features

January 11, 2016

1 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 [1]:
```
import random
import numpy as np
from cs231n.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
```

In [2]:
```
# Load the CIFAR10 data
from cs231n.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 = 'cs231n/datasets/cifar-10-batches-py'
  X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

  # Subsample the data
  mask = range(num_training, num_training + num_validation)
  X_val = X_train[mask]
  y_val = y_train[mask]
  mask = range(num_training)
  X_train = X_train[mask]
  y_train = y_train[mask]
  mask = 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

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

In [3]:
```
from cs231n.features import *

# 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 the bonus section.

# 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.

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=1)
mean_feat = np.expand_dims(mean_feat, axis=1)
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=1)
std_feat = np.expand_dims(std_feat, axis=1)
X_train_feats /= std_feat
X_val_feats /= std_feat
X_test_feats /= std_feat

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

In [4]:
```
# Use the validation set to tune the learning rate and regularization strength

from cs231n.classifiers.linear_classifier import LinearSVM

learning_rates = [1e-9, 1e-8, 1e-7]
regularization_strengths = [1e5, 1e6, 1e7]

results = {}
best_val = -1
best_svm = None
```

```
        pass
        ################################################################################
        # 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_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.       #
        ################################################################################
        import itertools as it
        import copy

        from cs231n.classifiers import Softmax

        for l_r, reg in it.product(learning_rates, regularization_strengths):
            softmax = Softmax()
            loss_hist = softmax.train(X_train_feats, y_train, learning_rate=l_r, reg=reg,
                                      num_iters=1500, verbose=True)
            y_train_pred = softmax.predict(X_train_feats)
            y_val_pred = softmax.predict(X_val_feats)

            train_acc = np.mean(y_train == y_train_pred)
            val_acc = np.mean(y_val == y_val_pred)

            if val_acc > best_val:
                best_val = val_acc
                best_softmax = copy.copy(softmax)

            results[(l_r, reg)] = (train_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
```

```
iteration 0 / 1500: loss 84.605062
iteration 100 / 1500: loss 82.975561
iteration 200 / 1500: loss 81.376104
iteration 300 / 1500: loss 79.811460
iteration 400 / 1500: loss 78.278291
iteration 500 / 1500: loss 76.772528
iteration 600 / 1500: loss 75.296805
iteration 700 / 1500: loss 73.851351
iteration 800 / 1500: loss 72.433825
iteration 900 / 1500: loss 71.046388
iteration 1000 / 1500: loss 69.684982
iteration 1100 / 1500: loss 68.350376
```

```
iteration 1200 / 1500: loss 67.041678
iteration 1300 / 1500: loss 65.762103
iteration 1400 / 1500: loss 64.503973
iteration 0 / 1500: loss 795.911621
iteration 100 / 1500: loss 651.988632
iteration 200 / 1500: loss 534.168457
iteration 300 / 1500: loss 437.713174
iteration 400 / 1500: loss 358.750349
iteration 500 / 1500: loss 294.109011
iteration 600 / 1500: loss 241.188816
iteration 700 / 1500: loss 197.867086
iteration 800 / 1500: loss 162.401066
iteration 900 / 1500: loss 133.367048
iteration 1000 / 1500: loss 109.598502
iteration 1100 / 1500: loss 90.139549
iteration 1200 / 1500: loss 74.210674
iteration 1300 / 1500: loss 61.169937
iteration 1400 / 1500: loss 50.494492
iteration 0 / 1500: loss 8003.142734
iteration 100 / 1500: loss 1074.252845
iteration 200 / 1500: loss 145.922026
iteration 300 / 1500: loss 21.544730
iteration 400 / 1500: loss 4.880641
iteration 500 / 1500: loss 2.647990
iteration 600 / 1500: loss 2.348864
iteration 700 / 1500: loss 2.308786
iteration 800 / 1500: loss 2.303416
iteration 900 / 1500: loss 2.302696
iteration 1000 / 1500: loss 2.302600
iteration 1100 / 1500: loss 2.302587
iteration 1200 / 1500: loss 2.302585
iteration 1300 / 1500: loss 2.302585
iteration 1400 / 1500: loss 2.302585
iteration 0 / 1500: loss 74.532598
iteration 100 / 1500: loss 61.433957
iteration 200 / 1500: loss 50.710779
iteration 300 / 1500: loss 41.931643
iteration 400 / 1500: loss 34.745543
iteration 500 / 1500: loss 28.861503
iteration 600 / 1500: loss 24.043767
iteration 700 / 1500: loss 20.101767
iteration 800 / 1500: loss 16.873878
iteration 900 / 1500: loss 14.231274
iteration 1000 / 1500: loss 12.067886
iteration 1100 / 1500: loss 10.297451
iteration 1200 / 1500: loss 8.846902
iteration 1300 / 1500: loss 7.660462
iteration 1400 / 1500: loss 6.688471
iteration 0 / 1500: loss 718.056707
iteration 100 / 1500: loss 98.199035
iteration 200 / 1500: loss 15.150905
iteration 300 / 1500: loss 4.023942
iteration 400 / 1500: loss 2.533213
iteration 500 / 1500: loss 2.333484
```

```
iteration 600 / 1500: loss 2.306724
iteration 700 / 1500: loss 2.303140
iteration 800 / 1500: loss 2.302659
iteration 900 / 1500: loss 2.302595
iteration 1000 / 1500: loss 2.302586
iteration 1100 / 1500: loss 2.302585
iteration 1200 / 1500: loss 2.302585
iteration 1300 / 1500: loss 2.302585
iteration 1400 / 1500: loss 2.302585
iteration 0 / 1500: loss 7813.763640
iteration 100 / 1500: loss 2.302591
iteration 200 / 1500: loss 2.302585
iteration 300 / 1500: loss 2.302585
iteration 400 / 1500: loss 2.302585
iteration 500 / 1500: loss 2.302585
iteration 600 / 1500: loss 2.302585
iteration 700 / 1500: loss 2.302585
iteration 800 / 1500: loss 2.302585
iteration 900 / 1500: loss 2.302585
iteration 1000 / 1500: loss 2.302585
iteration 1100 / 1500: loss 2.302585
iteration 1200 / 1500: loss 2.302585
iteration 1300 / 1500: loss 2.302585
iteration 1400 / 1500: loss 2.302585
iteration 0 / 1500: loss 80.815814
iteration 100 / 1500: loss 12.821447
iteration 200 / 1500: loss 3.711886
iteration 300 / 1500: loss 2.491348
iteration 400 / 1500: loss 2.327878
iteration 500 / 1500: loss 2.305968
iteration 600 / 1500: loss 2.303033
iteration 700 / 1500: loss 2.302644
iteration 800 / 1500: loss 2.302590
iteration 900 / 1500: loss 2.302583
iteration 1000 / 1500: loss 2.302581
iteration 1100 / 1500: loss 2.302582
iteration 1200 / 1500: loss 2.302582
iteration 1300 / 1500: loss 2.302581
iteration 1400 / 1500: loss 2.302582
iteration 0 / 1500: loss 788.909867
iteration 100 / 1500: loss 2.302585
iteration 200 / 1500: loss 2.302585
iteration 300 / 1500: loss 2.302585
iteration 400 / 1500: loss 2.302585
iteration 500 / 1500: loss 2.302585
iteration 600 / 1500: loss 2.302585
iteration 700 / 1500: loss 2.302585
iteration 800 / 1500: loss 2.302585
iteration 900 / 1500: loss 2.302585
iteration 1000 / 1500: loss 2.302585
iteration 1100 / 1500: loss 2.302585
iteration 1200 / 1500: loss 2.302585
iteration 1300 / 1500: loss 2.302585
iteration 1400 / 1500: loss 2.302585
```

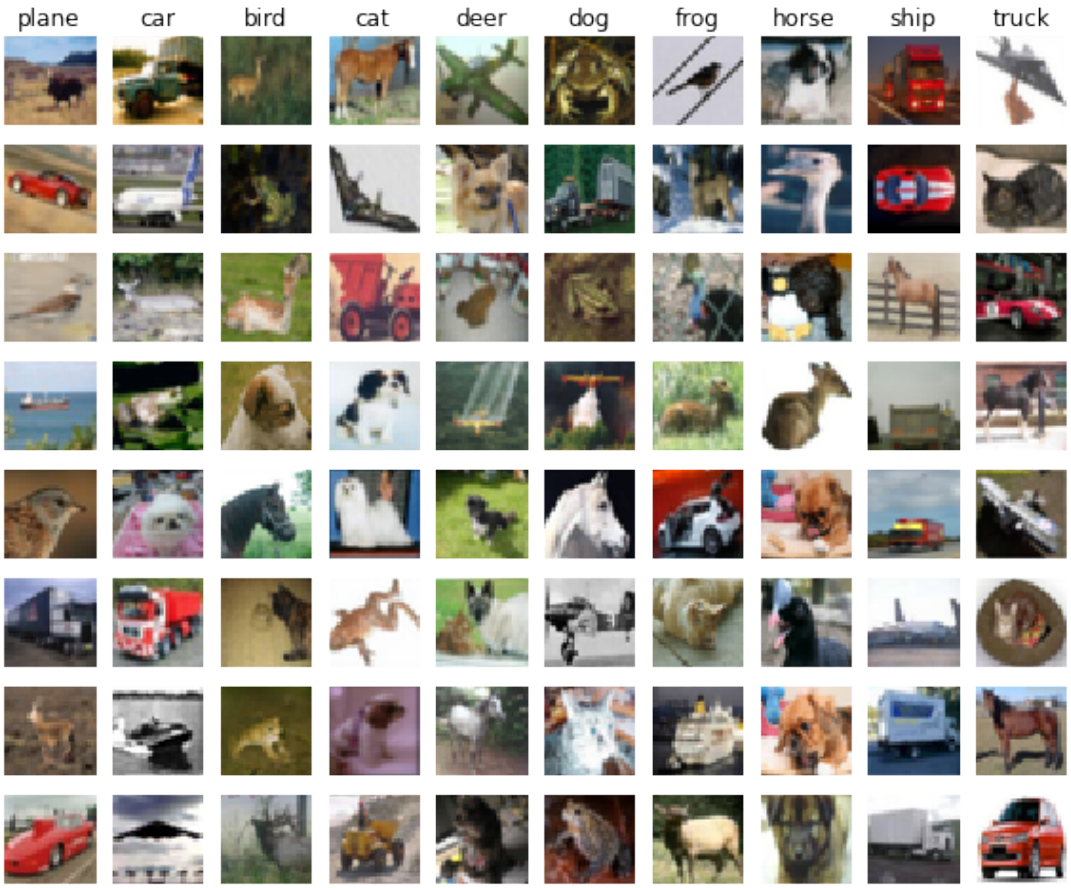
```
iteration 0 / 1500: loss 8163.719991
iteration 100 / 1500: loss 2.302585
iteration 200 / 1500: loss 2.302585
iteration 300 / 1500: loss 2.302585
iteration 400 / 1500: loss 2.302585
iteration 500 / 1500: loss 2.302585
iteration 600 / 1500: loss 2.302585
iteration 700 / 1500: loss 2.302585
iteration 800 / 1500: loss 2.302585
iteration 900 / 1500: loss 2.302585
iteration 1000 / 1500: loss 2.302585
iteration 1100 / 1500: loss 2.302585
iteration 1200 / 1500: loss 2.302585
iteration 1300 / 1500: loss 2.302585
iteration 1400 / 1500: loss 2.302585
lr 1.000000e-09 reg 1.000000e+05 train accuracy: 0.093816 val accuracy: 0.078000
lr 1.000000e-09 reg 1.000000e+06 train accuracy: 0.093612 val accuracy: 0.089000
lr 1.000000e-09 reg 1.000000e+07 train accuracy: 0.416673 val accuracy: 0.414000
lr 1.000000e-08 reg 1.000000e+05 train accuracy: 0.071082 val accuracy: 0.060000
lr 1.000000e-08 reg 1.000000e+06 train accuracy: 0.411041 val accuracy: 0.415000
lr 1.000000e-08 reg 1.000000e+07 train accuracy: 0.403633 val accuracy: 0.400000
lr 1.000000e-07 reg 1.000000e+05 train accuracy: 0.413184 val accuracy: 0.421000
lr 1.000000e-07 reg 1.000000e+06 train accuracy: 0.413714 val accuracy: 0.412000
lr 1.000000e-07 reg 1.000000e+07 train accuracy: 0.320265 val accuracy: 0.307000
best validation accuracy achieved during cross-validation: 0.421000
```

In [5]:
```
# Evaluate your classifier on the test set
y_test_pred = best_softmax.predict(X_test_feats)
test_accuracy = np.mean(y_test == y_test_pred)
print test_accuracy
```

```
0.419
```

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 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()
```



1.0.1 Inline question 1:

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

2 Bonus: Design your own features!

You have seen that simple image features can improve classification performance. So far we have tried HOG and color histograms, but other types of features may be able to achieve even better classification performance.

For bonus points, design and implement a new type of feature and use it for image classification on CIFAR-10. Explain how your feature works and why you expect it to be useful for image classification. Implement it in this notebook, cross-validate any hyperparameters, and compare its performance to the HOG + Color histogram baseline.

3 Bonus: Do something extra!

Use the material and code we have presented in this assignment to do something interesting. Was there another question we should have asked? Did any cool ideas pop into your head as you were working on the assignment? This is your chance to show off!