Softmax exercise

Complete and hand in this completed worksheet (including its outputs and any supporting code outside of the worksheet) with your assignment submission.

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 cecs551.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-ipyt
hon
%reload_ext autoreload
%autoreload 2
```

In [4]:
```
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 = 'cecs551/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)
```

```
Clear previously loaded data.
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 **cecs551/classifiers/softmax.py**.

In [5]:
```
# First implement the naive softmax loss function with nested loops.
# Open the file cecs551/classifiers/softmax.py and implement the
# softmax_loss_naive function.

from cecs551.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: 2.376488
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 know 0.1 means 1/10. Here, the number of classes are 10. So..

In [6]:
```
# 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 cecs551.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: -0.109316 analytic: -0.109316, relative error: 3.603506e-07
numerical: 2.434020 analytic: 2.434020, relative error: 1.508750e-08
numerical: -0.086424 analytic: -0.086424, relative error: 8.655078e-07
numerical: -3.733651 analytic: -3.733651, relative error: 7.839935e-09
numerical: -1.532499 analytic: -1.532499, relative error: 1.421454e-08
numerical: -0.199050 analytic: -0.199050, relative error: 6.552836e-08
numerical: 0.068618 analytic: 0.068618, relative error: 2.868472e-07
numerical: -1.997530 analytic: -1.997530, relative error: 5.295487e-09
numerical: 2.040039 analytic: 2.040039, relative error: 1.981730e-08
numerical: -1.309043 analytic: -1.309043, relative error: 2.343270e-08
numerical: -0.380486 analytic: -0.380486, relative error: 2.283512e-07
numerical: -0.183080 analytic: -0.183080, relative error: 1.401142e-07
numerical: -1.054993 analytic: -1.054993, relative error: 3.644825e-08
numerical: -0.457855 analytic: -0.457855, relative error: 4.546535e-08
numerical: 0.341838 analytic: 0.341838, relative error: 1.763441e-07
numerical: 0.006572 analytic: 0.006572, relative error: 9.462064e-06
numerical: -0.894371 analytic: -0.894371, relative error: 2.360891e-09
numerical: -2.082064 analytic: -2.082064, relative error: 1.506599e-09
numerical: -2.914452 analytic: -2.914452, relative error: 1.117781e-08
numerical: 1.112731 analytic: 1.112730, relative error: 6.038383e-08
```

In [7]:
```
# 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 cecs551.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: 2.376488e+00 computed in 41.785849s
vectorized loss: 2.376488e+00 computed in 0.016804s
Loss difference: 0.000000
Gradient difference: 0.000000
```

In [8]:
```
# 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 cecs551.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.
#
##############################################################################
##
def compute_accuracy(y, y_pred):
    return np.mean(y == y_pred)


for lr in learning_rates:
    for reg in regularization_strengths:
        # train softmax classifier
        model = Softmax()
        model.train(X_train, y_train, learning_rate=lr, reg=reg, num_iters=150
0, verbose=False)

        # compute accuracy
        train_accuracy = compute_accuracy(y_train, model.predict(X_train))
        val_accuracy = compute_accuracy(y_val, model.predict(X_val))

        # store accuracy in dictionary
        results[(lr, reg)] = (train_accuracy, val_accuracy)

        # check if validation accuracy is best
        if val_accuracy > best_val:
            best_val = val_accuracy
            best_softmax = model
##############################################################################
##
#                               END OF YOUR CODE
#
##############################################################################
##

# Print out results.
for lr, reg in sorted(results):
    train_accuracy, val_accuracy = results[(lr, reg)]
    print('Learning Rate %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)
```

```
Learning Rate 1.000000e-07 reg 2.500000e+04 train accuracy: 0.350449 val accu
racy: 0.365000
Learning Rate 1.000000e-07 reg 5.000000e+04 train accuracy: 0.326714 val accu
racy: 0.341000
Learning Rate 5.000000e-07 reg 2.500000e+04 train accuracy: 0.351082 val accu
racy: 0.348000
Learning Rate 5.000000e-07 reg 5.000000e+04 train accuracy: 0.324857 val accu
racy: 0.337000
best validation accuracy achieved during cross-validation: 0.365000
```

In [9]:

```
# 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: 0.363000
```

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

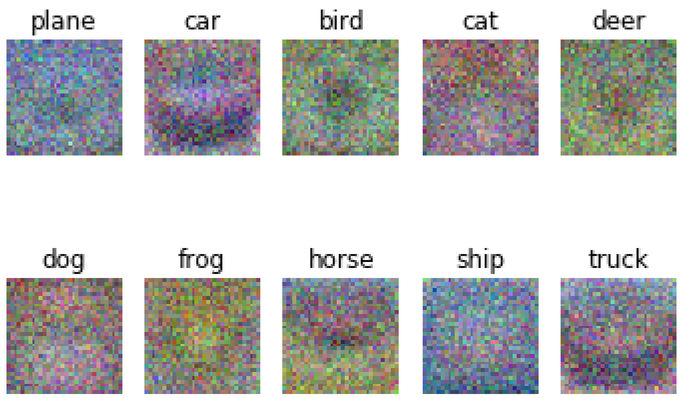
Your explanation: In SVM, if the new data point has a score which is out of margin as compared to correct class score, there would be no difference in loss, but in the case with softmax classifier loss, if the new added data point is close to positive infinity, it will profoundly affect the loss and loss of sogtmax will change.

In [10]:
```
# 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])
```



In [ ]: