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/) 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 [2]:
```
# 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 modules;
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

CIFAR-10 Data Loading and Preprocessing

In [3]:
```
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs682/datasets/cifar-10-batches-py'

# Cleaning up variables to prevent loading data multiple times (which may cause 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 [4]:
```python
# 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 [5]:
```python
# 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)
```

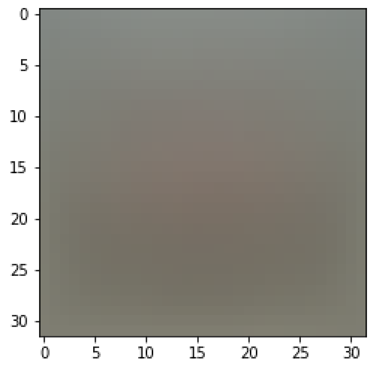
```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

In [6]:
```python
# 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 [7]: # 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 [8]: # 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 [9]: # 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 [10]: # 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: 9.427342
```

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 [11]:
```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 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: 5.643907 analytic: 5.643907, relative error: 4.990735e-12
numerical: -31.707712 analytic: -31.707712, relative error: 2.414137e-12
numerical: -15.094566 analytic: -15.094566, relative error: 6.928050e-12
numerical: 6.590847 analytic: 6.590847, relative error: 6.172296e-11
numerical: -0.195251 analytic: -0.195251, relative error: 4.938329e-11
numerical: 23.630563 analytic: 23.630563, relative error: 1.323223e-11
numerical: -12.854727 analytic: -12.854727, relative error: 8.826560e-12
numerical: -12.317041 analytic: -12.317041, relative error: 2.239764e-11
numerical: 5.725626 analytic: 5.725626, relative error: 2.955116e-11
numerical: 13.582976 analytic: 13.582976, relative error: 2.564645e-11
numerical: 34.546979 analytic: 34.546979, relative error: 1.409610e-11
numerical: -9.059835 analytic: -9.059835, relative error: 9.190069e-12
numerical: 12.961169 analytic: 12.961169, relative error: 4.125498e-11
numerical: 13.339584 analytic: 13.339584, relative error: 4.944858e-12
numerical: 1.247810 analytic: 1.247810, relative error: 2.075525e-11
numerical: 9.514842 analytic: 9.514842, relative error: 2.153143e-11
numerical: -19.383491 analytic: -19.383491, relative error: 4.305561e-12
numerical: -1.687203 analytic: -1.687203, relative error: 5.879986e-11
numerical: 15.154097 analytic: 15.154097, relative error: 2.513574e-11
numerical: -22.494673 analytic: -22.494673, relative error: 1.254444e-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: While calculating the analytic gradient, if the loss has a small positive value (lets call this f(x) = e). While calculating the numerical gradient, if we use a value of h > e then f(x) - h would cross over the 0 (the location of the kink in SVM loss) and introduce a 0 contribution towards the gradient. Hence, there will be a discrepancy during gradcheck. The same can occur for a small negative value of loss when f(x)-h crosses over 0 and incurs an error in calculation. We could easily get around this by considering fewer datapoints while validating the gradients thereby reducing the chances of our datapoints lying around the kink. Increasing the margin might lead to increase in the frequency of this happening as there would be more changes of a data point not belonging to the correct class being very close to the margin.

In [12]:
```
# Next implement the function svm_loss_vectorized; for now only compute the loss;
# we will implement the gradient in a moment.
tic = time.time()
loss_naive, grad_naive = svm_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Naive loss: %e computed in %fs' % (loss_naive, toc - tic))

from cs682.classifiers.linear_svm import svm_loss_vectorized
tic = time.time()
loss_vectorized, _ = svm_loss_vectorized(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Vectorized loss: %e computed in %fs' % (loss_vectorized, toc - tic))

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

```
Naive loss: 9.427342e+00 computed in 0.142856s
Vectorized loss: 9.427342e+00 computed in 0.009900s
difference: 0.000000
```

In [13]:
```
# 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 0.131190s
Vectorized loss and gradient: computed in 0.003910s
difference: 0.000000
```

