615.973846
iteration 1200 / 1600: loss 598.255905
iteration 1300 / 1600: loss 599.833429
iteration 1400 / 1600: loss 632.006654
iteration 1500 / 1600: loss 727.458912
iteration 0 / 1600: loss 602.443125
iteration 100 / 1600: loss 1262.824004
iteration 200 / 1600: loss 1414.389851
iteration 300 / 1600: loss 1183.862828
iteration 400 / 1600: loss 1321.575268
iteration 500 / 1600: loss 1223.004862
iteration 600 / 1600: loss 1171.074284
iteration 700 / 1600: loss 1173.574808
iteration 800 / 1600: loss 1128.134067
iteration 900 / 1600: loss 1172.045030
iteration 1000 / 1600: loss 1360.786491
iteration 1100 / 1600: loss 1281.131530
iteration 1200 / 1600: loss 1098.925459
iteration 1300 / 1600: loss 1004.083380
iteration 1400 / 1600: loss 1264.874933
iteration 1500 / 1600: loss 1201.918586
iteration 0 / 1600: loss 700.411197
iteration 100 / 1600: loss 2683.425714
iteration 200 / 1600: loss 2729.189306
iteration 300 / 1600: loss 2857.439574
iteration 400 / 1600: loss 2725.919284
iteration 500 / 1600: loss 2904.217218
iteration 600 / 1600: loss 3109.068843
iteration 700 / 1600: loss 2649.382576
iteration 800 / 1600: loss 3014.559541
iteration 900 / 1600: loss 3046.167634
iteration 1000 / 1600: loss 2678.637379
iteration 1100 / 1600: loss 3326.309448
iteration 1200 / 1600: loss 2870.791298
iteration 1300 / 1600: loss 3229.428196
iteration 1400 / 1600: loss 3633.816471
iteration 1500 / 1600: loss 3053.954516
iteration 0 / 1600: loss 786.904604
iteration 100 / 1600: loss 21875.810429
iteration 200 / 1600: loss 21645.511980
iteration 300 / 1600: loss 22791.665462
iteration 400 / 1600: loss 20575.231667
iteration 500 / 1600: loss 19062.840033
iteration 600 / 1600: loss 20943.709616
iteration 700 / 1600: loss 21858.776542
iteration 800 / 1600: loss 22419.070037
iteration 900 / 1600: loss 24046.178913
iteration 1000 / 1600: loss 20818.408909
iteration 1100 / 1600: loss 22588.363477
iteration 1200 / 1600: loss 21761.023021
iteration 1200 / 1600: loss 21701.025021
iteration 1300 / 1600: loss 22063.698600
iteration 1400 / 1600: loss 23626.758671
iteration 1500 / 1600: loss 20892.864501
iteration 0 / 1600: loss 405.708969
iteration 100 / 1600: loss 921.105729
iteration 200 / 1600: loss 1019.313620
iteration 300 / 1600: loss 926.372122
iteration 400 / 1600: loss 1252.336917
iteration 500 / 1600: loss 1128.945928
iteration 600 / 1600: loss 1023.060182
iteration 700 / 1600: loss 940.006980
iteration 800 / 1600: loss 945.646431
iteration 900 / 1600: loss 1010.637654
iteration 1000 / 1600: loss 1007.512814
iteration 1100 / 1600: loss 893.463627
iteration 1200 / 1600: loss 1013.013132
iteration 1300 / 1600: loss 859.469458
iteration 1400 / 1600: loss 1098.131237
iteration 1500 / 1600: loss 932.301108
iteration 0 / 1600: loss 509.192668
iteration 100 / 1600: loss 2652.436417
iteration 200 / 1600: loss 2163.409714
iteration 300 / 1600: loss 2790.761784
iteration 400 / 1600: loss 2628.291146
iteration 500 / 1600: loss 2572.346067
iteration 600 / 1600: loss 2358.331795
iteration 700 / 1600: loss 2456.153984
iteration 800 / 1600: loss 2694.058280
iteration 900 / 1600: loss 2815.729802
iteration 1000 / 1600: loss 2914.610887
iteration 1100 / 1600: loss 2160.916653
iteration 1200 / 1600: loss 2604.694967
iteration 1300 / 1600: loss 2269.840191
iteration 1400 / 1600: loss 3067.164129
iteration 1500 / 1600: loss 3025.821567
iteration 0 / 1600: loss 597.377067
iteration 100 / 1600: loss 29779.258663
iteration 200 / 1600: loss 28675.096580
iteration 300 / 1600: loss 29639.737890
iteration 400 / 1600: loss 29502.930703
iteration 500 / 1600: loss 27302.791413
iteration 600 / 1600: loss 28098.653415
iteration 700 / 1600: loss 28491.833633
iteration 800 / 1600: loss 30501.990548
iteration 900 / 1600: loss 26447.745738
iteration 1000 / 1600: loss 29166.654552
iteration 1100 / 1600: loss 27444.570677
iteration 1200 / 1600: loss 26616.580735
iteration 1300 / 1600: loss 27953.807022
iteration 1400 / 1600: loss 29487.481584
iteration 1500 / 1600: loss 28366.232009
iteration 0 / 1600: loss 690.675243
iteration 100 / 1600: loss 898876352169534668 8.000000
iteration 200 / 1600: loss 7593179053425098101859919104835584.000000
iteration 300 / 1600: loss 641427093618763363611382503676254799479651172 3520.000000
iteration 400 / 1600: loss 5418398717235936728729996912276334045111915030864710449724653568.000000
iteration 500 / 1600: loss
457714445663777725750326083067840122904293985412754366945801801373282446029 6192.000000
iteration 600 / 1600: loss
386650235064598569937682173772403786025146001384047496981746415715668276493329479578773199257 6.000

