Softmax exercise

Complete and hand in the completed notebook (including the output) with your assignment submission. You will be submitting the homework as a zip file including all the parts on the Blackboard.

This exercise is similar to the SVM exercise. In this part, you will:

- Implement a partially-vectorized and fully-vectorized **loss function** for the Softmax classifier.
- Implement the partially-vectorized and fully-vectorized expression for its **analytic gradient**.
- **Compare your implementation** with numerical gradient.
- Use a cross validation to **tune the learning rate and regularization** strength.
- **Optimize** the loss function with stochastic gradient descent (**SGD**).
- **Visualize** the final learned weights.

In [1]:
```
## Default modules
from __future__ import print_function
import random
import numpy as np
import matplotlib.pyplot as plt

## Custom modules
from ie590.data_utils import load_CIFAR10

## Ipython setup
%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]:
```
## Loading CIFAR 10 data
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 = 'ie590/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, X_test, y_test
   # 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 ie590/classifiers/softmax.py and implement the
softmax_loss_naive function.
'''

from ie590.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: 2.422585
sanity check: 2.302585
```

Inline Question 1:

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

$\color{blue}{\textit Your Answer:}$ Because we initialize W with small numbers resulting in $s=Wx \sim 0$, then each $e^{s_j} \sim 1$ for every class j. => $L_i \sim -log(1/10)$. => total loss $L = \frac{\sum{L_i}}{N} \sim L_i \sim -log(0.1)$.

Inline Question 2:

What would be the value of initial loss as number of classes becomes extremely large (inifinity)?

Your answer: When there are inifinite classes, the initial loss $-log(1/num\_class)$ will be infinity as $-log(1/\infty) = \infty$

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 ie590.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: 2.941233 analytic: 2.941233, relative error: 1.198814e-08
numerical: 2.105831 analytic: 2.105831, relative error: 2.515393e-08
numerical: 1.150306 analytic: 1.150306, relative error: 3.072279e-08
numerical: 0.037315 analytic: 0.037315, relative error: 3.110840e-07
numerical: -2.084582 analytic: -2.084582, relative error: 1.701550e-08
numerical: -0.165114 analytic: -0.165114, relative error: 1.045032e-07
numerical: 1.186945 analytic: 1.186945, relative error: 2.239306e-08
numerical: 4.630787 analytic: 4.630787, relative error: 1.792448e-08
numerical: 0.638071 analytic: 0.638071, relative error: 3.518170e-08
numerical: 0.601375 analytic: 0.601375, relative error: 4.155559e-08
numerical: 1.950394 analytic: 1.950394, relative error: 3.025558e-08
numerical: -3.551905 analytic: -3.551905, relative error: 6.656064e-09
numerical: -2.643458 analytic: -2.643458, relative error: 1.004101e-08
numerical: 3.216704 analytic: 3.216704, relative error: 1.455623e-08
numerical: 0.308688 analytic: 0.308688, relative error: 9.446612e-08
numerical: -2.073978 analytic: -2.073978, relative error: 3.175954e-09
numerical: -1.454531 analytic: -1.454531, relative error: 1.379258e-08
numerical: 2.612760 analytic: 2.612760, relative error: 1.811780e-08
numerical: 2.701114 analytic: 2.701114, relative error: 9.921668e-09
numerical: -1.423136 analytic: -1.423136, relative error: 6.897195e-10
```

In [5]:
```
# Now that we have a naive implementation of the softmax loss function and its gradient,
# implement a partially vectorized version in softmax_loss_partially_vectorized.
# The two versions should compute the same results, but the partially vectorized version should be
# relatively 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 ie590.classifiers.softmax import softmax_loss_partially_vectorized
tic = time.time()
loss_partially_vectorized, grad_partially_vectorized = softmax_loss_partially_vectorized(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('partially vectorized loss: %e computed in %fs' % (loss_partially_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_partially_vectorized, ord='fro')
print('Loss difference: %f' % np.abs(loss_naive - loss_partially_vectorized))
print('Gradient difference: %f' % grad_difference)
```

```
naive loss: 2.422585e+00 computed in 0.081546s
partially vectorized loss: 2.422585e+00 computed in 0.099844s
Loss difference: 0.000000
Gradient difference: 0.000000
```

