Softmax exercise

In thsi 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 [322]:

```
import random
import numpy as np
from lib.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-ipython
%load_ext autoreload
%autoreload 2

np.random.seed(1)
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

In [323]:
```python
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 = 'lib/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 **lib/classifiers/softmax.py**.

In [324]:
```
# First implement the naive softmax loss function with nested loops.
# Open the file lib/classifiers/softmax.py and implement the
# softmax_loss_naive function.

from lib.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: 2.381532
sanity check: 2.302585
```

Inline Question:

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

Your answer: *We are starting the calculation by assigning random weights to our model. This does not guarantee any chance of right prediction or expected prediction as all the class labels will have equal chance (attributing to the randomness of weights or unlearned weights) of being an outcome. As we have 10 classes and each class will have 1/10 probability for its correct prediction or being the correct class for that given sample, the softmax loss by definition is -log(probability). Hence the loss in our case would be -log(1/10)*

In [325]:
```
# 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 lib.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: 3.553219 analytic: 3.553219, relative error: 1.087524e-08
numerical: 1.326149 analytic: 1.326149, relative error: 8.562584e-08
numerical: 2.159517 analytic: 2.159517, relative error: 3.997632e-09
numerical: -0.195902 analytic: -0.195902, relative error: 9.305763e-08
numerical: 0.129356 analytic: 0.129356, relative error: 1.740311e-08
numerical: 2.230083 analytic: 2.230083, relative error: 1.211945e-08
numerical: -0.996805 analytic: -0.996805, relative error: 9.156072e-09
numerical: 1.054831 analytic: 1.054831, relative error: 3.946359e-08
numerical: -3.095895 analytic: -3.095895, relative error: 7.343672e-09
numerical: -1.426369 analytic: -1.426369, relative error: 2.471631e-08
numerical: 0.549162 analytic: 0.549162, relative error: 2.841099e-08
numerical: 2.230740 analytic: 2.230740, relative error: 5.818487e-09
numerical: -2.944565 analytic: -2.944565, relative error: 3.537883e-09
numerical: 1.103164 analytic: 1.103164, relative error: 2.845724e-09
numerical: -1.170893 analytic: -1.170893, relative error: 6.362529e-08
numerical: 0.681758 analytic: 0.681758, relative error: 1.256067e-07
numerical: -0.488536 analytic: -0.488536, relative error: 2.579783e-08
numerical: -4.027786 analytic: -4.027786, relative error: 1.389788e-08
numerical: -4.887112 analytic: -4.887112, relative error: 4.535726e-09
numerical: 4.899525 analytic: 4.899525, relative error: 1.186815e-08
```

In [326]:
```
# 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))
# print(grad_naive)
from lib.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))
# print(grad_vectorized)
# 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: 2.381532e+00 computed in 0.091284s
vectorized loss: 2.381532e+00 computed in 0.006767s
Loss difference: 0.000000
Gradient difference: 0.000000
```

In [327]:
```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You can experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy close to 0.35 on the validation set.
from lib.classifiers import Softmax
results = {}
best_val = -1
best_softmax = None
learning_rates = [1e-7, 5e-7] #[1e-5, 5e-6] #[1e-7, 5e-7]
regularization_strengths = [2.5e4, 5e4] #[5.5e5, 7e5] #[2.5e4, 5e4]

# Use the validation set to set the learning rate and regularization strength.
# This is almost identical to the validation that we used for the SVM; save
# the best trained softmax classifer in best_softmax.
grid_search=[(x,y) for x in learning_rates for y in regularization_strengths]

for lr, reg in grid_search:
    softmax=Softmax()
    softmax.train(X_train, y_train, learning_rate=lr, reg=reg, num_iters=1000)
    y_train_pred=softmax.predict(X_train)
    y_val_pred=softmax.predict(X_val)
    train_accuracy=np.mean(y_train_pred==y_train)
    val_accuracy=np.mean(y_val_pred==y_val)

    results[lr, reg] = (train_accuracy, val_accuracy)

    if val_accuracy > best_val:
        best_val=val_accuracy
        best_softmax=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' % best_val)
```

```
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.335918 val accuracy: 0.320000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.332408 val accuracy: 0.347000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.348531 val accuracy: 0.367000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.323898 val accuracy: 0.327000
best validation accuracy achieved during cross-validation: 0.367000
```

In [328]:
```
# 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 [329]:
```
# 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] - w_min) / (w_max - w_min)
    plt.imshow(wimg.astype('uint8'))
    plt.axis('off')
    plt.title(classes[i])
```