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 (https://compsci682-fa19.github.io/assignments2019/assignment1/)](https://compsci682-fa19.github.io/assignments2019/assignment1/) 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 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 pl
ots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading extenrnal modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-module
s-in-ipython
%load_ext autoreload
%autoreload 2
```

In [2]:
```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_tes
t=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_d
ev


# 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_CI
FAR10_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 [3]:
```
# 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 l
oss.
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: 2.340098
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 are initiating elements of W to be very small. Hence, when we take exponent of a very small number, we get exp(~0) => exp(0) = 1. In softmax function, if we approximate all the terms to 1, then we end up with 1/10 for each exponent of score. Essentially, we get -log(1/10) = -log(0.1). Since it's the same for every example, even if we average it over, we get -log(0.1).

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 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: 3.902427 analytic: 3.902427, relative error: 1.351823e-08
numerical: -2.169901 analytic: -2.169901, relative error: 9.049346e-0
9
numerical: 1.845966 analytic: 1.845966, relative error: 3.389752e-08
numerical: 2.336157 analytic: 2.336157, relative error: 1.254159e-08
numerical: -4.579238 analytic: -4.579238, relative error: 3.704144e-0
9
numerical: 1.454987 analytic: 1.454987, relative error: 9.621803e-09
numerical: -0.377293 analytic: -0.377293, relative error: 1.466624e-0
7
numerical: -0.012592 analytic: -0.012592, relative error: 1.709682e-0
6
numerical: -0.005379 analytic: -0.005379, relative error: 7.520723e-0
6
numerical: 2.694451 analytic: 2.694451, relative error: 9.489678e-09
numerical: -0.440033 analytic: -0.440033, relative error: 1.006129e-0
8
numerical: 0.960237 analytic: 0.960237, relative error: 4.966151e-08
numerical: 0.846898 analytic: 0.846898, relative error: 7.480986e-08
numerical: 2.486037 analytic: 2.486037, relative error: 2.742079e-08
numerical: 0.244251 analytic: 0.244251, relative error: 1.832118e-07
numerical: 1.300372 analytic: 1.300372, relative error: 3.288925e-08
numerical: -0.349705 analytic: -0.349705, relative error: 7.449021e-0
8
numerical: -0.803577 analytic: -0.803577, relative error: 3.878746e-0
8
numerical: 0.636018 analytic: 0.636018, relative error: 3.871182e-08
numerical: -0.159012 analytic: -0.159012, relative error: 1.621705e-0
7
```

In [5]:
```
# Now that we have a naive implementation of the softmax loss functio
n and its gradient,
# implement a vectorized version in softmax_loss_vectorized.
# The two versions should compute the same results, but the vectorize
d 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='f
ro')
print('Loss difference: %f' % np.abs(loss_naive - loss_vectorized))
print('Gradient difference: %f' % grad_difference)
```

```
naive loss: 2.340098e+00 computed in 0.084918s
vectorized loss: 2.340098e+00 computed in 0.002776s
Loss difference: 0.000000
Gradient difference: 0.000000
```

In [7]:
```
# Use the validation set to tune hyperparameters (regularization stre
ngth and
# learning rate). You should experiment with different ranges for the
learning
# rates and regularization strengths; if you are careful you should b
e 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 = [1e-8, 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 SV
