Multiclass Support Vector Machine exercise

Complete and hand in the completed notebook (including the output) with your assignment submission. You will be submitting the homework as a zip file including all the parts on the Blackboard.

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

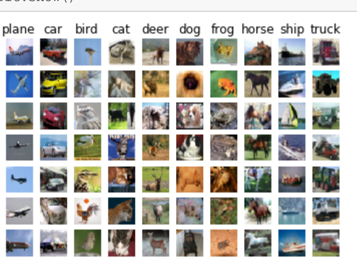
In [1]:
```
## Default modules
from __future__ import print_function
import random
import numpy as np
import matplotlib.pyplot as plt

## Custom modules
from ie590.data_utils import load_CIFAR10

# 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;

%load_ext autoreload
%autoreload 2
```

CIFAR-10 Data Loading and Preprocessing

In [2]:
```
# Load the raw CIFAR-10 data.
cifar10_dir = 'ie590/datasets/cifar-10-batches-py'
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)
```

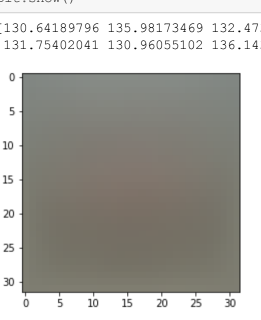
```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

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))

# 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 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
```

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 **ie590/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]:
```
# Evaluate the naive implementation of the loss we provided for you:
from ie590.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, 0.000004)
print('loss: %f' % (loss, ))
```

```
loss: 9.223893
```

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 dimensions, and
# compare them with your analytically computed gradient. The numbers should match
# almost exactly along all dimensions.
from ie590.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: 7.947813 analytic: 7.947813, relative error: 2.175071e-11
numerical: -9.409788 analytic: -9.409788, relative error: 6.969942e-11
numerical: -10.521413 analytic: -10.521413, relative error: 2.128826e-11
numerical: 8.810062 analytic: 8.810062, relative error: 6.696375e-12
numerical: 14.669548 analytic: 14.669548, relative error: 1.455253e-11
numerical: -6.361561 analytic: -6.361561, relative error: 3.288525e-12
numerical: -13.318344 analytic: -13.318344, relative error: 4.059987e-13
numerical: -55.304365 analytic: -55.304365, relative error: 5.833571e-13
numerical: -0.828118 analytic: -0.828118, relative error: 4.265885e-10
numerical: -4.290576 analytic: -4.290576, relative error: 7.937468e-12
numerical: -16.870974 analytic: -16.870974, relative error: 2.694021e-11
numerical: -1.226697 analytic: -1.226697, relative error: 3.313650e-10
numerical: -0.618606 analytic: -0.618606, relative error: 6.124806e-10
numerical: -0.796062 analytic: -0.796062, relative error: 5.749047e-10
numerical: -14.696856 analytic: -14.696856, relative error: 2.348677e-12
numerical: -5.343436 analytic: -5.343436, relative error: 2.555952e-11
numerical: -60.694161 analytic: -60.694161, relative error: 4.386238e-12
numerical: -7.763034 analytic: -7.763034, relative error: 4.359177e-11
numerical: 3.224499 analytic: 3.224499, relative error: 2.539276e-11
numerical: 9.688917 analytic: 9.688917, relative error: 2.017344e-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? *Hint: the SVM loss function is not strictly speaking differentiable*

Your Answer: *One reason could be the discontinuity of SVM loss function when $S_j-S_{y_i}+1 \sim 0$*, which leads to an ill-defined gradient at that point. Numerically, when the function hits that point, the numerical error could be very large, theseby leads to the possible fail of gradient check.

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, 0.000004)
toc = time.time()
print('Naive loss: %e computed in %fs' % (loss_naive, toc - tic))

from ie590.classifiers.linear_svm import svm_loss_vectorized
tic = time.time()
loss_vectorized, _ = svm_loss_vectorized(W, X_dev, y_dev, 0.000004)

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: 9.223893e+00 computed in 0.085121s
Vectorized loss: 9.223893e+00 computed in 0.006516s
difference: 0.000000
```

In [12]:
```
# 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, 0.000004)
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, 0.000004)
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 0.085052s
Vectorized loss and gradient: computed in 0.006338s
difference: 0.000000
```

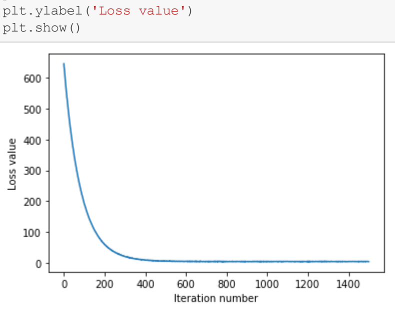
Stochastic Gradient Descent

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 ie590.classifiers import LinearSVM
svm = LinearSVM()
tic = time.time()
loss_hist = svm.train(X_train, y_train, learning_rate=1.5e-7, reg=2.0e4,
                      num_iters=1500, verbose=True)
toc = time.time()
print('That took %fs' % (toc - tic))
```

```
iteration 0 / 1500: loss 644.081611
iteration 100 / 1500: loss 193.641019
iteration 200 / 1500: loss 61.067365
iteration 300 / 1500: loss 21.638116
iteration 400 / 1500: loss 9.847789
iteration 500 / 1500: loss 7.007161
iteration 600 / 1500: loss 5.662183
iteration 700 / 1500: loss 5.467002
iteration 800 / 1500: loss 5.750516
iteration 900 / 1500: loss 5.217507
iteration 1000 / 1500: loss 4.659286
iteration 1100 / 1500: loss 5.972933
iteration 1200 / 1500: loss 5.225398
iteration 1300 / 1500: loss 5.707822
iteration 1400 / 1500: loss 5.591786
That took 6.049168s
```

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()
```



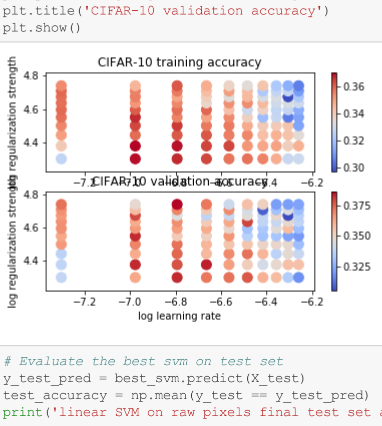
In [15]:
```
# Write the LinearSVM.predict function and evaluate the performance on both the
# training and validation set
y_train_pred = svm.predict(X_train)
print('training accuracy: %f' % (np.mean(y_train == y_train_pred), ))
y_val_pred = svm.predict(X_val)
print('validation accuracy: %f' % (np.mean(y_val == y_val_pred), ))
```

```
training accuracy: 0.370122
validation accuracy: 0.399000
```

In [29]:
```
# 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 = [1.5e-7, 6e-5]
regularization_strengths = [2e4, 5.5e4]

# results is dictionary mapping tuples of the form
# (learning_rate, regularization_strength) to tuples of the form
# (training_accuracy, validation_accuracy). The accuracy is simply the fraction
# of data points that are correctly classified.
results = {}
best_val = -1   # The highest validation accuracy that we have seen so far.
best_svm = None # The LinearSVM object that achieved the highest validation rate.

################################################################################
# TODO:                                                                        #
# Write code that chooses the best hyperparameters by tuning on the validation #
# set. For each combination of hyperparameters, train a linear SVM on the      #
# training set, compute its accuracy on the training and validation sets, and #
# store these numbers in the results dictionary. In addition, store the best   #
# validation accuracy in best_val and the LinearSVM object that achieves this  #
# accuracy in best_svm.                                                        #
#                                                                              #
# Hint: You should use a small value for num_iters as you develop your         #
# validation code so that the SVMs don't take much time to train; once you are #
# confident that your validation code works, you should rerun the validation   #
# code with a larger value for num_iters.                                      #
################################################################################
#                            START OF YOUR CODE                              #
##############################################################################
pass ## Write your code here
# define new lr and reg ranges
learning_rates = [0.5e-7, 5.5e-7]
regularization_strengths = [2e4, 5.5e4]
for lr in np.linspace(learning_rates[0], learning_rates[1], num = 10):
    for reg in np.linspace(regularization_strengths[0], regularization_strengths[1], num = 10):
        svm = LinearSVM()
        svm.train(X_train, y_train, lr, reg, num_iters=700, verbose=False)
        y_train_pred = svm.predict(X_train)
        train_accuracy = np.mean(y_train == y_train_pred)
        y_val_pred = svm.predict(X_val)
        val_accuracy = np.mean(y_val == y_val_pred)

        results[(lr, reg)] = (train_accuracy, val_accuracy)
        if val_accuracy > best_val:
            best_val = val_accuracy
            best_svm = svm

################################################################################
#                              END OF YOUR CODE                                #
################################################################################

# Print out results.
for lr, reg in sorted(results):
    train_accuracy, val_accuracy = results[(lr, reg)]
    print('lr %e reg %e train accuracy: %f val accuracy: %f' % (
                lr, reg, train_accuracy, val_accuracy))