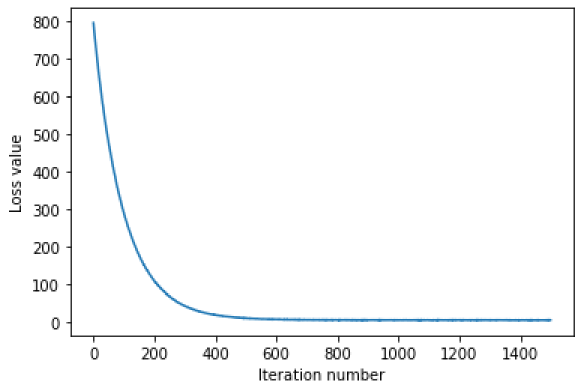
Stochastic Gradient Descent

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 [14]:
```
# 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 794.312538
iteration 100 / 1500: loss 287.904655
iteration 200 / 1500: loss 108.227064
iteration 300 / 1500: loss 42.795420
iteration 400 / 1500: loss 19.172778
iteration 500 / 1500: loss 10.514195
iteration 600 / 1500: loss 7.156890
iteration 700 / 1500: loss 5.543028
iteration 800 / 1500: loss 5.214884
iteration 900 / 1500: loss 5.433855
iteration 1000 / 1500: loss 5.367930
iteration 1100 / 1500: loss 5.485583
iteration 1200 / 1500: loss 5.316967
iteration 1300 / 1500: loss 5.635412
iteration 1400 / 1500: loss 5.149256
That took 2.742559s
```

In [15]:
```
# 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()
```



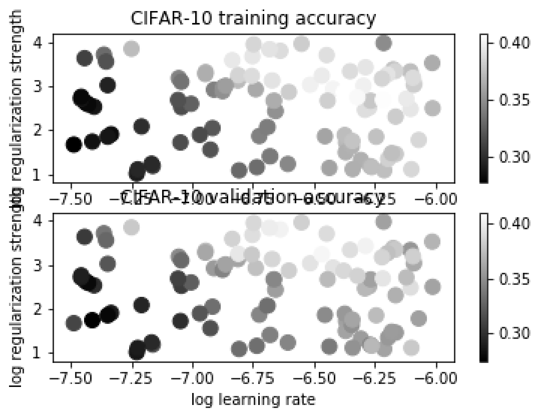
In [16]:
```
# Write the LinearSVM.predict function and evaluate the performance on both the
# training and validation set
y_train_pred = svm.predict(X_train)
print('training accuracy: %f' % (np.mean(y_train == y_train_pred), ))
y_val_pred = svm.predict(X_val)
print('validation accuracy: %f' % (np.mean(y_val == y_val_pred), ))
```

```
training accuracy: 0.371959
validation accuracy: 0.374000
```

In [45]:
```
# 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 = [4e-7,5e-7, 6e-7, 7e-7, 7.5e-7, 8e-7, 8.5e-7, 9e-7, 9.5e-7, 1e-6]
# regularization_strengths = [1e1,5e1,1e2,5e2,1e3,5e3, 1e4, 2e4]s

# 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.                                      #
################################################################################
# Setting the seed to ensure consistent behavior across runs with same params
np.random.seed(123)
num_trials = 100
for t in range(num_trials):
#   print ('learning rate: %f, regularization strength: %f' %(lr, rs))
    lr = 10 ** np.random.uniform (-7.5, -6)
    rs = 10 ** np.random.uniform (1,4)
    svm = LinearSVM()
    loss_hist = svm.train(X_train, y_train, learning_rate=lr, reg=rs,
                  num_iters=2000, batch_size=400, verbose=False)
    y_train_pred = svm.predict(X_train)
    train_acc = np.mean(y_train == y_train_pred)
#   print('training accuracy: %f' % (train_acc))
    y_val_pred = svm.predict(X_val)
    val_acc = np.mean(y_val == y_val_pred)
#   print('validation accuracy: %f' % (val_acc))
    results [(lr, rs)] = (train_acc, val_acc)
    if (val_acc > best_val):
        best_val = val_acc
        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 3.251255e-08 reg 4.668452e+01 train accuracy: 0.277184 val accuracy: 0.311000
lr 3.478788e-08 reg 5.137424e+02 train accuracy: 0.289204 val accuracy: 0.289000
lr 3.489476e-08 reg 5.721079e+02 train accuracy: 0.281510 val accuracy: 0.291000
lr 3.601438e-08 reg 4.329056e+03 train accuracy: 0.310653 val accuracy: 0.305000
lr 3.634694e-08 reg 4.207594e+02 train accuracy: 0.281388 val accuracy: 0.291000
lr 3.727577e-08 reg 3.857726e+02 train accuracy: 0.281898 val accuracy: 0.279000
lr 3.874409e-08 reg 5.461223e+01 train accuracy: 0.286000 val accuracy: 0.274000
lr 3.941344e-08 reg 3.390967e+02 train accuracy: 0.288122 val accuracy: 0.297000
lr 4.325203e-08 reg 5.212938e+03 train accuracy: 0.331776 val accuracy: 0.337000
lr 4.416854e-08 reg 3.617168e+03 train accuracy: 0.318265 val accuracy: 0.329000
lr 4.480725e-08 reg 7.048268e+01 train accuracy: 0.288469 val accuracy: 0.278000
lr 4.482619e-08 reg 1.051723e+03 train accuracy: 0.294755 val accuracy: 0.320000
lr 4.631187e-08 reg 8.015565e+01 train accuracy: 0.283694 val accuracy: 0.288000
lr 5.619378e-08 reg 7.101248e+03 train accuracy: 0.372755 val accuracy: 0.381000
lr 5.905107e-08 reg 1.022389e+01 train accuracy: 0.291959 val accuracy: 0.286000
lr 5.964594e-08 reg 1.278224e+01 train accuracy: 0.297367 val accuracy: 0.289000
lr 6.182823e-08 reg 1.207073e+02 train accuracy: 0.300020 val accuracy: 0.294000
lr 6.800233e-08 reg 1.708376e+01 train accuracy: 0.298224 val accuracy: 0.296000
lr 6.874855e-08 reg 1.522251e+01 train accuracy: 0.306102 val accuracy: 0.315000
lr 8.744384e-08 reg 4.855520e+02 train accuracy: 0.318551 val accuracy: 0.309000
lr 8.800857e-08 reg 1.525428e+03 train accuracy: 0.343735 val accuracy: 0.347000
lr 8.944845e-08 reg 5.222336e+01 train accuracy: 0.315980 val accuracy: 0.298000
lr 9.013629e-08 reg 1.255389e+03 train accuracy: 0.335571 val accuracy: 0.330000
lr 9.020229e-08 reg 3.310228e+02 train accuracy: 0.316633 val accuracy: 0.306000
lr 9.893984e-08 reg 3.961656e+02 train accuracy: 0.326286 val accuracy: 0.324000
lr 1.075437e-07 reg 7.783641e+01 train accuracy: 0.315020 val accuracy: 0.319000
