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 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 [3]:
```
from __future__ import print_function
import random
import numpy as np
from cs682.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 [4]:
```
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 = 'cs682/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 **cs682/classifiers/softmax.py**.

In [5]:
```
# First implement the naive softmax loss function with nested loops.
# Open the file cs682/classifiers/softmax.py and implement the
# softmax_loss_naive function.

from cs682.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: 2.329227
sanity check: 2.302585
```

Inline Question 1:

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

Your answer:

The loss should be around -log(0.1 because we have 10 classes and assuming the scores are all approximately the same, the loss should be around -log(e^score / (10 * e^score)) = -log(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 cs682.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: -1.544645 analytic: -1.544645, relative error: 5.320434e-08
numerical: -2.178610 analytic: -2.178610, relative error: 5.554907e-09
numerical: 0.361697 analytic: 0.361697, relative error: 1.125100e-07
numerical: 0.076178 analytic: 0.076178, relative error: 5.267031e-07
numerical: -0.248585 analytic: -0.248585, relative error: 2.649104e-08
numerical: 0.309712 analytic: 0.309712, relative error: 5.420240e-08
numerical: 1.630478 analytic: 1.630478, relative error: 1.205564e-08
numerical: -0.452599 analytic: -0.452599, relative error: 7.546752e-08
numerical: -0.276927 analytic: -0.276928, relative error: 2.554157e-07
numerical: -3.078943 analytic: -3.078943, relative error: 1.107484e-08
numerical: -1.695526 analytic: -1.695526, relative error: 2.552773e-08
numerical: -6.392450 analytic: -6.392450, relative error: 9.196914e-09
numerical: 0.545001 analytic: 0.545001, relative error: 1.411039e-07
numerical: 2.092425 analytic: 2.092425, relative error: 2.684974e-08
numerical: 0.603232 analytic: 0.603232, relative error: 8.809838e-08
numerical: -2.125793 analytic: -2.125793, relative error: 1.717946e-09
numerical: 0.652094 analytic: 0.652094, relative error: 9.618587e-08
numerical: -6.011416 analytic: -6.011416, relative error: 7.344141e-11
numerical: -3.414886 analytic: -3.414886, relative error: 2.819845e-09
numerical: 1.816323 analytic: 1.816322, relative error: 5.299007e-08
```

In [6]:
```
# 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 cs682.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: 2.418885e+00 computed in 0.148879s
vectorized loss: 2.418885e+00 computed in 0.005713s
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 cs682.classifiers import Softmax
results = {}
best_val = -1
best_softmax = None
learning_rates = [5e-8, 1e-7, 5e-7]
regularization_strengths = [5e3, 1.5e4, 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:
        print("Testing LR=", lr, " reg=", reg)
        softmax = Softmax()
        softmax.train(X_train, y_train, learning_rate=lr, reg=reg,
                      num_iters=6000, verbose=False)

        y_train_pred = softmax.predict(X_train)
        tr_acc = np.mean(y_train == y_train_pred)
        print('training accuracy: %f' % tr_acc)

        y_val_pred = softmax.predict(X_val)
        val_acc = np.mean(y_val == y_val_pred)
        print('validation accuracy: %f' % val_acc)

        results[(lr, reg)] = tr_acc, val_acc, softmax
################################################################################
#                              END OF YOUR CODE                                #
################################################################################

# Print out results.
for lr, reg in sorted(results):
    train_accuracy, val_accuracy, softmax = results[(lr, reg)]
    if val_accuracy > best_val:
        best_val = val_accuracy
        best_softmax = softmax
    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)
```

```
Testing LR= 5e-08  reg= 5000.0
training accuracy: 0.374224
validation accuracy: 0.379000
Testing LR= 5e-08  reg= 15000.0
training accuracy: 0.346388
validation accuracy: 0.365000
Testing LR= 5e-08  reg= 25000.0
training accuracy: 0.332714
validation accuracy: 0.347000
Testing LR= 5e-08  reg= 50000.0
training accuracy: 0.314714
validation accuracy: 0.332000
Testing LR= 1e-07  reg= 5000.0
training accuracy: 0.377020
validation accuracy: 0.386000
Testing LR= 1e-07  reg= 15000.0
training accuracy: 0.342898
validation accuracy: 0.360000
Testing LR= 1e-07  reg= 25000.0
training accuracy: 0.330020
validation accuracy: 0.346000
Testing LR= 1e-07  reg= 50000.0
training accuracy: 0.306755
validation accuracy: 0.318000
Testing LR= 5e-07  reg= 5000.0
training accuracy: 0.371041
validation accuracy: 0.382000
Testing LR= 5e-07  reg= 15000.0
training accuracy: 0.338633
validation accuracy: 0.350000
Testing LR= 5e-07  reg= 25000.0
training accuracy: 0.337837
validation accuracy: 0.345000
Testing LR= 5e-07  reg= 50000.0
training accuracy: 0.306224
validation accuracy: 0.321000
lr 5.000000e-08 reg 5.000000e+03 train accuracy: 0.374224 val accuracy: 0.379000
lr 5.000000e-08 reg 1.500000e+04 train accuracy: 0.346388 val accuracy: 0.365000
lr 5.000000e-08 reg 2.500000e+04 train accuracy: 0.332714 val accuracy: 0.347000
lr 5.000000e-08 reg 5.000000e+04 train accuracy: 0.314714 val accuracy: 0.332000
lr 1.000000e-07 reg 5.000000e+03 train accuracy: 0.377020 val accuracy: 0.386000
lr 1.000000e-07 reg 1.500000e+04 train accuracy: 0.342898 val accuracy: 0.360000
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.330020 val accuracy: 0.346000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.306755 val accuracy: 0.318000
lr 5.000000e-07 reg 5.000000e+03 train accuracy: 0.371041 val accuracy: 0.382000
lr 5.000000e-07 reg 1.500000e+04 train accuracy: 0.338633 val accuracy: 0.350000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.337837 val accuracy: 0.345000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.306224 val accuracy: 0.321000
best validation accuracy achieved during cross-validation: 0.386000
```

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: 0.383000
```

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:

This is because svm loss uses a hinge loss type function and margins so if the new added datapoint's score is outside the margin for it's class the loss wouldn't change however softmax loss takes all data points into account no matter what their score is so the loss would change.

In [44]:
```
# 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])
```