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/) 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 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-ipytho
n
%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 = '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 [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 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: 2.337618
sanity check: 2.302585
```

Inline Question 1:

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

Your answer: Here, the weight matrix is a random set weights. Since we're using the CIFAR-10 dataset, there are 10 classes. Without any learning, on random, the probability of a correct classification is 0.1 and the softmax loss is the negative log of the probability of the correct class ie. -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: -0.702325 analytic: -0.702325, relative error: 8.434424e-
08
numerical: 3.212848 analytic: 3.212847, relative error: 5.669634e-09
numerical: 1.485497 analytic: 1.485497, relative error: 3.899620e-08
numerical: -0.058346 analytic: -0.058346, relative error: 2.205874e-
07
numerical: 0.229335 analytic: 0.229335, relative error: 3.806034e-07
numerical: 3.503413 analytic: 3.503413, relative error: 3.943833e-09
numerical: -1.735472 analytic: -1.735472, relative error: 2.420797e-
08
numerical: -3.558007 analytic: -3.558007, relative error: 2.351884e-
08
numerical: 1.499690 analytic: 1.499690, relative error: 3.377313e-08
numerical: 0.982472 analytic: 0.982471, relative error: 6.262293e-08
numerical: -0.315025 analytic: -0.315025, relative error: 1.025980e-
07
numerical: 3.793594 analytic: 3.793594, relative error: 2.105647e-08
numerical: 2.206939 analytic: 2.206938, relative error: 2.439230e-08
numerical: 2.070291 analytic: 2.070291, relative error: 4.208052e-08
numerical: -1.980808 analytic: -1.980808, relative error: 1.392543e-
08
numerical: 0.736043 analytic: 0.736043, relative error: 1.321836e-08
numerical: 2.634442 analytic: 2.634442, relative error: 1.652637e-08
numerical: 2.535977 analytic: 2.535977, relative error: 1.157886e-08
numerical: 1.540301 analytic: 1.540301, relative error: 4.495272e-08
numerical: 2.846523 analytic: 2.846523, relative error: 2.604984e-09
```

In [5]:

```
# Now that we have a naive implementation of the softmax loss function and its g
radient,
# implement a vectorized version in softmax_loss_vectorized.
# The two versions should compute the same results, but the vectorized version s
hould 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, 0.00
0005)
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: 2.337618e+00 computed in 0.124062s
vectorized loss: 2.337618e+00 computed in 0.005528s
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 = [1e-5, 1e-6]
regularization_strengths = [1e3, 1.5e3, 2e3]

################################################################################
# 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:
        # new instance of SoftMax
        sm = Softmax()
        loss_hist = sm.train(X_train, y_train, learning_rate=lr, reg=reg,
                      num_iters=1000, verbose=False)
        # evaluate the performance on the training set
        y_train_pred = sm.predict(X_train)
        train_accuracy = np.mean(y_train == y_train_pred)
        # evaluate the performance on the validation set
        y_val_pred = sm.predict(X_val)
        val_accuracy = np.mean(y_val == y_val_pred)
        # store the results
        results[(lr, reg)] = (train_accuracy, val_accuracy)
        if (val_accuracy > best_val):
            best_val = val_accuracy
            best_softmax = sm
################################################################################
#                              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-06 reg 1.000000e+03 train accuracy: 0.394449 val accura
cy: 0.397000
lr 1.000000e-06 reg 1.500000e+03 train accuracy: 0.395714 val accura
cy: 0.384000
lr 1.000000e-06 reg 2.000000e+03 train accuracy: 0.391959 val accura
cy: 0.402000
lr 1.000000e-05 reg 1.000000e+03 train accuracy: 0.224571 val accura
cy: 0.238000
lr 1.000000e-05 reg 1.500000e+03 train accuracy: 0.255796 val accura
cy: 0.241000
lr 1.000000e-05 reg 2.000000e+03 train accuracy: 0.228469 val accura
cy: 0.234000
best validation accuracy achieved during cross-validation: 0.402000
```

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: 0.381000
```

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: The SVM loss is a sum of non-negative margins between the score of an incorrect class and that of a correct class (and a constant). On the other hand, the Softmax classifier loss is the negative log of the probability of the correct class. If I add a new datapoint to the training set where the margin between the score of the new datapoint and that of the correct class is negative, it doesn't add to the loss and leaves the SVM loss unchanged. However, the Softmax classifier loss considers the distribution of probabilities of all datapoints and so, alters the loss.

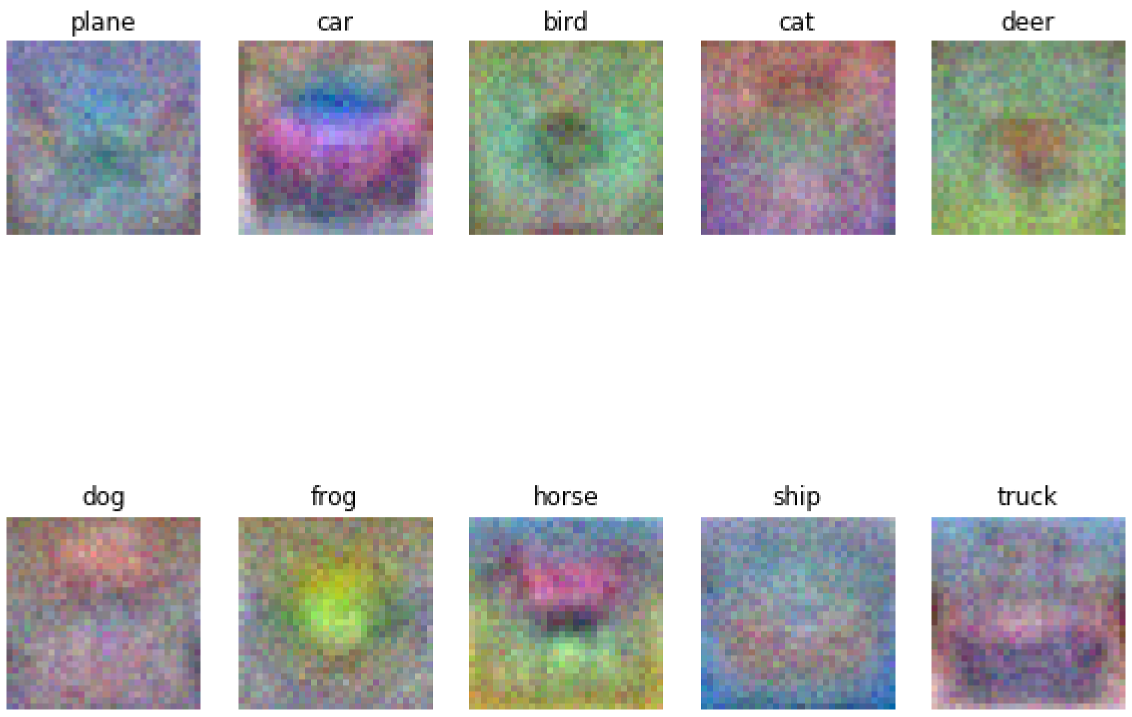
In [8]:

```
# 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])
```


plane
car
bird
cat
deer
dog
frog
horse
ship
truck