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

```
lr 5.000000e-08 reg 2.000000e+04 train accuracy: 0.325796 val accuracy: 0.339000
lr 5.000000e-08 reg 2.388889e+04 train accuracy: 0.341082 val accuracy: 0.338000
lr 5.000000e-08 reg 2.777778e+04 train accuracy: 0.350510 val accuracy: 0.337000
lr 5.000000e-08 reg 3.166667e+04 train accuracy: 0.354857 val accuracy: 0.366000
lr 5.000000e-08 reg 3.555556e+04 train accuracy: 0.358122 val accuracy: 0.363000
lr 5.000000e-08 reg 3.944444e+04 train accuracy: 0.359388 val accuracy: 0.368000
lr 5.000000e-08 reg 4.333333e+04 train accuracy: 0.358531 val accuracy: 0.368000
lr 5.000000e-08 reg 4.722222e+04 train accuracy: 0.358551 val accuracy: 0.366000
lr 5.000000e-08 reg 5.111111e+04 train accuracy: 0.359490 val accuracy: 0.368000
lr 5.000000e-08 reg 5.500000e+04 train accuracy: 0.353776 val accuracy: 0.373000
lr 1.055556e-07 reg 2.000000e+04 train accuracy: 0.368918 val accuracy: 0.363000
lr 1.055556e-07 reg 2.388889e+04 train accuracy: 0.369755 val accuracy: 0.383000
lr 1.055556e-07 reg 2.777778e+04 train accuracy: 0.358653 val accuracy: 0.362000
lr 1.055556e-07 reg 3.166667e+04 train accuracy: 0.363796 val accuracy: 0.370000
lr 1.055556e-07 reg 3.555556e+04 train accuracy: 0.367061 val accuracy: 0.383000
lr 1.055556e-07 reg 3.944444e+04 train accuracy: 0.361122 val accuracy: 0.366000
lr 1.055556e-07 reg 4.333333e+04 train accuracy: 0.351959 val accuracy: 0.362000
lr 1.055556e-07 reg 4.722222e+04 train accuracy: 0.355857 val accuracy: 0.379000
lr 1.055556e-07 reg 5.111111e+04 train accuracy: 0.355551 val accuracy: 0.358000
lr 1.055556e-07 reg 5.500000e+04 train accuracy: 0.346020 val accuracy: 0.349000
lr 1.611111e-07 reg 2.000000e+04 train accuracy: 0.366939 val accuracy: 0.383000
lr 1.611111e-07 reg 2.388889e+04 train accuracy: 0.368694 val accuracy: 0.378000
lr 1.611111e-07 reg 2.777778e+04 train accuracy: 0.361204 val accuracy: 0.375000
lr 1.611111e-07 reg 3.166667e+04 train accuracy: 0.365367 val accuracy: 0.373000
lr 1.611111e-07 reg 3.555556e+04 train accuracy: 0.355347 val accuracy: 0.354000
lr 1.611111e-07 reg 3.944444e+04 train accuracy: 0.358429 val accuracy: 0.365000
lr 1.611111e-07 reg 4.333333e+04 train accuracy: 0.354551 val accuracy: 0.368000
lr 1.611111e-07 reg 4.722222e+04 train accuracy: 0.349408 val accuracy: 0.346000
lr 1.611111e-07 reg 5.111111e+04 train accuracy: 0.356408 val accuracy: 0.370000
lr 1.611111e-07 reg 5.500000e+04 train accuracy: 0.361490 val accuracy: 0.386000
lr 2.166667e-07 reg 2.000000e+04 train accuracy: 0.363816 val accuracy: 0.372000
lr 2.166667e-07 reg 2.388889e+04 train accuracy: 0.364327 val accuracy: 0.384000
lr 2.166667e-07 reg 2.777778e+04 train accuracy: 0.361735 val accuracy: 0.359000
lr 2.166667e-07 reg 3.166667e+04 train accuracy: 0.356796 val accuracy: 0.360000
lr 2.166667e-07 reg 3.555556e+04 train accuracy: 0.349673 val accuracy: 0.352000
lr 2.166667e-07 reg 3.944444e+04 train accuracy: 0.359571 val accuracy: 0.362000
lr 2.166667e-07 reg 4.333333e+04 train accuracy: 0.336612 val accuracy: 0.354000
lr 2.166667e-07 reg 4.722222e+04 train accuracy: 0.339898 val accuracy: 0.338000
lr 2.166667e-07 reg 5.111111e+04 train accuracy: 0.343878 val accuracy: 0.355000
lr 2.166667e-07 reg 5.500000e+04 train accuracy: 0.347612 val accuracy: 0.372000
lr 2.722222e-07 reg 2.000000e+04 train accuracy: 0.357612 val accuracy: 0.373000
lr 2.722222e-07 reg 2.388889e+04 train accuracy: 0.363898 val accuracy: 0.365000
lr 2.722222e-07 reg 2.777778e+04 train accuracy: 0.356102 val accuracy: 0.369000
lr 2.722222e-07 reg 3.166667e+04 train accuracy: 0.356510 val accuracy: 0.356000
lr 2.722222e-07 reg 3.555556e+04 train accuracy: 0.354286 val accuracy: 0.367000
lr 2.722222e-07 reg 3.944444e+04 train accuracy: 0.343327 val accuracy: 0.350000
lr 2.722222e-07 reg 4.333333e+04 train accuracy: 0.353224 val accuracy: 0.363000
lr 2.722222e-07 reg 4.722222e+04 train accuracy: 0.333939 val accuracy: 0.340000
lr 2.722222e-07 reg 5.111111e+04 train accuracy: 0.350878 val accuracy: 0.369000
lr 2.722222e-07 reg 5.500000e+04 train accuracy: 0.337878 val accuracy: 0.341000
