Image features 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 on the course website.

We have seen that we can achieve reasonable performance on an image classification task by training a linear classifier on the pixels of the input image. In this exercise we will show that we can improve our classification performance by training linear classifiers not on raw pixels but on features that are computed from the raw pixels.

All of your work for this exercise will be done in this notebook.

In [105]:
```
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-ipython
%load_ext autoreload
%autoreload 2
```

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

Load data

Similar to previous exercises, we will load CIFAR-10 data from disk.

In [106]:
```
from cs682.features import color_histogram_hsv, hog_feature

def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000):
    # 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]

    return X_train, y_train, X_val, y_val, X_test, y_test

# 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

X_train, y_train, X_val, y_val, X_test, y_test = get_CIFAR10_data()
```

```
Clear previously loaded data.
```

Extract Features

For each image we will compute a Histogram of Oriented Gradients (HOG) as well as a color histogram using the hue channel in HSV color space. We form our final feature vector for each image by concatenating the HOG and color histogram feature vectors.

Roughly speaking, HOG should capture the texture of the image while ignoring color information, and the color histogram represents the color of the input image while ignoring texture. As a result, we expect that using both together ought to work better than using either alone. Verifying this assumption would be a good thing to try for your interests.

The `hog_feature` and `color_histogram_hsv` functions both operate on a single image and return a feature vector for that image. The extract_features function takes a set of images and a list of feature functions and evaluates each feature function on each image, storing the results in a matrix where each column is the concatenation of all feature vectors for a single image.

In [107]:
```
from cs682.features import *

num_color_bins = 10 # Number of bins in the color histogram
feature_fns = [hog_feature, lambda img: color_histogram_hsv(img, nbin=num_color_bins)]
X_train_feats = extract_features(X_train, feature_fns, verbose=True)
X_val_feats = extract_features(X_val, feature_fns)
X_test_feats = extract_features(X_test, feature_fns)

# Preprocessing: Subtract the mean feature
mean_feat = np.mean(X_train_feats, axis=0, keepdims=True)
X_train_feats -= mean_feat
X_val_feats -= mean_feat
X_test_feats -= mean_feat

# Preprocessing: Divide by standard deviation. This ensures that each feature
# has roughly the same scale.
std_feat = np.std(X_train_feats, axis=0, keepdims=True)
X_train_feats /= std_feat
X_val_feats /= std_feat
X_test_feats /= std_feat

# Preprocessing: Add a bias dimension
X_train_feats = np.hstack([X_train_feats, np.ones((X_train_feats.shape[0], 1))])
X_val_feats = np.hstack([X_val_feats, np.ones((X_val_feats.shape[0], 1))])
X_test_feats = np.hstack([X_test_feats, np.ones((X_test_feats.shape[0], 1))])
```

```
Done extracting features for 1000 / 49000 images
Done extracting features for 2000 / 49000 images
Done extracting features for 3000 / 49000 images
Done extracting features for 4000 / 49000 images
Done extracting features for 5000 / 49000 images
Done extracting features for 6000 / 49000 images
Done extracting features for 7000 / 49000 images
Done extracting features for 8000 / 49000 images
Done extracting features for 9000 / 49000 images
Done extracting features for 10000 / 49000 images
Done extracting features for 11000 / 49000 images
Done extracting features for 12000 / 49000 images
Done extracting features for 13000 / 49000 images
Done extracting features for 14000 / 49000 images
Done extracting features for 15000 / 49000 images
Done extracting features for 16000 / 49000 images
Done extracting features for 17000 / 49000 images
Done extracting features for 18000 / 49000 images
Done extracting features for 19000 / 49000 images
Done extracting features for 20000 / 49000 images
Done extracting features for 21000 / 49000 images
Done extracting features for 22000 / 49000 images
Done extracting features for 23000 / 49000 images
Done extracting features for 24000 / 49000 images
Done extracting features for 25000 / 49000 images
Done extracting features for 26000 / 49000 images
Done extracting features for 27000 / 49000 images
Done extracting features for 28000 / 49000 images
Done extracting features for 29000 / 49000 images
Done extracting features for 30000 / 49000 images
Done extracting features for 31000 / 49000 images
Done extracting features for 32000 / 49000 images
Done extracting features for 33000 / 49000 images
Done extracting features for 34000 / 49000 images
Done extracting features for 35000 / 49000 images
Done extracting features for 36000 / 49000 images
Done extracting features for 37000 / 49000 images
Done extracting features for 38000 / 49000 images
Done extracting features for 39000 / 49000 images
Done extracting features for 40000 / 49000 images
Done extracting features for 41000 / 49000 images
Done extracting features for 42000 / 49000 images
Done extracting features for 43000 / 49000 images
Done extracting features for 44000 / 49000 images
Done extracting features for 45000 / 49000 images
Done extracting features for 46000 / 49000 images
Done extracting features for 47000 / 49000 images
Done extracting features for 48000 / 49000 images
```

Train SVM on features

Using the multiclass SVM code developed earlier in the assignment, train SVMs on top of the features extracted above; this should achieve better results than training SVMs directly on top of raw pixels.