lr 1.132648e-07 reg 1.888104e+03 train accuracy: 0.365939 val accuracy: 0.363000
lr 1.135399e-07 reg 2.088986e+03 train accuracy: 0.368571 val accuracy: 0.374000
lr 1.188303e-07 reg 3.583559e+01 train accuracy: 0.319694 val accuracy: 0.317000
lr 1.212182e-07 reg 1.109603e+02 train accuracy: 0.321612 val accuracy: 0.333000
lr 1.229384e-07 reg 8.309898e+02 train accuracy: 0.342286 val accuracy: 0.356000
lr 1.334196e-07 reg 7.648317e+02 train accuracy: 0.350347 val accuracy: 0.344000
lr 1.341470e-07 reg 9.542580e+02 train accuracy: 0.352714 val accuracy: 0.369000
lr 1.372996e-07 reg 1.022020e+03 train accuracy: 0.357592 val accuracy: 0.361000
lr 1.461940e-07 reg 3.897038e+03 train accuracy: 0.395306 val accuracy: 0.386000
lr 1.541637e-07 reg 1.339302e+03 train accuracy: 0.371571 val accuracy: 0.361000
lr 1.548008e-07 reg 1.771345e+03 train accuracy: 0.384980 val accuracy: 0.397000
lr 1.560149e-07 reg 1.205858e+01 train accuracy: 0.330571 val accuracy: 0.332000
lr 1.759830e-07 reg 1.707677e+03 train accuracy: 0.389102 val accuracy: 0.381000
lr 1.769711e-07 reg 4.778015e+03 train accuracy: 0.394163 val accuracy: 0.392000
lr 1.790853e-07 reg 8.932212e+03 train accuracy: 0.385735 val accuracy: 0.389000
lr 1.826721e-07 reg 8.478981e+02 train accuracy: 0.362918 val accuracy: 0.376000
lr 1.845477e-07 reg 1.426261e+01 train accuracy: 0.332592 val accuracy: 0.333000
lr 1.889344e-07 reg 7.298913e+01 train accuracy: 0.342408 val accuracy: 0.345000
lr 2.035732e-07 reg 1.172258e+02 train accuracy: 0.340898 val accuracy: 0.333000
lr 2.078649e-07 reg 6.924201e+03 train accuracy: 0.392347 val accuracy: 0.400000
lr 2.086973e-07 reg 2.527523e+01 train accuracy: 0.339265 val accuracy: 0.342000
lr 2.129682e-07 reg 1.342818e+03 train accuracy: 0.392449 val accuracy: 0.388000
lr 2.165040e-07 reg 4.664767e+02 train accuracy: 0.365020 val accuracy: 0.346000
lr 2.196915e-07 reg 2.704577e+02 train accuracy: 0.350980 val accuracy: 0.358000
lr 2.295802e-07 reg 6.368351e+03 train accuracy: 0.389061 val accuracy: 0.403000
lr 2.465536e-07 reg 1.692118e+01 train accuracy: 0.346612 val accuracy: 0.345000
lr 2.490223e-07 reg 7.355208e+02 train accuracy: 0.381633 val accuracy: 0.384000
lr 2.807149e-07 reg 9.202233e+03 train accuracy: 0.381367 val accuracy: 0.385000
lr 3.046898e-07 reg 1.079173e+03 train accuracy: 0.398939 val accuracy: 0.395000
lr 3.402921e-07 reg 1.736155e+03 train accuracy: 0.396714 val accuracy: 0.405000
lr 3.505127e-07 reg 7.218019e+01 train accuracy: 0.360551 val accuracy: 0.348000
lr 3.563784e-07 reg 5.091546e+03 train accuracy: 0.390347 val accuracy: 0.391000
lr 3.594539e-07 reg 4.665094e+02 train accuracy: 0.382388 val accuracy: 0.373000
lr 3.654896e-07 reg 1.373534e+01 train accuracy: 0.360286 val accuracy: 0.339000
lr 3.789462e-07 reg 8.655728e+02 train accuracy: 0.401429 val accuracy: 0.395000
lr 4.187680e-07 reg 7.297520e+02 train accuracy: 0.407020 val accuracy: 0.389000
lr 4.229739e-07 reg 7.705630e+01 train accuracy: 0.373306 val accuracy: 0.355000
lr 4.268465e-07 reg 3.109065e+01 train accuracy: 0.366776 val accuracy: 0.361000
lr 4.281898e-07 reg 5.203413e+01 train accuracy: 0.371082 val accuracy: 0.360000
lr 4.451041e-07 reg 2.180340e+03 train accuracy: 0.388796 val accuracy: 0.392000
lr 4.520011e-07 reg 4.252222e+01 train accuracy: 0.367939 val accuracy: 0.351000
lr 4.549080e-07 reg 5.627173e+01 train accuracy: 0.376939 val accuracy: 0.362000