lr 3.277778e-07 reg 2.000000e+04 train accuracy: 0.357776 val accuracy: 0.377000
lr 3.277778e-07 reg 2.388889e+04 train accuracy: 0.351898 val accuracy: 0.353000
lr 3.277778e-07 reg 2.777778e+04 train accuracy: 0.354510 val accuracy: 0.373000
lr 3.277778e-07 reg 3.166667e+04 train accuracy: 0.348102 val accuracy: 0.356000
lr 3.277778e-07 reg 3.555556e+04 train accuracy: 0.338857 val accuracy: 0.336000
lr 3.277778e-07 reg 3.944444e+04 train accuracy: 0.328469 val accuracy: 0.331000
lr 3.277778e-07 reg 4.333333e+04 train accuracy: 0.353469 val accuracy: 0.380000
lr 3.277778e-07 reg 4.722222e+04 train accuracy: 0.344633 val accuracy: 0.347000
lr 3.277778e-07 reg 5.111111e+04 train accuracy: 0.344776 val accuracy: 0.344000
lr 3.277778e-07 reg 5.500000e+04 train accuracy: 0.334531 val accuracy: 0.357000
lr 3.833333e-07 reg 2.000000e+04 train accuracy: 0.355327 val accuracy: 0.361000
lr 3.833333e-07 reg 2.388889e+04 train accuracy: 0.348939 val accuracy: 0.343000
lr 3.833333e-07 reg 2.777778e+04 train accuracy: 0.343653 val accuracy: 0.360000
lr 3.833333e-07 reg 3.166667e+04 train accuracy: 0.341122 val accuracy: 0.351000
lr 3.833333e-07 reg 3.555556e+04 train accuracy: 0.318061 val accuracy: 0.331000
lr 3.833333e-07 reg 3.944444e+04 train accuracy: 0.338245 val accuracy: 0.333000
lr 3.833333e-07 reg 4.333333e+04 train accuracy: 0.345041 val accuracy: 0.338000
lr 3.833333e-07 reg 4.722222e+04 train accuracy: 0.331653 val accuracy: 0.339000
lr 3.833333e-07 reg 5.111111e+04 train accuracy: 0.321633 val accuracy: 0.310000
lr 3.833333e-07 reg 5.500000e+04 train accuracy: 0.342204 val accuracy: 0.352000
lr 4.388889e-07 reg 2.000000e+04 train accuracy: 0.347490 val accuracy: 0.358000
lr 4.388889e-07 reg 2.388889e+04 train accuracy: 0.349469 val accuracy: 0.349000
lr 4.388889e-07 reg 2.777778e+04 train accuracy: 0.346633 val accuracy: 0.337000
lr 4.388889e-07 reg 3.166667e+04 train accuracy: 0.340796 val accuracy: 0.350000
lr 4.388889e-07 reg 3.555556e+04 train accuracy: 0.327510 val accuracy: 0.332000
lr 4.388889e-07 reg 3.944444e+04 train accuracy: 0.326510 val accuracy: 0.347000
lr 4.388889e-07 reg 4.333333e+04 train accuracy: 0.329327 val accuracy: 0.340000
lr 4.388889e-07 reg 4.722222e+04 train accuracy: 0.332000 val accuracy: 0.350000
lr 4.388889e-07 reg 5.111111e+04 train accuracy: 0.326592 val accuracy: 0.331000
lr 4.388889e-07 reg 5.500000e+04 train accuracy: 0.321939 val accuracy: 0.322000
lr 4.944444e-07 reg 2.000000e+04 train accuracy: 0.328020 val accuracy: 0.335000
lr 4.944444e-07 reg 2.388889e+04 train accuracy: 0.339347 val accuracy: 0.357000
lr 4.944444e-07 reg 2.777778e+04 train accuracy: 0.342796 val accuracy: 0.348000
lr 4.944444e-07 reg 3.166667e+04 train accuracy: 0.342592 val accuracy: 0.349000
lr 4.944444e-07 reg 3.555556e+04 train accuracy: 0.304837 val accuracy: 0.323000
lr 4.944444e-07 reg 3.944444e+04 train accuracy: 0.326612 val accuracy: 0.358000
lr 4.944444e-07 reg 4.333333e+04 train accuracy: 0.331612 val accuracy: 0.349000
lr 4.944444e-07 reg 4.722222e+04 train accuracy: 0.296612 val accuracy: 0.306000
lr 4.944444e-07 reg 5.111111e+04 train accuracy: 0.327388 val accuracy: 0.345000
lr 4.944444e-07 reg 5.500000e+04 train accuracy: 0.315082 val accuracy: 0.322000
lr 5.500000e-07 reg 2.000000e+04 train accuracy: 0.329061 val accuracy: 0.319000
lr 5.500000e-07 reg 2.388889e+04 train accuracy: 0.346796 val accuracy: 0.357000
lr 5.500000e-07 reg 2.777778e+04 train accuracy: 0.334857 val accuracy: 0.340000
lr 5.500000e-07 reg 3.166667e+04 train accuracy: 0.312592 val accuracy: 0.330000
lr 5.500000e-07 reg 3.555556e+04 train accuracy: 0.312122 val accuracy: 0.321000
lr 5.500000e-07 reg 3.944444e+04 train accuracy: 0.314327 val accuracy: 0.330000
lr 5.500000e-07 reg 4.333333e+04 train accuracy: 0.329327 val accuracy: 0.338000
lr 5.500000e-07 reg 4.722222e+04 train accuracy: 0.321245 val accuracy: 0.336000
lr 5.500000e-07 reg 5.111111e+04 train accuracy: 0.310898 val accuracy: 0.330000
lr 5.500000e-07 reg 5.500000e+04 train accuracy: 0.305265 val accuracy: 0.323000
best validation accuracy achieved during cross-validation: 0.386000
```

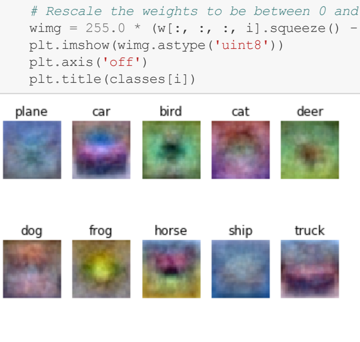
In [30]:
```
# Visualize the cross-validation results
import math
x_scatter = [math.log10(x[0]) for x in results]
y_scatter = [math.log10(x[1]) for x in results]

# 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, cmap=plt.cm.coolwarm)
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, cmap=plt.cm.coolwarm)
plt.colorbar()
plt.xlabel('log learning rate')
plt.ylabel('log regularization strength')
plt.title('CIFAR-10 validation accuracy')
plt.show()
```



In [31]:
```
# 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: 0.359000
```

In [32]:
```
# 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 SVM here is only one layer, using the matrix W to capture one pattern of that class. So the W would show the unique pattern, where the car class looks like a red car, the dog class looks like a brown dog with a big head and big ear, the cat class has a fat body, the frog class has a blue side and bright belly, the horse has heavy body and thin legs, etc.*

Inline question 3:

How does changing the regularization parameter affect your results/classifier?

Your answer: *In this test, changing regularization parameter can slightly change the accuracy. But the change may not have single best value because we can see two peaks in accuracy in above figure.*

Inline question 4:

How does changing the learning rate affect the training time? Should we choose a learning rate that only minimizes the training time? Why / why not? **Your answer:** *Typically, a larger learning rate makes the training faster because the W can approach the minimum at a larger step size. We should not choose learning rate that only minimizes the training time because a too large learning rate may miss the fine features of the shape, thus fail to find the best minimum. This is also demonstrated in above figure where a very small lr could reduce the overall accuracy.*

Inline question 5:

What is the advantage of stochastic gradient descent over the regular gradient descent?