iteration 700 / 1600: loss
32661937085841069773561863854660831509851532739952727461781136389921788601356003898483602789275094 2
606656.000000
iteration 800 / 1600: loss
27590882856212592752346839484135141852179163094590952367186070974969706237053069540596914961372111 7
188772996899275800576.000000
iteration 900 / 1600: loss
23307154587449451076213911884268651850258970212798996625231699585882255958266388199917503945626212 4
453094804059402365871177287203815424.000000
iteration 1000 / 1600: loss
19688512969817830514409302061772571427458381666168175853418546425760523243219691789409489733844460 4
786217541942826923761157699392958257144431772696576.000000
iteration 1100 / 1600: loss
16631697426138058672035772016091937905136697244533475325152077123559766423961878687306671286771497 7
35352046164450350730528312958771276854680418874284155192723111936 0.000000
iteration 1200 / 1600: loss
iteration 1200 / 1600: loss
140494794959198335450660306175303713476522366541491861460285812356860600497381221887252544398953409
666045464102304708285226299781061426065845656173163896357722597480467917550125056.000000
iteration 1300 / 1600: loss
118681737076373711782718396838542943013495124602603989984718379248224966133675303481785595590508895
383179227293192081933191692327286686219693684617599709717050404003379212286157257728446479466496.00
 
iteration 1400 / 1600: loss
100255349100698604272668746828549361765213034365506737954085682785199148791024455815614743849025380
383285288268497631027156046697499589273541332938860131216820938472858713132053268116560719403928568
31816960.000000
iteration 1500 / 1600: loss
846898206152381276166700817901467836249082738711808982509244584461274639577704796284353324401814343
378542931216848559584781281456108087021646865645812057418804923915049717774560171072625416176656879
8375114038768637575168.000000
iteration 0 / 1600: loss 800.284195
iteration 100 / 1600: loss 399623613528847684339137947353382649856.000000
iteration 200 / 1600: loss
66054582762297485612517849043734100144089796219366202771900347317961621504.000000
iteration 300 / 1600: loss
109182935046610799758259035126121160000245701764653011974716116403685874709968158890780780758585037
1864064.000000
iteration 400 / 1600: loss
180470647256841508514627165527199335309332535455882852580836161301067052850469623710029877369618490
713660839708706450129175342035456848035840.000000
iteration 500 / 1600: loss
298303526163673340793332185982952134410385745396299134013338437119515666164319395014402619294636797
53458409819463108451004977915550769820833560977570878791647856960496893689856.000000
iteration 600 / 1600: loss
493071837854274772042491675262716398373394244174267258725399717899232652176714634474305276776537887
235378937795993717462823244657653050448293816732585570791105924687525539062714507937649687031168025
143480320.000000
iteration 700 / 1600: loss
815008258238278633139551042467236002596242285472085622620468858345902750010523937880671322084166633
359079003889680484912923399809957204054673371221437700930518146735636691853261923554050278420453475
09992360225449019662388376166623380477313024.000000
iteration 800 / 1600: loss
134714337749888991589992821872524737416652989734652946794494333019252373655570984633798835712474576
052468358295701250559123211690676878227262197244046361434566854205295631750804668765352468874189649
22800266651418708295315669903857526128376779057237752910743576571695585654996992.000000
```

```
/Users/colby/Documents/CS291 Independent Study/assignment1/cs231n/classifiers/linear_svm.py:99: Ru
ntimeWarning: overflow encountered in double_scalars
  loss += 0.5 * reg * np.sum(np.square(W))
/anaconda3/lib/python3.7/site-packages/numpy/core/fromnumeric.py:83: RuntimeWarning: overflow enco
untered in reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
/Users/colby/Documents/CS291 Independent Study/assignment1/cs231n/classifiers/linear_svm.py:99: Ru
ntimeWarning: overflow encountered in square
  loss += 0.5 * reg * np.sum(np.square(W))
```

