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/cs231n/assignments.html)](http://vision.stanford.edu/teaching/cs231n/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]:

```
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

from __future__ import print_function

%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-i
python
%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 prepar
e
    it for the linear classifier. These are the same steps as we used for t
he
    SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = 'cs231n/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 **cs231n/classifiers/softmax.py**.

In [31]:

```
# First implement the naive softmax loss function with nested loops.
# Open the file cs231n/classifiers/softmax.py and implement the
# softmax_loss_naive function.

from cs231n.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: 2.373003
sanity check: 2.302585
```

Inline Question 1:

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

Your answer: *Fill this in*

In [32]:
```
# 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 cs231n.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: -2.632612 analytic: -2.632612, relative error: 1.442458e-09
numerical: 0.332096 analytic: 0.332096, relative error: 6.510503e-08
numerical: -2.973513 analytic: -2.973513, relative error: 1.045611e-08
numerical: 0.150323 analytic: 0.150323, relative error: 6.700865e-08
numerical: -1.207583 analytic: -1.207583, relative error: 1.425016e-08
numerical: 1.179986 analytic: 1.179986, relative error: 1.304283e-08
numerical: 5.133099 analytic: 5.133099, relative error: 7.469337e-09
numerical: 0.549502 analytic: 0.549502, relative error: 4.522896e-10
numerical: -4.675290 analytic: -4.675290, relative error: 1.710769e-09
numerical: -0.263622 analytic: -0.263622, relative error: 5.262522e-08
numerical: 1.166657 analytic: 1.166657, relative error: 1.959579e-08
numerical: 1.168515 analytic: 1.168515, relative error: 7.786343e-09
numerical: -0.081474 analytic: -0.081474, relative error: 1.167332e-08
numerical: 0.432656 analytic: 0.432656, relative error: 9.790118e-08
numerical: -3.305547 analytic: -3.305548, relative error: 2.075457e-08
numerical: 2.443017 analytic: 2.443017, relative error: 2.031570e-08
numerical: -2.179839 analytic: -2.179839, relative error: 1.478041e-08
numerical: -1.583506 analytic: -1.583506, relative error: 3.169155e-08
numerical: 0.061351 analytic: 0.061351, relative error: 1.160553e-07
numerical: -0.853399 analytic: -0.853400, relative error: 4.621826e-08
```

In [110]:

```
# 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 vers
ion 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 cs231n.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 versi
ons
# 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: 2.373003e+00 computed in 0.124771s
vectorized loss: 2.373003e+00 computed in 0.011381s
Loss difference: 0.000000
Gradient difference: 0.000000
```

In [112]:

```
# Use the validation set to tune hyperparameters (regularization strength a
nd
# learning rate). You should experiment with different ranges for the learn
ing
# 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 cs231n.classifiers import Softmax
results = {}
best_val = -1
best_softmax = None
learning_rates = [1e-7,1e-6,1e-5, 1e-4, 1e-3]
regularization_strengths = [1e3,8e4, 2.5e4, 5e4, 8e4]

###########################################################################
#####
# TODO:
#
# Use the validation set to set the learning rate and regularization streng
th. #
# This should be identical to the validation that you did for the SVM; save
#
# the best trained softmax classifer in best_softmax.
#
###########################################################################
#####
for alpha_ in learning_rates:
    for beta_ in regularization_strengths:
        print('----------------------------')
        print('Current alpha: %f, beta: %f' % (alpha_, beta_))
        softmax = Softmax()
        loss_hist = softmax.train(X_train, y_train, learning_rate=alpha_, r
eg=beta_,
                    num_iters=1500, verbose=False)
        y_val_pred = softmax.predict(X_val)
        val_pred_acc = np.mean(y_val_pred == y_val)
        y_train_pred = softmax.predict(X_train)
        train_pred_acc = np.mean(y_train_pred == y_train)
        results[(alpha_,beta_)]=(train_pred_acc, val_pred_acc)
        if val_pred_acc > best_val:
            best_val=val_pred_acc
            best_softmax=softmax

##softmax##################################################################
#########
#                              END OF YOUR CODE
#
####softmax################################################################
#########