lr 4.558419e-07 reg 1.286003e+01 train accuracy: 0.366327 val accuracy: 0.352000
lr 4.772910e-07 reg 5.604356e+02 train accuracy: 0.403633 val accuracy: 0.386000
lr 4.928970e-07 reg 2.082830e+01 train accuracy: 0.366469 val accuracy: 0.357000
lr 5.050506e-07 reg 1.025022e+03 train accuracy: 0.399020 val accuracy: 0.410000
lr 5.122127e-07 reg 2.883737e+02 train accuracy: 0.389939 val accuracy: 0.372000
lr 5.153676e-07 reg 2.686289e+03 train accuracy: 0.393490 val accuracy: 0.403000
lr 5.330233e-07 reg 3.240582e+02 train accuracy: 0.389796 val accuracy: 0.369000
lr 5.393970e-07 reg 1.505624e+01 train accuracy: 0.369857 val accuracy: 0.344000
lr 5.408464e-07 reg 4.649674e+02 train accuracy: 0.378143 val accuracy: 0.360000
lr 5.462577e-07 reg 1.431611e+01 train accuracy: 0.373878 val accuracy: 0.372000
lr 5.701436e-07 reg 7.038480e+01 train accuracy: 0.375551 val accuracy: 0.374000
lr 6.023699e-07 reg 1.480942e+03 train accuracy: 0.384837 val accuracy: 0.387000
lr 6.070402e-07 reg 4.885039e+02 train accuracy: 0.405612 val accuracy: 0.396000
lr 6.117351e-07 reg 9.572465e+03 train accuracy: 0.348286 val accuracy: 0.360000
lr 6.149036e-07 reg 1.213045e+01 train accuracy: 0.374082 val accuracy: 0.361000
lr 6.341515e-07 reg 1.837654e+03 train accuracy: 0.392469 val accuracy: 0.376000
lr 6.382017e-07 reg 2.197040e+01 train accuracy: 0.383551 val accuracy: 0.355000
lr 6.603845e-07 reg 2.987210e+01 train accuracy: 0.374204 val accuracy: 0.357000
lr 6.701202e-07 reg 2.410460e+03 train accuracy: 0.374612 val accuracy: 0.377000
lr 6.830692e-07 reg 2.209866e+03 train accuracy: 0.384857 val accuracy: 0.381000
lr 6.854474e-07 reg 2.020754e+02 train accuracy: 0.381102 val accuracy: 0.368000
lr 6.864593e-07 reg 2.847938e+01 train accuracy: 0.379449 val accuracy: 0.356000
lr 7.591890e-07 reg 5.169790e+02 train accuracy: 0.396102 val accuracy: 0.374000
lr 7.887476e-07 reg 2.162084e+03 train accuracy: 0.373735 val accuracy: 0.354000
lr 7.899653e-07 reg 3.022374e+02 train accuracy: 0.394020 val accuracy: 0.392000
lr 7.958802e-07 reg 1.284899e+01 train accuracy: 0.386388 val accuracy: 0.381000
lr 8.082795e-07 reg 1.553257e+03 train accuracy: 0.388510 val accuracy: 0.384000
lr 8.145460e-07 reg 1.077362e+03 train accuracy: 0.381163 val accuracy: 0.364000
lr 8.199476e-07 reg 1.129479e+03 train accuracy: 0.390531 val accuracy: 0.395000
lr 8.535624e-07 reg 5.961523e+01 train accuracy: 0.386653 val accuracy: 0.359000
lr 9.702025e-07 reg 3.358292e+03 train accuracy: 0.364816 val accuracy: 0.370000
lr 9.738339e-07 reg 3.141903e+02 train accuracy: 0.371510 val accuracy: 0.361000
best validation accuracy achieved during cross-validation: 0.410000
```

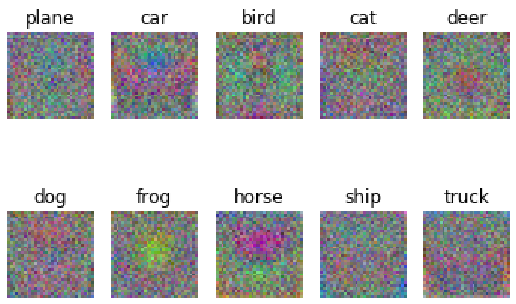
In [46]:
```python
# Visualize the cross-validation results
import math
x_scatter = [math.log10(x[0]) for x in results]
y_scatter = [math.log10(x[1]) for x in results]

# 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 [47]:
```python
# 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: 0.388000
```

In [48]:
```
# 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 weights look like a combination of various different classes with dominant colors and gradients of the correct class. This reflects the various tones in the images that the classifier is using to differentiate the correct class from all other classes, thereby trying to find tones that exist in one and not in others.

In [ ]:
```

```