M; save    #
# the best trained softmax classifer in best_softmax.
#
######################################################################
###########
num_iters = 10
for it in range(num_iters):
    for jt in range(num_iters):
        softmax = Softmax()
        learning_rate = learning_rates[0] + it * ((learning_rates[1]
- learning_rates[0]) / num_iters)
        reg = regularization_strengths[0] + jt * ((regularization_str
engths[1] - regularization_strengths[0])/ num_iters)
        loss_hist = softmax.train(X_train, y_train, learning_rate=lea
rning_rate, reg=reg,
                              num_iters=3000, verbose=False)

        y_train_pred = softmax.predict(X_train)
        y_val_pred = softmax.predict(X_val)
        training_accuracy = np.mean(y_train == y_train_pred)
        validation_accuracy = np.mean(y_val == y_val_pred)
        results[(learning_rate, reg)] = (training_accuracy, validatio
n_accuracy)
        if validation_accuracy > best_val:
            best_val = validation_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-08 reg 2.500000e+04 train accuracy: 0.263367 val accurac
y: 0.275000
lr 1.000000e-08 reg 2.750000e+04 train accuracy: 0.261469 val accurac
y: 0.257000
lr 1.000000e-08 reg 3.000000e+04 train accuracy: 0.258673 val accurac
y: 0.270000
lr 1.000000e-08 reg 3.250000e+04 train accuracy: 0.262061 val accurac
y: 0.269000
lr 1.000000e-08 reg 3.500000e+04 train accuracy: 0.274041 val accurac
y: 0.305000
lr 1.000000e-08 reg 3.750000e+04 train accuracy: 0.288204 val accurac
y: 0.306000
lr 1.000000e-08 reg 4.000000e+04 train accuracy: 0.283612 val accurac
y: 0.297000
lr 1.000000e-08 reg 4.250000e+04 train accuracy: 0.295265 val accurac
y: 0.311000
lr 1.000000e-08 reg 4.500000e+04 train accuracy: 0.293816 val accurac
y: 0.301000
lr 1.000000e-08 reg 4.750000e+04 train accuracy: 0.291673 val accurac
y: 0.281000
lr 5.900000e-08 reg 2.500000e+04 train accuracy: 0.325327 val accurac
y: 0.338000
lr 5.900000e-08 reg 2.750000e+04 train accuracy: 0.326388 val accurac
y: 0.341000
lr 5.900000e-08 reg 3.000000e+04 train accuracy: 0.326429 val accurac
y: 0.340000
lr 5.900000e-08 reg 3.250000e+04 train accuracy: 0.323653 val accurac
y: 0.337000
lr 5.900000e-08 reg 3.500000e+04 train accuracy: 0.319531 val accurac
y: 0.334000
lr 5.900000e-08 reg 3.750000e+04 train accuracy: 0.321755 val accurac
y: 0.336000
lr 5.900000e-08 reg 4.000000e+04 train accuracy: 0.307571 val accurac
y: 0.328000
lr 5.900000e-08 reg 4.250000e+04 train accuracy: 0.307551 val accurac
y: 0.322000
lr 5.900000e-08 reg 4.500000e+04 train accuracy: 0.310224 val accurac
y: 0.327000
lr 5.900000e-08 reg 4.750000e+04 train accuracy: 0.304286 val accurac
y: 0.323000
lr 1.080000e-07 reg 2.500000e+04 train accuracy: 0.330469 val accurac
y: 0.350000
lr 1.080000e-07 reg 2.750000e+04 train accuracy: 0.325612 val accurac
y: 0.336000
lr 1.080000e-07 reg 3.000000e+04 train accuracy: 0.319612 val accurac
y: 0.335000
lr 1.080000e-07 reg 3.250000e+04 train accuracy: 0.314571 val accurac
y: 0.329000
lr 1.080000e-07 reg 3.500000e+04 train accuracy: 0.322959 val accurac
y: 0.327000
lr 1.080000e-07 reg 3.750000e+04 train accuracy: 0.308837 val accurac
y: 0.319000
lr 1.080000e-07 reg 4.000000e+04 train accuracy: 0.316163 val accurac
y: 0.329000
lr 1.080000e-07 reg 4.250000e+04 train accuracy: 0.315898 val accurac
y: 0.330000
lr 1.080000e-07 reg 4.500000e+04 train accuracy: 0.317490 val accurac
```

```
y: 0.334000
lr 1.080000e-07 reg 4.750000e+04 train accuracy: 0.311612 val accurac
y: 0.325000
lr 1.570000e-07 reg 2.500000e+04 train accuracy: 0.331857 val accurac
y: 0.344000
lr 1.570000e-07 reg 2.750000e+04 train accuracy: 0.329612 val accurac
y: 0.339000
lr 1.570000e-07 reg 3.000000e+04 train accuracy: 0.316510 val accurac
y: 0.336000
lr 1.570000e-07 reg 3.250000e+04 train accuracy: 0.316367 val accurac
y: 0.340000
lr 1.570000e-07 reg 3.500000e+04 train accuracy: 0.320592 val accurac
y: 0.328000
lr 1.570000e-07 reg 3.750000e+04 train accuracy: 0.320959 val accurac
y: 0.333000
lr 1.570000e-07 reg 4.000000e+04 train accuracy: 0.313571 val accurac
y: 0.330000
lr 1.570000e-07 reg 4.250000e+04 train accuracy: 0.312449 val accurac
y: 0.328000
lr 1.570000e-07 reg 4.500000e+04 train accuracy: 0.312102 val accurac
y: 0.324000
lr 1.570000e-07 reg 4.750000e+04 train accuracy: 0.309510 val accurac
y: 0.323000
lr 2.060000e-07 reg 2.500000e+04 train accuracy: 0.330082 val accurac
y: 0.340000
lr 2.060000e-07 reg 2.750000e+04 train accuracy: 0.327980 val accurac
y: 0.350000
lr 2.060000e-07 reg 3.000000e+04 train accuracy: 0.325673 val accurac
y: 0.327000
lr 2.060000e-07 reg 3.250000e+04 train accuracy: 0.311286 val accurac
y: 0.327000
lr 2.060000e-07 reg 3.500000e+04 train accuracy: 0.316714 val accurac
y: 0.334000
lr 2.060000e-07 reg 3.750000e+04 train accuracy: 0.322102 val accurac
y: 0.336000
lr 2.060000e-07 reg 4.000000e+04 train accuracy: 0.310796 val accurac
y: 0.333000
lr 2.060000e-07 reg 4.250000e+04 train accuracy: 0.310857 val accurac
y: 0.329000
lr 2.060000e-07 reg 4.500000e+04 train accuracy: 0.314449 val accurac
y: 0.331000
lr 2.060000e-07 reg 4.750000e+04 train accuracy: 0.308980 val accurac
y: 0.328000
lr 2.550000e-07 reg 2.500000e+04 train accuracy: 0.333082 val accurac
y: 0.349000
lr 2.550000e-07 reg 2.750000e+04 train accuracy: 0.329592 val accurac
y: 0.339000
lr 2.550000e-07 reg 3.000000e+04 train accuracy: 0.311163 val accurac
y: 0.338000
lr 2.550000e-07 reg 3.250000e+04 train accuracy: 0.326837 val accurac
y: 0.347000
lr 2.550000e-07 reg 3.500000e+04 train accuracy: 0.314898 val accurac
y: 0.337000
lr 2.550000e-07 reg 3.750000e+04 train accuracy: 0.308265 val accurac
y: 0.325000
lr 2.550000e-07 reg 4.000000e+04 train accuracy: 0.308531 val accurac
y: 0.317000
```

```
lr 2.550000e-07 reg 4.250000e+04 train accuracy: 0.314122 val accurac
y: 0.322000
lr 2.550000e-07 reg 4.500000e+04 train accuracy: 0.306184 val accurac
y: 0.327000
lr 2.550000e-07 reg 4.750000e+04 train accuracy: 0.305959 val accurac
y: 0.323000
lr 3.040000e-07 reg 2.500000e+04 train accuracy: 0.335796 val accurac
y: 0.344000
lr 3.040000e-07 reg 2.750000e+04 train accuracy: 0.328959 val accurac
y: 0.336000
lr 3.040000e-07 reg 3.000000e+04 train accuracy: 0.313224 val accurac
y: 0.329000
lr 3.040000e-07 reg 3.250000e+04 train accuracy: 0.306837 val accurac
y: 0.320000
lr 3.040000e-07 reg 3.500000e+04 train accuracy: 0.313612 val accurac
y: 0.325000
lr 3.040000e-07 reg 3.750000e+04 train accuracy: 0.315184 val accurac
y: 0.330000
lr 3.040000e-07 reg 4.000000e+04 train accuracy: 0.299490 val accurac
y: 0.322000
lr 3.040000e-07 reg 4.250000e+04 train accuracy: 0.307796 val accurac
y: 0.338000
lr 3.040000e-07 reg 4.500000e+04 train accuracy: 0.305531 val accurac
y: 0.326000
