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 cs231n.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
hon
%load_ext autoreload
%autoreload 2
```

In [2]:
```
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 = '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 caus
e 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_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:  (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: 2.321239
sanity check: 2.302585
```

Inline Question 1:

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

Your answer: *we havent trained the network yet, so the score for one class/ 10 class should be 0.1. And our loss = -log(scores)*

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: -3.096165 analytic: -3.096165, relative error: 1.754217e-08
numerical: 0.679032 analytic: 0.679032, relative error: 1.410851e-08
numerical: 0.671098 analytic: 0.671098, relative error: 6.494369e-09
numerical: -1.704792 analytic: -1.704792, relative error: 1.168830e-08
numerical: -0.588981 analytic: -0.588981, relative error: 6.732529e-08
numerical: 0.433609 analytic: 0.433609, relative error: 6.301351e-08
numerical: -0.373927 analytic: -0.373926, relative error: 1.524976e-07
numerical: -0.634650 analytic: -0.634650, relative error: 1.930650e-08
numerical: 0.234695 analytic: 0.234695, relative error: 9.369205e-08
numerical: 0.231109 analytic: 0.231109, relative error: 4.535916e-07
numerical: -2.348198 analytic: -2.348198, relative error: 2.581794e-08
numerical: -0.268808 analytic: -0.268808, relative error: 9.955429e-08
numerical: 0.442188 analytic: 0.442188, relative error: 4.803573e-08
numerical: 0.325792 analytic: 0.325792, relative error: 2.643950e-08
numerical: -1.062069 analytic: -1.062069, relative error: 7.476666e-08
numerical: -1.993377 analytic: -1.993377, relative error: 2.301985e-08
numerical: 1.884687 analytic: 1.884687, relative error: 3.293437e-08
numerical: -0.151403 analytic: -0.151403, relative error: 1.240941e-07
numerical: -0.233695 analytic: -0.233695, relative error: 8.297509e-08
numerical: -0.937028 analytic: -0.937028, relative error: 1.110804e-08
```

In [21]:
```
# 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: 2.321239e+00 computed in 0.153648s
vectorized loss: 2.321239e+00 computed in 0.015630s
Loss difference: 0.000000
Gradient difference: 0.000000
```

In [18]:
```
# 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 = [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.
 #
################################################################################
##
# Can divide the learning rate and regularization to smaller steps, but my
#computer is too weak. Dont want to train too much

for lr in learning_rates:
    for reg in regularization_strengths:
        softmax= Softmax()
        # Traing will call the loss, grad  which calculated above
        loss_hist = softmax.train(X_train, y_train,lr, reg,
                                  num_iters=500, verbose=False)
        # Predict functions are the same between SVM and Softmax
        y_train_pred = softmax.predict(X_train)
        train_accuracy=np.mean(y_train == y_train_pred)

        y_val_pred = softmax.predict(X_val)
        val_accuracy= np.mean(y_val == y_val_pred)

        results[(lr,reg)]=(train_accuracy,val_accuracy)
        if best_val< val_accuracy:
            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_v
al)
```

```
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.261939 val accuracy: 0.271
000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.306327 val accuracy: 0.311
000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.351020 val accuracy: 0.361
000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.324510 val accuracy: 0.332
000
best validation accuracy achieved during cross-validation: 0.361000
```

In [19]:

```
# 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: 0.362000
```

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: It is possiple

The new data is seperate from correct class bigger than margin -> not effect to the loss of SVM, but it is still effect to the loss of Softmax:

In [20]:
```python
# 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])
```