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]:

```
from __future__ import print_function

import random
import numpy as np
from cs231n.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-ipython
%load_ext autoreload
%autoreload 2
```

In [2]:

```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000, num_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 = 'cs231n/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 cause memory issue)
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_data()
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:  (49000, 3073)
Train labels shape:  (49000,)
Validation data shape:  (1000, 3073)
Validation labels shape:  (1000,)
Test data shape:  (1000, 3073)
Test labels shape:  (1000,)
dev data shape:  (500, 3073)
dev labels shape:  (500,)
```

Softmax Classifier

Your code for this section will all be written inside **cs231n/classifiers/softmax.py**.

In [3]:

```
# 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 cs231n.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: 2.439783
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 when the ininitial W is randomly selected, the scores should be approximately the same, so that the loss is about -log(1/C), where C is the number of classes.*

In [4]:

```
# 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 cs231n.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: 2.588460 analytic: 2.588460, relative error: 4.491622e-08
numerical: -1.988616 analytic: -1.988616, relative error: 4.715352e-09
numerical: 1.396795 analytic: 1.396794, relative error: 3.652356e-08
numerical: 3.544632 analytic: 3.544632, relative error: 1.863504e-08
numerical: 1.072708 analytic: 1.072708, relative error: 5.016998e-09
numerical: 0.302906 analytic: 0.302906, relative error: 1.862514e-07
numerical: -2.578414 analytic: -2.578414, relative error: 2.304479e-08
numerical: -0.155734 analytic: -0.155734, relative error: 1.095533e-07
numerical: 0.764819 analytic: 0.764819, relative error: 4.014913e-09
numerical: -0.653797 analytic: -0.653797, relative error: 1.211519e-09
numerical: -1.117606 analytic: -1.117606, relative error: 4.387970e-08
numerical: 1.400587 analytic: 1.400587, relative error: 3.961031e-08
numerical: 2.365878 analytic: 2.365878, relative error: 8.350434e-09
numerical: 2.903318 analytic: 2.903318, relative error: 9.046451e-09
numerical: -1.993702 analytic: -1.993702, relative error: 3.485217e-08
numerical: 0.886856 analytic: 0.886856, relative error: 6.893794e-10
numerical: -2.238476 analytic: -2.238476, relative error: 5.845339e-09
numerical: -0.436448 analytic: -0.436448, relative error: 5.903415e-08
numerical: -2.442069 analytic: -2.442069, relative error: 2.513662e-09
numerical: 2.004773 analytic: 2.004773, relative error: 2.474076e-08
```

In [5]:

```
# 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 cs231n.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: 2.439783e+00 computed in 0.086769s
vectorized loss: 2.439783e+00 computed in 0.004987s
Loss difference: 0.000000
Gradient difference: 0.000000
```

In [6]:

```
# 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 cs231n.classifiers import Softmax
results = {}
best_val = -1
best_softmax = None
learning_rates = [3e-8, 1e-7, 3e-7, 1e-6]
regularization_strengths = [2.5e3, 5e3, 1.25e4, 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 lr in learning_rates:
    for reg in regularization_strengths:
        softmax = Softmax()
        softmax.train(X_train, y_train, learning_rate=lr, reg=reg, num_iters=1500)

        # compute accuracy
        y_train_pred = softmax.predict(X_train)
        y_val_pred = softmax.predict(X_val)
        train_accuracy = np.mean(y_train == y_train_pred)
        val_accuracy = np.mean(y_val == y_val_pred)

        # save results
        results[(lr, reg)] = (train_accuracy, val_accuracy)

        # update best accuracy and classifier
        if(val_accuracy > best_val):
            best_val = val_accuracy
            best_softmax = softmax
################################################################################
#                              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 3.000000e-08 reg 2.500000e+03 train accuracy: 0.190408 val accuracy: 0.204000
lr 3.000000e-08 reg 5.000000e+03 train accuracy: 0.208918 val accuracy: 0.216000
lr 3.000000e-08 reg 1.250000e+04 train accuracy: 0.255531 val accuracy: 0.263000
lr 3.000000e-08 reg 2.500000e+04 train accuracy: 0.298204 val accuracy: 0.308000
lr 3.000000e-08 reg 5.000000e+04 train accuracy: 0.306592 val accuracy: 0.324000
lr 1.000000e-07 reg 2.500000e+03 train accuracy: 0.291041 val accuracy: 0.275000
lr 1.000000e-07 reg 5.000000e+03 train accuracy: 0.330551 val accuracy: 0.347000
lr 1.000000e-07 reg 1.250000e+04 train accuracy: 0.350510 val accuracy: 0.366000
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.323061 val accuracy: 0.336000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.301041 val accuracy: 0.327000
lr 3.000000e-07 reg 2.500000e+03 train accuracy: 0.383306 val accuracy: 0.404000
lr 3.000000e-07 reg 5.000000e+03 train accuracy: 0.371857 val accuracy: 0.390000
lr 3.000000e-07 reg 1.250000e+04 train accuracy: 0.353837 val accuracy: 0.369000
lr 3.000000e-07 reg 2.500000e+04 train accuracy: 0.323347 val accuracy: 0.341000
lr 3.000000e-07 reg 5.000000e+04 train accuracy: 0.301306 val accuracy: 0.319000
lr 1.000000e-06 reg 2.500000e+03 train accuracy: 0.382612 val accuracy: 0.393000
lr 1.000000e-06 reg 5.000000e+03 train accuracy: 0.369592 val accuracy: 0.365000
lr 1.000000e-06 reg 1.250000e+04 train accuracy: 0.346490 val accuracy: 0.356000
lr 1.000000e-06 reg 2.500000e+04 train accuracy: 0.332224 val accuracy: 0.342000
lr 1.000000e-06 reg 5.000000e+04 train accuracy: 0.299510 val accuracy: 0.314000
best validation accuracy achieved during cross-validation: 0.404000
```

In [7]:

```
# 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: 0.379000
```

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: When the new score is out of the margin range of the correct class, the SVM loss is unchanged. But the Softmax loss is always changed when the new score is a real number.

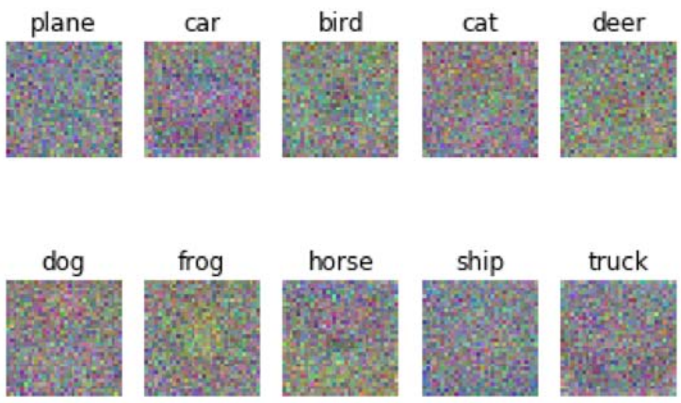
In [8]:

```
# 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', '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])
```


plane
car
bird
cat
deer
dog
frog
horse
ship
truck