In [130]:
```
# Use the validation set to tune the learning rate and regularization strength

from cs682.classifiers.linear_classifier import LinearSVM

learning_rates = [5e-7, 7.5e-7]
regularization_strengths = [1.5e4, 2.5e4]

results = {}
best_val = -1
best_svm = None

################################################################################
# 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 classifer in best_svm. You might also want to play          #
# with different numbers of bins in the color histogram. If you are careful    #
# you should be able to get accuracy of near 0.44 on the validation set.       #
################################################################################
for lr in learning_rates:
    for reg in regularization_strengths:
        print("Testing LR=", lr, " reg=", reg)
        svm = LinearSVM()
        svm.train(X_train_feats, y_train, learning_rate=lr, reg=reg,
                      num_iters=2000, verbose=False)

        y_train_pred = svm.predict(X_train_feats)
        tr_acc = np.mean(y_train == y_train_pred)
        print('training accuracy: %f' % tr_acc)

        y_val_pred = svm.predict(X_val_feats)
        val_acc = np.mean(y_val == y_val_pred)
        print('validation accuracy: %f' % val_acc)

        if val_acc > best_val:
            best_val = val_acc
            best_svm = svm

        results[(lr, reg)] = tr_acc, val_acc
################################################################################
#                              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)
```

```
Testing LR= 5e-07  reg= 15000.0
training accuracy: 0.409592
validation accuracy: 0.403000
Testing LR= 5e-07  reg= 25000.0
training accuracy: 0.413633
validation accuracy: 0.424000
Testing LR= 7.5e-07  reg= 15000.0
training accuracy: 0.415755
validation accuracy: 0.411000
Testing LR= 7.5e-07  reg= 25000.0
training accuracy: 0.412224
validation accuracy: 0.413000
lr 5.000000e-07 reg 1.500000e+04 train accuracy: 0.409592 val accuracy: 0.403000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.413633 val accuracy: 0.424000
lr 7.500000e-07 reg 1.500000e+04 train accuracy: 0.415755 val accuracy: 0.411000
lr 7.500000e-07 reg 2.500000e+04 train accuracy: 0.412224 val accuracy: 0.413000
best validation accuracy achieved during cross-validation: 0.424000
```

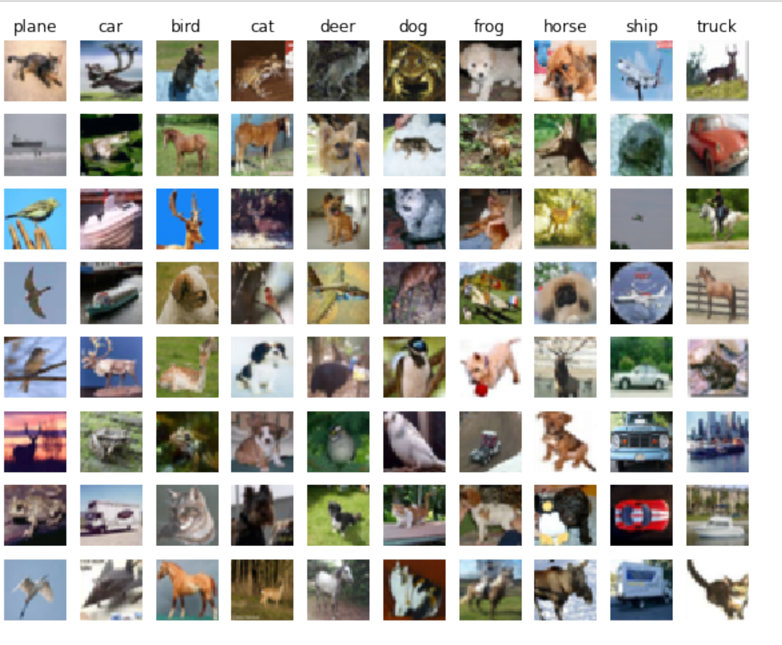
In [131]:
```
# Evaluate your trained SVM on the test set
y_test_pred = best_svm.predict(X_test_feats)
test_accuracy = np.mean(y_test == y_test_pred)
print(test_accuracy)
```

```
0.425
```

In [ ]:

In [132]:
```
# An important way to gain intuition about how an algorithm works is to
# visualize the mistakes that it makes. In this visualization, we show examples
# of images that are misclassified by our current system. The first column
# shows images that our system labeled as "plane" but whose true label is
# something other than "plane".

examples_per_class = 8
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
for cls, cls_name in enumerate(classes):
    idxs = np.where((y_test != cls) & (y_test_pred == cls))[0]
    idxs = np.random.choice(idxs, examples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt.subplot(examples_per_class, len(classes), i * len(classes) + cls + 1)
        plt.imshow(X_test[idx].astype('uint8'))
        plt.axis('off')
        if i == 0:
            plt.title(cls_name)
plt.show()
```



Inline question 1:

Describe the misclassification results that you see. Do they make sense?

While some of the misclassifications are more difficult to explain, many of them have features similar to what the class looks like which could cause them to be misclassified. For example some birds are misclassified as planes because they have similar texture features (such as outspread wings) that could be confused for a plane. Similar features like this can cause the scores for a specific class to be higher than the correct class.

Neural Network on image features

Earlier in this assigment we saw that training a two-layer neural network on raw pixels achieved better classification performance than linear classifiers on raw pixels. In this notebook we have seen that linear classifiers on image features outperform linear classifiers on raw pixels.

For completeness, we should also try training a neural network on image features. This approach should outperform all previous approaches: you should easily be able to achieve over 55% classification accuracy on the test set; our best model achieves about 60% classification accuracy.

In [30]:
```
# Preprocessing: Remove the bias dimension
# Make sure to run this cell only ONCE
print(X_train_feats.shape)
X_train_feats = X_train_feats[:, :-1]
X_val_feats = X_val_feats[:, :-1]
X_test_feats = X_test_feats[:, :-1]

print(X_train_feats.shape)
```

```
(49000, 155)
(49000, 154)
```

In [76]:
```
from cs682.classifiers.neural_net import TwoLayerNet

input_dim = X_train_feats.shape[1]
hidden_dim = 50
num_classes = 10

net = TwoLayerNet(input_dim, hidden_dim, num_classes)
best_net = None

################################################################################
# TODO: Train a two-layer neural network on image features. You may want to    #
# cross-validate various parameters as in previous sections. Store your best   #
# model in the best_net variable.                                              #
################################################################################
best_net = TwoLayerNet(input_dim, hidden_dim, num_classes)

# Train the network
stats = best_net.train(X_train_feats, y_train, X_val_feats, y_val,
        num_iters=4000, batch_size=400,
        learning_rate=0.15, learning_rate_decay=0.98,
        reg=1e-5, verbose=True)

y_train_pred = best_net.predict(X_train_feats)
tr_acc = np.mean(y_train == y_train_pred)
print('training accuracy: %f' % tr_acc)

y_val_pred = best_net.predict(X_val_feats)
val_acc = np.mean(y_val == y_val_pred)
print('validation accuracy: %f' % val_acc)

################################################################################
#                              END OF YOUR CODE                                #
################################################################################
```

```
iteration 0 / 4000: loss 2.302585
iteration 100 / 4000: loss 2.300256
iteration 200 / 4000: loss 1.867220
iteration 300 / 4000: loss 1.617228
iteration 400 / 4000: loss 1.407140
iteration 500 / 4000: loss 1.415947
iteration 600 / 4000: loss 1.359692
iteration 700 / 4000: loss 1.439941
iteration 800 / 4000: loss 1.307052
iteration 900 / 4000: loss 1.289611
iteration 1000 / 4000: loss 1.359644
iteration 1100 / 4000: loss 1.289393
iteration 1200 / 4000: loss 1.265420
iteration 1300 / 4000: loss 1.224796
iteration 1400 / 4000: loss 1.202857
iteration 1500 / 4000: loss 1.225035
iteration 1600 / 4000: loss 1.196401
iteration 1700 / 4000: loss 1.274015
iteration 1800 / 4000: loss 1.180586
iteration 1900 / 4000: loss 1.183681
iteration 2000 / 4000: loss 1.237354
iteration 2100 / 4000: loss 1.217919
iteration 2200 / 4000: loss 1.099059
iteration 2300 / 4000: loss 1.178842
iteration 2400 / 4000: loss 1.159751
iteration 2500 / 4000: loss 1.107960
iteration 2600 / 4000: loss 1.069555
iteration 2700 / 4000: loss 1.085661
iteration 2800 / 4000: loss 1.165164
iteration 2900 / 4000: loss 1.111434
iteration 3000 / 4000: loss 1.127369
iteration 3100 / 4000: loss 1.124869
iteration 3200 / 4000: loss 1.170346
iteration 3300 / 4000: loss 1.140856
iteration 3400 / 4000: loss 1.081912
iteration 3500 / 4000: loss 1.113889
iteration 3600 / 4000: loss 1.100916
iteration 3700 / 4000: loss 1.121366
iteration 3800 / 4000: loss 1.065460
iteration 3900 / 4000: loss 1.071608
training accuracy: 0.620898
validation accuracy: 0.574000
```

In [75]:
```
# Run your best neural net classifier on the test set. You should be able
# to get more than 55% accuracy.

test_acc = (best_net.predict(X_test_feats) == y_test).mean()
print(test_acc)
```

```
0.565
```