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/cs231n/assignments.html)](http://vision.stanford.edu/teaching/cs231n/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 [1]:
```
import random
import numpy as np
from hmwk5_2.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

from __future__ import print_function

%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
%load_ext autoreload
%autoreload 2
```

In [2]:
```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000, n
    """
    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 = 'hmwk5_2/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


# Cleaning up variables to prevent loading data multiple times (which may caus
try:
   del X_train, y_train
   del X_test, y_test
   print('Clear previously loaded data.')
except:
   pass

# 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
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)
```

Softmax Classifier

Your code for this section will all be written inside **hmwk5_2/classifiers/softmax.py**.

In [4]:
```
# First implement the naive softmax loss function with nested loops.
# Open the file cs231n/classifiers/softmax.py and implement the
# softmax_loss_naive function.

from hmwk5_2.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: 2.372611
sanity check: 2.302585
```

Inline Question 1:

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

Your answer: * Because we use a random weight matrix W,

Each class to have same expected scores s_j for each picture or we expect each class to be picked randomly

As we have 10 classes,

$s_j == s_k$

and

for all j,k in 0,...,9 {

$e^{s_j} == e^{s_k}$

}

Thus the expecation of the term inside log becomes

$e^{s_k}/sum(e^{s_j}$ for j in range(10)) $== e^{s_j}/sum(10 * e^{s_j}) == e^{s_j}/10 * e^{s_j} == 1/10.$*

In [5]:
```
# 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 hmwk5_2.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: 2.905540 analytic: 2.905540, relative error: 2.030850e-08
numerical: -0.626421 analytic: -0.626421, relative error: 5.418711e-09
numerical: 0.679245 analytic: 0.679245, relative error: 3.200919e-08
numerical: -5.140725 analytic: -5.140725, relative error: 4.507416e-09
numerical: 4.178988 analytic: 4.178988, relative error: 1.539717e-08
numerical: 0.219912 analytic: 0.219912, relative error: 1.287380e-07
numerical: -0.158448 analytic: -0.158448, relative error: 1.129315e-07
numerical: -0.002162 analytic: -0.002162, relative error: 1.271332e-05
numerical: 2.990328 analytic: 2.990328, relative error: 1.173248e-08
numerical: 2.800795 analytic: 2.800795, relative error: 1.554212e-08
numerical: -0.750818 analytic: -0.750818, relative error: 4.228010e-08
numerical: -1.765167 analytic: -1.765167, relative error: 2.565882e-08
numerical: -1.024704 analytic: -1.024704, relative error: 9.646090e-09
numerical: -1.306675 analytic: -1.306675, relative error: 4.064074e-08
numerical: 0.376995 analytic: 0.376995, relative error: 7.080839e-08
numerical: -1.882359 analytic: -1.882359, relative error: 1.028506e-08
numerical: 1.889889 analytic: 1.889889, relative error: 1.891304e-08
numerical: -2.526498 analytic: -2.526499, relative error: 1.304408e-08
numerical: 2.591603 analytic: 2.591603, relative error: 3.653577e-09
numerical: 0.373286 analytic: 0.373286, relative error: 5.092874e-08
```

In [6]:
```
# Now that we have a naive implementation of the softmax loss function and its
# implement a vectorized version in softmax_loss_vectorized.
# The two versions should compute the same results, but the vectorized version
# 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 hmwk5_2.classifiers.softmax import softmax_loss_vectorized
tic = time.time()
loss_vectorized, grad_vectorized = softmax_loss_vectorized(W, X_dev, y_dev, 0.
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: 2.372611e+00 computed in 0.097880s
vectorized loss: 2.372611e+00 computed in 0.007375s
Loss difference: 0.000000
Gradient difference: 0.000000
```

In [7]:

```
# 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 hmwk5_2.classifiers import Softmax
results = {}
best_val = -1
best_softmax = None
learning_rates = [1e-7, 5e-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 i in range(128):

    reg = np.random.uniform(regularization_strengths[0], regularization_streng

    learning_rate = np.random.uniform(learning_rates[0], learning_rates[1])

    softmax = Softmax()

    loss = softmax.train(X_train, y_train, learning_rate=learning_rate,reg=reg

    train_prediction = softmax.predict(X_train)

    train_acc = np.mean(train_prediction == y_train)

    val_prediction   = softmax.predict(X_val)

    val_acc = np.mean(val_prediction == y_val)

    if best_val < val_acc:
        best_val = val_acc
        best_softmax = softmax

    results[learning_rate, 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
```

```
lr 1.009938e-07 reg 3.845963e+04 train accuracy: 0.234306 val accuracy: 0.256
000
lr 1.033997e-07 reg 3.042589e+04 train accuracy: 0.217673 val accuracy: 0.238
000
lr 1.075325e-07 reg 4.863788e+04 train accuracy: 0.259633 val accuracy: 0.264
000
lr 1.208711e-07 reg 3.484924e+04 train accuracy: 0.246163 val accuracy: 0.263
000
lr 1.350039e-07 reg 2.850439e+04 train accuracy: 0.248469 val accuracy: 0.248
000
```

In [12]:
```
# 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 final test set accuracy: %f' % (test_accuracy, ))
```

```
softmax final test set accuracy: 0.357000
```

Inline Question - *True or False*

It's possible to add a new datapoint to a training set that would leave the SVM loss unchanged, but this is not the case with the Softmax classifier loss.

Your answer: True

Your explanation:

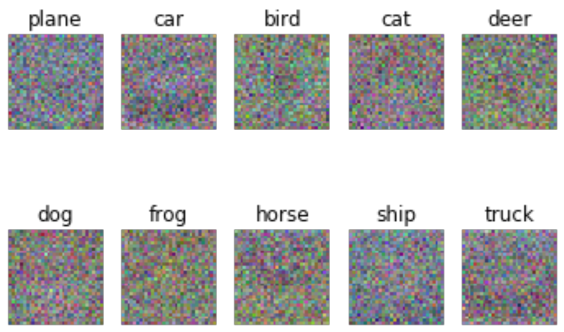
In the SVM if the new data point has a score that is out of the margin range from the correct class score the loss wouldn't change but in the Softmax loss if the score of the new added datapoint be close to +infinity it will adversely affect the loss, but definitely the loss of Softmax will change.

In [11]:
```
# 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
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])
```



In [ ]: