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/cs175/assignments.html)](http://vision.stanford.edu/teaching/cs175/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]:

```python
import random
import numpy as np
from cs175.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 [2]:

```
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 = 'cs175\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


# 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 **cs175/classifiers/softmax.py**.

In [3]:

```
# First implement the naive softmax loss function with nested loops.
# Open the file cs175/classifiers/softmax.py and implement the
# softmax_loss_naive function.

from cs175.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: 2.328687
sanity check: 2.302585
```

Inline Question 1:

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

Your answer: because we use the random.randn to get w, which means we have the probability to choose correct from 10 classes.

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 cs175.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: 0.145382 analytic: 0.145382, relative error: 1.960608e-07
numerical: -0.520126 analytic: -0.520126, relative error: 5.982185e-08
numerical: 0.021003 analytic: 0.021003, relative error: 2.360348e-06
numerical: 0.334317 analytic: 0.334317, relative error: 1.948847e-08
numerical: -0.466329 analytic: -0.466329, relative error: 1.780132e-08
numerical: -3.139259 analytic: -3.139259, relative error: 2.058990e-08
numerical: -2.472193 analytic: -2.472193, relative error: 4.141986e-08
numerical: 2.965425 analytic: 2.965425, relative error: 1.417015e-08
numerical: 3.437026 analytic: 3.437026, relative error: 1.020942e-09
numerical: -0.551950 analytic: -0.551950, relative error: 4.074553e-08
numerical: 0.539567 analytic: 0.540316, relative error: 6.932523e-04
numerical: -1.955623 analytic: -1.953069, relative error: 6.536117e-04
numerical: -0.918482 analytic: -0.912230, relative error: 3.414982e-03
numerical: 0.513119 analytic: 0.519976, relative error: 6.638013e-03
numerical: 0.561208 analytic: 0.560337, relative error: 7.760685e-04
numerical: 1.081409 analytic: 1.077163, relative error: 1.966669e-03
numerical: 1.009476 analytic: 1.007877, relative error: 7.928273e-04
numerical: -1.080304 analytic: -1.078940, relative error: 6.315615e-04
numerical: -3.622707 analytic: -3.628089, relative error: 7.422056e-04
numerical: -3.945447 analytic: -3.943237, relative error: 2.801827e-04
```

In [5]:

```
# 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 cs175.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: 2.328687e+00 computed in 0.061868s
vectorized loss: 2.328687e+00 computed in 0.004000s
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 cs175.classifiers import Softmax
results = {}
best_val = -1
best_softmax = None
learning_rates = [1e-7,2e-7,3e-7,4e-7]
regularization_strengths = [2e4,3e4,4e4,5e4]

################################################################################
# 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 i in learning_rates:
    for j in regularization_strengths:
        s1 = Softmax()

        s1.train(X_train, y_train, learning_rate=i, reg=j, num_iters=1500)

        ytr = s1.predict(X_train)
        train_accuracy = np.mean(y_train == ytr)

        yval = s1.predict(X_val)
        val_accuracy = np.mean(y_val == yval)
        results[(i, j)] = (train_accuracy, val_accuracy)

        if best_val < val_accuracy:
            best_val = val_accuracy
            best_softmax = s1


################################################################################
#                              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 2.000000e+04 train accuracy: 0.353061 val accuracy: 0.370000
lr 1.000000e-07 reg 3.000000e+04 train accuracy: 0.346612 val accuracy: 0.358000
lr 1.000000e-07 reg 4.000000e+04 train accuracy: 0.334469 val accuracy: 0.345000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.328571 val accuracy: 0.340000
lr 2.000000e-07 reg 2.000000e+04 train accuracy: 0.355122 val accuracy: 0.379000
lr 2.000000e-07 reg 3.000000e+04 train accuracy: 0.347898 val accuracy: 0.362000
lr 2.000000e-07 reg 4.000000e+04 train accuracy: 0.333633 val accuracy: 0.354000
lr 2.000000e-07 reg 5.000000e+04 train accuracy: 0.327694 val accuracy: 0.351000
lr 3.000000e-07 reg 2.000000e+04 train accuracy: 0.353673 val accuracy: 0.372000
lr 3.000000e-07 reg 3.000000e+04 train accuracy: 0.347592 val accuracy: 0.370000
lr 3.000000e-07 reg 4.000000e+04 train accuracy: 0.336000 val accuracy: 0.350000
```

```
lr 3.000000e-07 reg 5.000000e+04 train accuracy: 0.333224 val accuracy: 0.358000
lr 4.000000e-07 reg 2.000000e+04 train accuracy: 0.347755 val accuracy: 0.365000
lr 4.000000e-07 reg 3.000000e+04 train accuracy: 0.340878 val accuracy: 0.351000
lr 4.000000e-07 reg 4.000000e+04 train accuracy: 0.330061 val accuracy: 0.344000
lr 4.000000e-07 reg 5.000000e+04 train accuracy: 0.320673 val accuracy: 0.349000
best validation accuracy achieved during cross-validation: 0.379000
```

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: 0.361000
```

In [8]:

```
# 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])
```