# 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 (https://compsci682-fa19.github.io/assignments2019/assignment1)](https://compsci682-fa19.github.io/assignments2019/assignment1) 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 [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-i
n-ipython
%load_ext autoreload
%autoreload 2
```

## Load data

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

In [2]:
```
from cs682.features import color_histogram_hsv, hog_feature

def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1
000):
    # 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 ma
y 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()
```

## 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 [3]:
```
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=nu
m_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 fe
ature
# 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 [4]:
```
# Use the validation set to tune the learning rate and regularization st
rength

from cs682.classifiers.linear_classifier import LinearSVM

learning_rates = [1e-9, 1e-8, 1e-7]
regularization_strengths = [5e4, 5e5, 5e6]

results = {}
best_val = -1
best_svm = None

########################################################################
########
# TODO:
#
# Use the validation set to set the learning rate and regularization str
ength. #
# This should be identical to the validation that you did for the SVM; s
ave    #
# 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 care
ful    #
# you should be able to get accuracy of near 0.44 on the validation set.
#
########################################################################
########
for learning_rate in learning_rates:
    for regularization_strength in regularization_strengths:
        svm = LinearSVM()
        lost_hist = svm.train(X_train_feats, y_train, learning_rate=lear
ning_rate, reg=regularization_strength,
                      num_iters=1500, verbose=True)
        y_train_pred = svm.predict(X_train_feats)
        y_val_pred = svm.predict(X_val_feats)
        validation_accuracy = np.mean(y_val == y_val_pred)
        training_acccuracy = np.mean(y_train == y_train_pred)
        results[(learning_rate, regularization_strength)] = (training_ac
ccuracy, validation_accuracy)
        if validation_accuracy>best_val:
            best_val = validation_accuracy
            best_svm = svm
########################################################################
########
#                              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)
```

```
iteration 0 / 1500: loss 79.166070
iteration 100 / 1500: loss 77.768600
iteration 200 / 1500: loss 76.414778
iteration 300 / 1500: loss 75.080441
iteration 400 / 1500: loss 73.779850
iteration 500 / 1500: loss 72.485710
iteration 600 / 1500: loss 71.238291
iteration 700 / 1500: loss 70.013033
iteration 800 / 1500: loss 68.795203
iteration 900 / 1500: loss 67.614803
iteration 1000 / 1500: loss 66.440817
iteration 1100 / 1500: loss 65.316504
iteration 1200 / 1500: loss 64.196338
iteration 1300 / 1500: loss 63.104636
iteration 1400 / 1500: loss 62.033931
iteration 0 / 1500: loss 755.942970
iteration 100 / 1500: loss 620.486281
iteration 200 / 1500: loss 509.609503
iteration 300 / 1500: loss 418.806658
iteration 400 / 1500: loss 344.497968
iteration 500 / 1500: loss 283.651302
iteration 600 / 1500: loss 233.848654
iteration 700 / 1500: loss 193.070441
iteration 800 / 1500: loss 159.690730
iteration 900 / 1500: loss 132.364303
iteration 1000 / 1500: loss 109.986008
iteration 1100 / 1500: loss 91.680847
iteration 1200 / 1500: loss 76.681588
iteration 1300 / 1500: loss 64.405516
iteration 1400 / 1500: loss 54.356738
iteration 0 / 1500: loss 7732.568919
iteration 100 / 1500: loss 1043.801058
iteration 200 / 1500: loss 147.643163
iteration 300 / 1500: loss 27.575384
iteration 400 / 1500: loss 11.488741
iteration 500 / 1500: loss 9.333462
iteration 600 / 1500: loss 9.044694
iteration 700 / 1500: loss 9.005975
iteration 800 / 1500: loss 9.000799
iteration 900 / 1500: loss 9.000103
iteration 1000 / 1500: loss 9.000012
iteration 1100 / 1500: loss 8.999999
iteration 1200 / 1500: loss 8.999997
iteration 1300 / 1500: loss 8.999997
iteration 1400 / 1500: loss 8.999997
iteration 0 / 1500: loss 84.055900
iteration 100 / 1500: loss 70.455940
iteration 200 / 1500: loss 59.290380
iteration 300 / 1500: loss 50.179523
iteration 400 / 1500: loss 42.711529
iteration 500 / 1500: loss 36.596346
iteration 600 / 1500: loss 31.591242
iteration 700 / 1500: loss 27.492897
iteration 800 / 1500: loss 24.142511
iteration 900 / 1500: loss 21.394813
iteration 1000 / 1500: loss 19.150993
iteration 1100 / 1500: loss 17.303275
```

```
iteration 1200 / 1500: loss 15.805308
iteration 1300 / 1500: loss 14.569930
iteration 1400 / 1500: loss 13.557524
iteration 0 / 1500: loss 814.481016
iteration 100 / 1500: loss 116.916452
iteration 200 / 1500: loss 23.458066
iteration 300 / 1500: loss 10.938091
iteration 400 / 1500: loss 9.259585
iteration 500 / 1500: loss 9.034736
iteration 600 / 1500: loss 9.004613
iteration 700 / 1500: loss 9.000584
iteration 800 / 1500: loss 9.000049
iteration 900 / 1500: loss 8.999980
iteration 1000 / 1500: loss 8.999957
iteration 1100 / 1500: loss 8.999960
iteration 1200 / 1500: loss 8.999977
iteration 1300 / 1500: loss 8.999970
iteration 1400 / 1500: loss 8.999961
iteration 0 / 1500: loss 7589.625120
iteration 100 / 1500: loss 9.000002
iteration 200 / 1500: loss 8.999996
iteration 300 / 1500: loss 8.999995
iteration 400 / 1500: loss 8.999997
iteration 500 / 1500: loss 8.999996
iteration 600 / 1500: loss 8.999997
iteration 700 / 1500: loss 8.999997
iteration 800 / 1500: loss 8.999996
iteration 900 / 1500: loss 8.999997
iteration 1000 / 1500: loss 8.999997
iteration 1100 / 1500: loss 8.999996
iteration 1200 / 1500: loss 8.999998
iteration 1300 / 1500: loss 8.999997
iteration 1400 / 1500: loss 8.999997
iteration 0 / 1500: loss 87.150589
iteration 100 / 1500: loss 19.470111
iteration 200 / 1500: loss 10.403918
iteration 300 / 1500: loss 9.188352
iteration 400 / 1500: loss 9.024850
iteration 500 / 1500: loss 9.003086
iteration 600 / 1500: loss 9.000128
iteration 700 / 1500: loss 8.999621
iteration 800 / 1500: loss 8.999666
iteration 900 / 1500: loss 8.999632
iteration 1000 / 1500: loss 8.999703
iteration 1100 / 1500: loss 8.999701
iteration 1200 / 1500: loss 8.999615
iteration 1300 / 1500: loss 8.999606
iteration 1400 / 1500: loss 8.999564
iteration 0 / 1500: loss 785.925380
iteration 100 / 1500: loss 8.999969
iteration 200 / 1500: loss 8.999969
iteration 300 / 1500: loss 8.999963
iteration 400 / 1500: loss 8.999974
iteration 500 / 1500: loss 8.999963
iteration 600 / 1500: loss 8.999961
iteration 700 / 1500: loss 8.999966
iteration 800 / 1500: loss 8.999965
```

```
iteration 900 / 1500: loss 8.999969
iteration 1000 / 1500: loss 8.999966
iteration 1100 / 1500: loss 8.999970
iteration 1200 / 1500: loss 8.999966
iteration 1300 / 1500: loss 8.999972
iteration 1400 / 1500: loss 8.999969
iteration 0 / 1500: loss 8216.190257
iteration 100 / 1500: loss 8.999999
iteration 200 / 1500: loss 8.999999
iteration 300 / 1500: loss 9.000000
iteration 400 / 1500: loss 8.999999
iteration 500 / 1500: loss 9.000000
iteration 600 / 1500: loss 8.999999
iteration 700 / 1500: loss 9.000000
iteration 800 / 1500: loss 9.000000
iteration 900 / 1500: loss 9.000000
iteration 1000 / 1500: loss 9.000000
iteration 1100 / 1500: loss 8.999999
iteration 1200 / 1500: loss 9.000001
iteration 1300 / 1500: loss 8.999999
iteration 1400 / 1500: loss 9.000000
lr 1.000000e-09 reg 5.000000e+04 train accuracy: 0.074408 val accuracy:
0.076000
lr 1.000000e-09 reg 5.000000e+05 train accuracy: 0.110980 val accuracy:
0.097000
lr 1.000000e-09 reg 5.000000e+06 train accuracy: 0.414388 val accuracy:
0.418000
lr 1.000000e-08 reg 5.000000e+04 train accuracy: 0.102388 val accuracy:
0.086000
lr 1.000000e-08 reg 5.000000e+05 train accuracy: 0.416408 val accuracy:
0.421000
lr 1.000000e-08 reg 5.000000e+06 train accuracy: 0.401163 val accuracy:
0.384000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.412755 val accuracy:
0.412000
lr 1.000000e-07 reg 5.000000e+05 train accuracy: 0.407857 val accuracy:
0.400000
lr 1.000000e-07 reg 5.000000e+06 train accuracy: 0.313816 val accuracy:
0.332000
best validation accuracy achieved during cross-validation: 0.421000
```

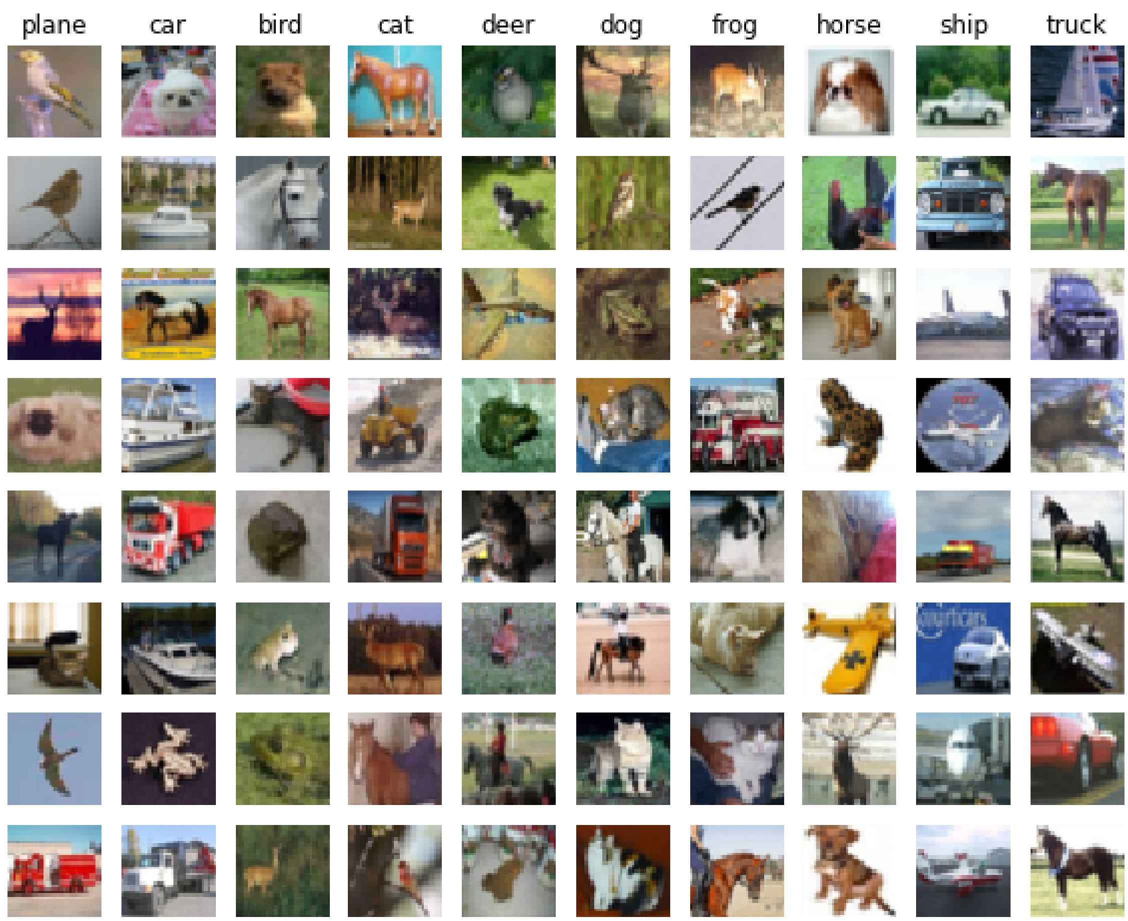
In [5]:

```
# 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)
```

```
0.422
```

In [6]:
```
# An important way to gain intuition about how an algorithm works is to
# visualize the mistakes that it makes. In this visualization, we show e
xamples
# of images that are misclassified by our current system. The first colu
mn
# shows images that our system labeled as "plane" but whose true label i
s
# 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?
Yes the misclassifications do make sense. For example, we see a couple trucks misclassified as cars and a couple cars misclassified as truks; one of the reasons for such misclassifications could be because the tires of a trucks were mapped to the tires of a car and visa versa. We see that most pictures misclassified as deer have green background, so could be that the background was mapped to the deer class' background. Similarly, we see that there is a sky in most misclassified pictures as plane, probably because the sky's features were mapped the best with the plane class for these pictures.

## 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 [7]:
```
# 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 [8]:
```
from cs682.classifiers.neural_net import TwoLayerNet

input_dim = X_train_feats.shape[1]
hidden_dim = 500
num_classes = 10

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

best_val_net = -1
learning_rates_net = [1e-1, 1e-3, 1e-5]
regularization_strengths_net = [1e-3, 1e-1, 1e1, 5e4]
batch_sizes = [100, 200, 400, 800]
results_net = {}
########################################################################
########
# 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.
#
########################################################################
########
for learning_rate in learning_rates_net:
    for regularization_strength in regularization_strengths_net:
        for batch_size in batch_sizes:
                net = TwoLayerNet(input_dim, hidden_dim, num_classes)
                net.train(X_train_feats, y_train, X_val_feats, y_val, le
arning_rate=learning_rate, reg=regularization_strength,
                          num_iters=1000, batch_size=batch_size, learnin
g_rate_decay=0.95, verbose=False)
                y_train_pred = net.predict(X_train_feats)
                y_val_pred = net.predict(X_val_feats)
                validation_accuracy = np.mean(y_val == y_val_pred)
                training_acccuracy = np.mean(y_train == y_train_pred)
                results_net[(learning_rate, regularization_strength, bat
ch_size)] = (training_acccuracy, validation_accuracy)
                if validation_accuracy > best_val_net:
                    best_val_net = validation_accuracy
                    best_net = net
########################################################################
########
#                              END OF YOUR CODE
#
########################################################################
########
```

```
/Users/satwikgoyal/Desktop/CS 682/assignment1/cs682/classifiers/neural_
net.py:98: RuntimeWarning: overflow encountered in power
  e_s = np.e**s
/Users/satwikgoyal/Desktop/CS 682/assignment1/cs682/classifiers/neural_
net.py:99: RuntimeWarning: invalid value encountered in true_divide
  p = e_s/np.sum(e_s, axis=1)[:, None]
/Users/satwikgoyal/Desktop/CS 682/assignment1/cs682/classifiers/neural_
net.py:101: RuntimeWarning: divide by zero encountered in log
  loss = np.sum(-1*np.log(p_y))
```

In [9]:

```
# Run your best neural net classifier on the test set. You should be abl
e
# to get more than 55% accuracy.

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

```
0.508
```