In [6]:
```
# Now you will implement a fully vectorized version in softmax_loss_vectorized.
# The two versions should compute the same results, but the fully 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 ie590.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: 2.422585e+00 computed in 0.080823s
vectorized loss: 2.422585e+00 computed in 0.007833s
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 0.34 on the validation set.
from ie590.classifiers import Softmax
results = {}
best_val = -1
best_softmax = None ## Overwrite this variable with best trained softmax classifier.
# Feel free to experiment with some other learning rates
learning_rates = [1e-7, 5e-7]
# Feel free to experiment with other values of regularization strength
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.                          #
################################################################################
#                          START OF YOUR CODE                                  #
################################################################################
pass ## Write your code here
# define new lr and reg ranges
# learning_rates = [0.5e-7, 5.5e-7]
# regularization_strengths = [2e4, 5.5e4]
for lr in np.linspace(learning_rates[0], learning_rates[1], num = 6):
    for reg in np.linspace(regularization_strengths[0], regularization_strengths[1], num = 6):
        softmax = Softmax()
        softmax.train(X_train, y_train, lr, reg, num_iters=700, verbose=False)
        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 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 1.000000e-07 reg 2.500000e+04 train accuracy: 0.320959 val accuracy: 0.342000
lr 1.000000e-07 reg 3.000000e+04 train accuracy: 0.329184 val accuracy: 0.340000
lr 1.000000e-07 reg 3.500000e+04 train accuracy: 0.321306 val accuracy: 0.337000
lr 1.000000e-07 reg 4.000000e+04 train accuracy: 0.314673 val accuracy: 0.337000
lr 1.000000e-07 reg 4.500000e+04 train accuracy: 0.305041 val accuracy: 0.324000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.302041 val accuracy: 0.316000
lr 1.800000e-07 reg 2.500000e+04 train accuracy: 0.327612 val accuracy: 0.356000
lr 1.800000e-07 reg 3.000000e+04 train accuracy: 0.330755 val accuracy: 0.343000
lr 1.800000e-07 reg 3.500000e+04 train accuracy: 0.312286 val accuracy: 0.333000
lr 1.800000e-07 reg 4.000000e+04 train accuracy: 0.315449 val accuracy: 0.326000
lr 1.800000e-07 reg 4.500000e+04 train accuracy: 0.307980 val accuracy: 0.327000
lr 1.800000e-07 reg 5.000000e+04 train accuracy: 0.305980 val accuracy: 0.321000
lr 2.600000e-07 reg 2.500000e+04 train accuracy: 0.322347 val accuracy: 0.345000
lr 2.600000e-07 reg 3.000000e+04 train accuracy: 0.320306 val accuracy: 0.341000
lr 2.600000e-07 reg 3.500000e+04 train accuracy: 0.319592 val accuracy: 0.338000
lr 2.600000e-07 reg 4.000000e+04 train accuracy: 0.313143 val accuracy: 0.332000
lr 2.600000e-07 reg 4.500000e+04 train accuracy: 0.312245 val accuracy: 0.332000
lr 2.600000e-07 reg 5.000000e+04 train accuracy: 0.312714 val accuracy: 0.328000
lr 3.400000e-07 reg 2.500000e+04 train accuracy: 0.321633 val accuracy: 0.341000
lr 3.400000e-07 reg 3.000000e+04 train accuracy: 0.326857 val accuracy: 0.330000
lr 3.400000e-07 reg 3.500000e+04 train accuracy: 0.300490 val accuracy: 0.313000
lr 3.400000e-07 reg 4.000000e+04 train accuracy: 0.308694 val accuracy: 0.327000
lr 3.400000e-07 reg 4.500000e+04 train accuracy: 0.312735 val accuracy: 0.321000
lr 3.400000e-07 reg 5.000000e+04 train accuracy: 0.294184 val accuracy: 0.314000
lr 4.200000e-07 reg 2.500000e+04 train accuracy: 0.330755 val accuracy: 0.336000
lr 4.200000e-07 reg 3.000000e+04 train accuracy: 0.324245 val accuracy: 0.335000
lr 4.200000e-07 reg 3.500000e+04 train accuracy: 0.319388 val accuracy: 0.331000
lr 4.200000e-07 reg 4.000000e+04 train accuracy: 0.316143 val accuracy: 0.329000
lr 4.200000e-07 reg 4.500000e+04 train accuracy: 0.310306 val accuracy: 0.320000
lr 4.200000e-07 reg 5.000000e+04 train accuracy: 0.302673 val accuracy: 0.319000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.329000 val accuracy: 0.335000
lr 5.000000e-07 reg 3.000000e+04 train accuracy: 0.315816 val accuracy: 0.322000
lr 5.000000e-07 reg 3.500000e+04 train accuracy: 0.312571 val accuracy: 0.330000
lr 5.000000e-07 reg 4.000000e+04 train accuracy: 0.304735 val accuracy: 0.322000
lr 5.000000e-07 reg 4.500000e+04 train accuracy: 0.311429 val accuracy: 0.324000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.313408 val accuracy: 0.322000
best validation accuracy achieved during cross-validation: 0.356000
```

In [8]:
```
# 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: 0.344000
```

Inline Question 3 (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: *For SVM, a new datapoint may lead to a new score that makes $L_i=max(0, S_j-S_{y_i}+1) = 0$, therefore SVM L_i is unchanged. But for Softmax, the predicted score of the new datapoint always contributes to a positive term e^s and the scores of the wrong class cannot be $-\infty$, so the $e^{s_{y_i}}/\sum(e^s_j) \neq 1$, thus a new nonezero Softmax L_i is always added.*

In [9]:
```
# 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])
```