lr 3.040000e-07 reg 4.750000e+04 train accuracy: 0.292571 val accurac
y: 0.314000
lr 3.530000e-07 reg 2.500000e+04 train accuracy: 0.329408 val accurac
y: 0.337000
lr 3.530000e-07 reg 2.750000e+04 train accuracy: 0.325857 val accurac
y: 0.346000
lr 3.530000e-07 reg 3.000000e+04 train accuracy: 0.317551 val accurac
y: 0.331000
lr 3.530000e-07 reg 3.250000e+04 train accuracy: 0.314531 val accurac
y: 0.345000
lr 3.530000e-07 reg 3.500000e+04 train accuracy: 0.322980 val accurac
y: 0.336000
lr 3.530000e-07 reg 3.750000e+04 train accuracy: 0.321857 val accurac
y: 0.332000
lr 3.530000e-07 reg 4.000000e+04 train accuracy: 0.319143 val accurac
y: 0.327000
lr 3.530000e-07 reg 4.250000e+04 train accuracy: 0.319837 val accurac
y: 0.340000
lr 3.530000e-07 reg 4.500000e+04 train accuracy: 0.307857 val accurac
y: 0.319000
lr 3.530000e-07 reg 4.750000e+04 train accuracy: 0.303306 val accurac
y: 0.312000
lr 4.020000e-07 reg 2.500000e+04 train accuracy: 0.329490 val accurac
y: 0.339000
lr 4.020000e-07 reg 2.750000e+04 train accuracy: 0.329184 val accurac
y: 0.347000
lr 4.020000e-07 reg 3.000000e+04 train accuracy: 0.311041 val accurac
y: 0.325000
lr 4.020000e-07 reg 3.250000e+04 train accuracy: 0.311306 val accurac
y: 0.325000
lr 4.020000e-07 reg 3.500000e+04 train accuracy: 0.316041 val accurac
y: 0.329000
lr 4.020000e-07 reg 3.750000e+04 train accuracy: 0.323592 val accurac
```

```
y: 0.335000
lr 4.020000e-07 reg 4.000000e+04 train accuracy: 0.312347 val accurac
y: 0.327000
lr 4.020000e-07 reg 4.250000e+04 train accuracy: 0.305918 val accurac
y: 0.318000
lr 4.020000e-07 reg 4.500000e+04 train accuracy: 0.308531 val accurac
y: 0.323000
lr 4.020000e-07 reg 4.750000e+04 train accuracy: 0.309878 val accurac
y: 0.319000
lr 4.510000e-07 reg 2.500000e+04 train accuracy: 0.322449 val accurac
y: 0.333000
lr 4.510000e-07 reg 2.750000e+04 train accuracy: 0.329714 val accurac
y: 0.347000
lr 4.510000e-07 reg 3.000000e+04 train accuracy: 0.321061 val accurac
y: 0.332000
lr 4.510000e-07 reg 3.250000e+04 train accuracy: 0.305776 val accurac
y: 0.321000
lr 4.510000e-07 reg 3.500000e+04 train accuracy: 0.310490 val accurac
y: 0.323000
lr 4.510000e-07 reg 3.750000e+04 train accuracy: 0.316796 val accurac
y: 0.328000
lr 4.510000e-07 reg 4.000000e+04 train accuracy: 0.312612 val accurac
y: 0.324000
lr 4.510000e-07 reg 4.250000e+04 train accuracy: 0.294061 val accurac
y: 0.306000
lr 4.510000e-07 reg 4.500000e+04 train accuracy: 0.301694 val accurac
y: 0.320000
lr 4.510000e-07 reg 4.750000e+04 train accuracy: 0.307633 val accurac
y: 0.312000
best validation accuracy achieved during cross-validation: 0.350000
```

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_acc
uracy, ))
```

```
softmax on raw pixels final test set accuracy: 0.339000
```

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: We have accounted for numerical instability for Softmax because the scores of some points might be very large and when we do an exp on that number, it tends to go to infinity. In case of SVM, the loss remains unchanged because its a max function of difference of scores. But in case of Softmax, if the score is too large, loss changes by a lot because there is exponential function in the loss.

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', 'hor
se', '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])
```

In [ ]:
```
```