Softmax 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 (http://vision.stanford.edu/teaching/cs175/assignments.html)](http://vision.stanford.edu/teaching/cs175/assignments.html) on the course website.

This exercise is analogous to the SVM exercise. You will:

- implement a fully-vectorized **loss function** for the Softmax classifier
- implement the fully-vectorized expression for its **analytic gradient**
- **check your implementation** with 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 [43]:
```
from __future__ import print_function

import random
import numpy as np
from cs175.data_utils import load_CIFAR10
import matplotlib.pyplot as plt


%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'

# for auto-reloading extenrnal modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipyt
hon
%load_ext autoreload
%autoreload 2
```

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

In [44]:
```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000, n
um_dev=500):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    it for the linear classifier. These are the same steps as we used for the
    SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = 'cs175/datasets/cifar-10-batches-py'
    X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

    # subsample the data
    mask = list(range(num_training, num_training + num_validation))
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = list(range(num_training))
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = list(range(num_test))
    X_test = X_test[mask]
    y_test = y_test[mask]
    mask = np.random.choice(num_training, num_dev, replace=False)
    X_dev = X_train[mask]
    y_dev = y_train[mask]

    # 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))

    # Normalize the data: subtract the mean image
    mean_image = np.mean(X_train, axis = 0)
    X_train -= mean_image
    X_val -= mean_image
    X_test -= mean_image
    X_dev -= mean_image

    # add bias dimension and transform into columns
    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))])

    return X_train, y_train, X_val, y_val, X_test, y_test, X_dev, y_dev


# Invoke the above function to get our data.
X_train, y_train, X_val, y_val, X_test, y_test, X_dev, y_dev = get_CIFAR10_dat
a()
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)
```

```
print('dev data shape: ', X_dev.shape)
print('dev labels shape: ', y_dev.shape)
```

```
Train data shape:  (49000L, 3073L)
Train labels shape:  (49000L,)
Validation data shape:  (1000L, 3073L)
Validation labels shape:  (1000L,)
Test data shape:  (1000L, 3073L)
Test labels shape:  (1000L,)
dev data shape:  (500L, 3073L)
dev labels shape:  (500L,)
```

Softmax Classifier

Your code for this section will all be written inside **cs175/classifiers/softmax.py**.

In [45]:
```
# First implement the naive softmax loss function with nested loops.
# Open the file cs175/classifiers/softmax.py and implement the
# softmax_loss_naive function.

from cs175.classifiers.softmax import softmax_loss_naive
import time

# Generate a random softmax weight matrix and use it to compute the loss.
W = np.random.randn(3073, 10) * 0.0001
loss, grad = softmax_loss_naive(W, X_dev, y_dev, 0.0)

# As a rough sanity check, our loss should be something close to -log(0.1).
print('loss: %f' % loss)
print('sanity check: %f' % (-np.log(0.1)))
```

```
loss: 2.359843
sanity check: 2.302585
```

Inline Question 1:

Why do we expect our loss to be close to -log(0.1)? Explain briefly.**

Your answer: This is to be expected as $h(x) \; = \frac{a}{\sum_1^c a}$ where c in our case is 10, this essentially means the loss function of $-log(h_y(x))$ is close to $-log(0.1)$ as $h_y(x)$ nears $\frac{1}{10}$

In [46]:
```
# Complete the implementation of softmax_loss_naive and implement a (naive)
# version of the gradient that uses nested loops.
loss, grad = softmax_loss_naive(W, X_dev, y_dev, 0.0)

# As we did for the SVM, use numeric gradient checking as a debugging tool.
# The numeric gradient should be close to the analytic gradient.
from cs175.gradient_check import grad_check_sparse
f = lambda w: softmax_loss_naive(w, X_dev, y_dev, 0.0)[0]
grad_numerical = grad_check_sparse(f, W, grad, 10)

# similar to SVM case, do another gradient check with regularization
loss, grad = softmax_loss_naive(W, X_dev, y_dev, 5e1)
f = lambda w: softmax_loss_naive(w, X_dev, y_dev, 5e1)[0]
grad_numerical = grad_check_sparse(f, W, grad, 10)
```

```
numerical: 1.252828 analytic: 1.252828, relative error: 1.656416e-08
numerical: 1.458513 analytic: 1.458513, relative error: 1.330685e-08
numerical: -2.724058 analytic: -2.724058, relative error: 1.394652e-08
numerical: 2.386529 analytic: 2.386529, relative error: 1.244620e-08
numerical: -0.966360 analytic: -0.966360, relative error: 8.150500e-09
numerical: 0.426791 analytic: 0.426791, relative error: 4.089810e-08
numerical: -0.924963 analytic: -0.924964, relative error: 9.586758e-08
numerical: -1.307433 analytic: -1.307433, relative error: 9.606610e-09
numerical: 0.143448 analytic: 0.143448, relative error: 5.627124e-07
numerical: 1.217917 analytic: 1.217917, relative error: 4.109891e-08
numerical: 0.977352 analytic: 0.977381, relative error: 1.460902e-05
numerical: -4.424606 analytic: -4.424723, relative error: 1.328006e-05
numerical: -0.067250 analytic: -0.067314, relative error: 4.760699e-04
numerical: 1.762786 analytic: 1.762774, relative error: 3.429167e-06
numerical: -0.613788 analytic: -0.613867, relative error: 6.435099e-05
numerical: -1.778204 analytic: -1.778203, relative error: 3.436324e-07
numerical: -4.569534 analytic: -4.569656, relative error: 1.339551e-05
numerical: -1.472622 analytic: -1.472602, relative error: 6.967391e-06
numerical: 0.695901 analytic: 0.695981, relative error: 5.714115e-05
numerical: 2.037849 analytic: 2.037903, relative error: 1.316992e-05
```

In [47]:
```
from cs175.classifiers.softmax import softmax_loss_vectorized
tic = time.time()
loss_vectorized, grad_vectorized = softmax_loss_vectorized(W, X_dev, y_dev, 0.
000005)
toc = time.time()
print('vectorized loss: %e computed in %fs' % (loss_vectorized, toc - tic))
```

```
vectorized loss: 2.359843e+00 computed in 0.007000s
```

In [48]:
```
# Now that we have a naive implementation of the softmax loss function and its
 gradient,
# implement a vectorized version in softmax_loss_vectorized.
# The two versions should compute the same results, but the vectorized version
 should be
# much faster.
tic = time.time()
loss_naive, grad_naive = softmax_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('naive loss: %e computed in %fs' % (loss_naive, toc - tic))

from cs175.classifiers.softmax import softmax_loss_vectorized
tic = time.time()
loss_vectorized, grad_vectorized = softmax_loss_vectorized(W, X_dev, y_dev, 0.
000005)
toc = time.time()
print('vectorized loss: %e computed in %fs' % (loss_vectorized, toc - tic))

# As we did for the SVM, we use the Frobenius norm to compare the two versions
# of the gradient.
grad_difference = np.linalg.norm(grad_naive - grad_vectorized, ord='fro')
print('Loss difference: %f' % np.abs(loss_naive - loss_vectorized))
print('Gradient difference: %f' % grad_difference)
```

```
naive loss: 2.359843e+00 computed in 0.082000s
vectorized loss: 2.359843e+00 computed in 0.006000s
Loss difference: 0.000000
Gradient difference: 321.570616
```

In [52]:

```
# 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 over 0.35 on the validation set.
from cs175.classifiers import Softmax
results = {}
best_val = -1
best_softmax = None
learning_rates = [1e-7, 8e-7]
regularization_strengths = [2.5e4, 5e4]

################################################################################
##
# TODO:
 #
# Use the validation set to set the learning rate and regularization strength.
 #
# This should be identical to the validation that you did for the SVM; save
 #
# the best trained softmax classifer in best_softmax.
 #
################################################################################
##

for learn in learning_rates:
    for reg in regularization_strengths:
        softmax = Softmax()
        softmax.train(X_train, y_train, learning_rate=learn, reg=reg,
                      num_iters=2000)
        p_train = softmax.predict(X_train)
        p_val   = softmax.predict(X_val)

        train_acc = np.mean(p_train == y_train)
        val_acc   = np.mean(p_val == y_val)

        if val_acc > best_val:
            best_val = val_acc
            best_softmax = softmax

        results[(learn, reg)] = train_acc, val_acc
################################################################################
##
#                              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_v
al)
```

```
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.079714 val accuracy: 0.080
000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.102408 val accuracy: 0.094
000
lr 8.000000e-07 reg 2.500000e+04 train accuracy: 0.100122 val accuracy: 0.111
000
lr 8.000000e-07 reg 5.000000e+04 train accuracy: 0.096061 val accuracy: 0.090
000
best validation accuracy achieved during cross-validation: 0.111000
```

In [50]:

```
# evaluate on test set
# Evaluate the best softmax on test set
y_test_pred = best_softmax.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('softmax on raw pixels final test set accuracy: %f' % (test_accuracy, ))
```

```
softmax on raw pixels final test set accuracy: 0.144000
```

In [51]:
```
# Visualize the learned weights for each class
w = best_softmax.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', 'shi
p', '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])
```