```
iteration 900 / 1600: loss inf
iteration 1000 / 1600: loss inf
iteration 1100 / 1600: loss inf
iteration 1200 / 1600: loss inf
iteration 1300 / 1600: loss inf
iteration 1400 / 1600: loss inf
iteration 1500 / 1600: loss inf
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.379449 val accuracy: 0.389000
lr 1.000000e-07 reg 3.125000e+04 train accuracy: 0.375490 val accuracy: 0.369000
lr 1.000000e-07 reg 3.750000e+04 train accuracy: 0.377776 val accuracy: 0.389000
lr 1.000000e-07 reg 4.375000e+04 train accuracy: 0.371000 val accuracy: 0.375000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.373306 val accuracy: 0.376000
lr 1.257500e-05 reg 2.500000e+04 train accuracy: 0.204755 val accuracy: 0.221000
lr 1.257500e-05 reg 3.125000e+04 train accuracy: 0.181735 val accuracy: 0.182000
lr 1.257500e-05 reg 3.750000e+04 train accuracy: 0.198102 val accuracy: 0.185000
lr 1.257500e-05 reg 4.375000e+04 train accuracy: 0.190082 val accuracy: 0.202000
lr 1.257500e-05 reg 5.000000e+04 train accuracy: 0.187224 val accuracy: 0.186000
lr 2.505000e-05 reg 2.500000e+04 train accuracy: 0.175653 val accuracy: 0.155000
lr 2.505000e-05 reg 3.125000e+04 train accuracy: 0.169571 val accuracy: 0.179000
lr 2.505000e-05 reg 3.750000e+04 train accuracy: 0.159429 val accuracy: 0.154000
lr 2.505000e-05 reg 4.375000e+04 train accuracy: 0.185490 val accuracy: 0.175000
lr 2.505000e-05 reg 5.000000e+04 train accuracy: 0.143388 val accuracy: 0.149000
lr 3.752500e-05 reg 2.500000e+04 train accuracy: 0.131020 val accuracy: 0.127000
lr 3.752500e-05 reg 3.125000e+04 train accuracy: 0.163306 val accuracy: 0.175000
lr 3.752500e-05 reg 3.750000e+04 train accuracy: 0.134898 val accuracy: 0.122000
lr 3.752500e-05 reg 4.375000e+04 train accuracy: 0.102694 val accuracy: 0.080000
lr 3.752500e-05 reg 5.000000e+04 train accuracy: 0.068714 val accuracy: 0.054000
lr 5.000000e-05 reg 2.500000e+04 train accuracy: 0.160592 val accuracy: 0.178000
lr 5.000000e-05 reg 3.125000e+04 train accuracy: 0.107408 val accuracy: 0.111000
lr 5.000000e-05 reg 3.750000e+04 train accuracy: 0.110245 val accuracy: 0.128000
lr 5.000000e-05 reg 4.375000e+04 train accuracy: 0.088898 val accuracy: 0.084000
lr 5.000000e-05 reg 5.000000e+04 train accuracy: 0.080163 val accuracy: 0.092000
best validation accuracy achieved during cross-validation: 0.389000
```

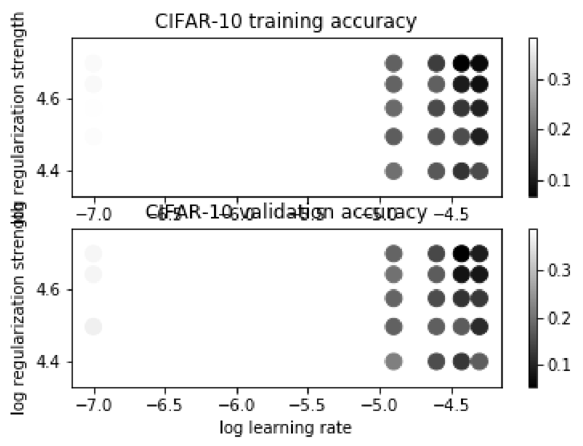
In [16]:

```
# 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: 0.080163
validation accuracy: 0.092000
```

In [19]:

```
# 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()
```



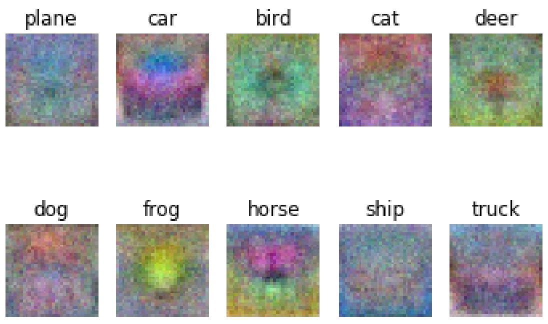
In [20]:

```
# 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: 0.379000
```

In [21]:

```python
# 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: The multiclass SVM templates resemble similar characteristics to how these object classes look like. The car template is mostly red, which shows a bias in the car dataset, whilst the horse SVM weights show a horse possibly facing both sides. The ship and plane both have dominant blue, for sky or ocean water. These trained weights for the SVM classes overall gave me a more intuitive understanding of SGD and linear classification&optimisation, as these optiomal template from the random ones assigned indicate pattern recognition from the CIFAR 10 image data. The pixel weights are the features that lead to the best prediction scores and viewing these as a human, these templates closely resemble how objects in these classes look, tho unclear for objects like animals (dog, dear and frog), although general shape outline is very clear. This would not be very easy to distinguish for a multiple animals dataset, with just this method.

In [ ]:

```

```

In [ ]:

```

```