# 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' % bes
t_val)
```

```
----------------------------
Current alpha: 0.000000, beta: 1000.000000
----------------------------
Current alpha: 0.000000, beta: 80000.000000
----------------------------
Current alpha: 0.000000, beta: 25000.000000
----------------------------
Current alpha: 0.000000, beta: 50000.000000
----------------------------
Current alpha: 0.000000, beta: 80000.000000
----------------------------
Current alpha: 0.000001, beta: 1000.000000
----------------------------
Current alpha: 0.000001, beta: 80000.000000
----------------------------
Current alpha: 0.000001, beta: 25000.000000
----------------------------
Current alpha: 0.000001, beta: 50000.000000
----------------------------
Current alpha: 0.000001, beta: 80000.000000
----------------------------
Current alpha: 0.000010, beta: 1000.000000
----------------------------
Current alpha: 0.000010, beta: 80000.000000
----------------------------
Current alpha: 0.000010, beta: 25000.000000
----------------------------
Current alpha: 0.000010, beta: 50000.000000
----------------------------
Current alpha: 0.000010, beta: 80000.000000
----------------------------
Current alpha: 0.000100, beta: 1000.000000
----------------------------
Current alpha: 0.000100, beta: 80000.000000
----------------------------
Current alpha: 0.000100, beta: 25000.000000
----------------------------
Current alpha: 0.000100, beta: 50000.000000
----------------------------
Current alpha: 0.000100, beta: 80000.000000
----------------------------
Current alpha: 0.001000, beta: 1000.000000
----------------------------
Current alpha: 0.001000, beta: 80000.000000
----------------------------
Current alpha: 0.001000, beta: 25000.000000
----------------------------
Current alpha: 0.001000, beta: 50000.000000
----------------------------
Current alpha: 0.001000, beta: 80000.000000
lr 1.000000e-07 reg 1.000000e+03 train accuracy: 0.267122 val accuracy: 0.2
62000
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.329980 val accuracy: 0.3
50000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.305571 val accuracy: 0.3
25000
lr 1.000000e-07 reg 8.000000e+04 train accuracy: 0.287918 val accuracy: 0.3
07000
lr 1.000000e-06 reg 1.000000e+03 train accuracy: 0.401367 val accuracy: 0.4
09000
lr 1.000000e-06 reg 2.500000e+04 train accuracy: 0.314224 val accuracy: 0.3
19000
lr 1.000000e-06 reg 5.000000e+04 train accuracy: 0.307020 val accuracy: 0.3
```

```
/home/nan/StanfordCS231/assignment1/cs231n/classifiers/softmax.py:82: Runti
meWarning: overflow encountered in exp
  denoms=np.sum(np.exp(scores), axis=1)
/home/nan/StanfordCS231/assignment1/cs231n/classifiers/softmax.py:83: Runti
meWarning: overflow encountered in exp
  temp=np.exp(scores_y)/denoms
/home/nan/StanfordCS231/assignment1/cs231n/classifiers/softmax.py:83: Runti
meWarning: invalid value encountered in true_divide
  temp=np.exp(scores_y)/denoms
/home/nan/StanfordCS231/assignment1/cs231n/classifiers/softmax.py:84: Runti
meWarning: divide by zero encountered in log
  loss = np.sum(-np.log(temp))
/home/nan/StanfordCS231/assignment1/cs231n/classifiers/softmax.py:85: Runti
meWarning: overflow encountered in exp
  temp=np.exp(scores)
/home/nan/StanfordCS231/assignment1/cs231n/classifiers/softmax.py:86: Runti
meWarning: invalid value encountered in true_divide
  temp[np.arange(num_train),y]-=1
```

In [113]:

```
# 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,
))
```

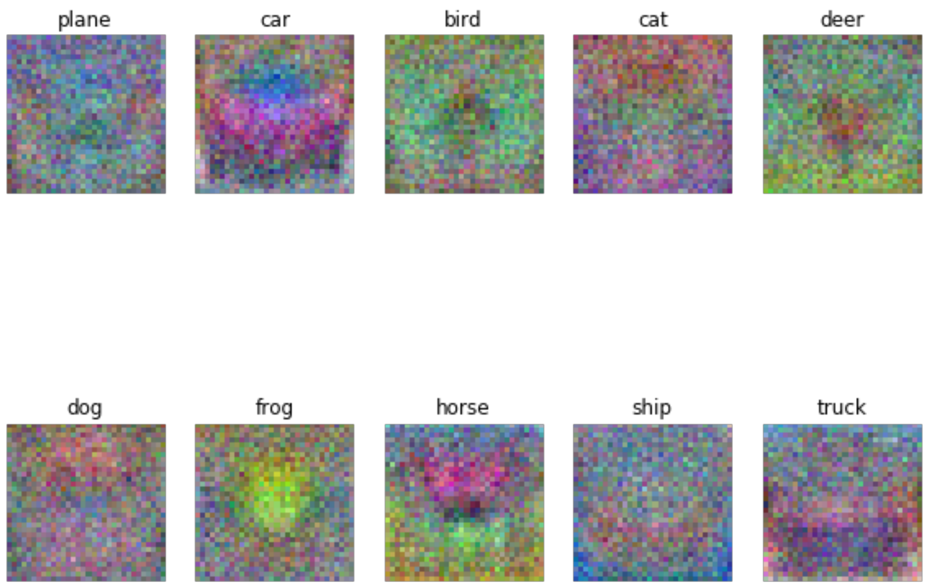
```
softmax on raw pixels final test set accuracy: 0.390000
```

In [114]:
```
# 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 [ ]:
```
```