knn

September 28, 2018

1 k-Nearest Neighbor (kNN) 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.

The kNN classifier consists of two stages:

- During training, the classifier takes the training data and simply remembers it
- During testing, kNN classifies every test image by comparing to all training images and transfering the labels of the k most similar training examples
- The value of k is cross-validated

In this exercise you will implement these steps and understand the basic Image Classification pipeline, cross-validation, and gain proficiency in writing efficient, vectorized code.

In [1]:
```
# Run some setup code for this notebook.

import random
import numpy as np
from cs682.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

from __future__ import print_function

# This is a bit of magic to make matplotlib figures appear inline in the notebook
# rather than in a new window.
%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'

# Some more magic so that the notebook will reload external python modules;
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

In [2]:
```
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs682/datasets/cifar-10-batches-py'
```

```
        # Cleaning up variables to prevent loading data multiple times (which may cause memory
        try:
           del X_train, y_train
           del X_test, y_test
           print('Clear previously loaded data.')
        except:
           pass

        X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

        # As a sanity check, we print out the size of the training and test data.
        print('Training data shape: ', X_train.shape)
        print('Training labels shape: ', y_train.shape)
        print('Test data shape: ', X_test.shape)
        print('Test labels shape: ', y_test.shape)
```

```
Training data shape:  (50000, 32, 32, 3)
Training labels shape:  (50000,)
Test data shape:  (10000, 32, 32, 3)
Test labels shape:  (10000,)
```

In [3]:
```
        # Visualize some examples from the dataset.
        # We show a few examples of training images from each class.
        classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'tru
        num_classes = len(classes)
        samples_per_class = 7
        for y, cls in enumerate(classes):
            idxs = np.flatnonzero(y_train == y)
            idxs = np.random.choice(idxs, samples_per_class, replace=False)
            for i, idx in enumerate(idxs):
                plt_idx = i * num_classes + y + 1
                plt.subplot(samples_per_class, num_classes, plt_idx)
                plt.imshow(X_train[idx].astype('uint8'))
                plt.axis('off')
                if i == 0:
                    plt.title(cls)
        plt.show()
```

In [4]:
```
# Subsample the data for more efficient code execution in this exercise
num_training = 5000
mask = list(range(num_training))
X_train = X_train[mask]
y_train = y_train[mask]

num_test = 500
mask = list(range(num_test))
X_test = X_test[mask]
y_test = y_test[mask]
```

In [5]:
```
# Reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
print(X_train.shape, X_test.shape)
```

```
(5000, 3072) (500, 3072)
```

In [6]:
```
from cs682.classifiers import KNearestNeighbor

# Create a kNN classifier instance.
# Remember that training a kNN classifier is a noop:
# the Classifier simply remembers the data and does no further processing
classifier = KNearestNeighbor()
classifier.train(X_train, y_train)
```

We would now like to classify the test data with the kNN classifier. Recall that we can break down this process into two steps:

1. First we must compute the distances between all test examples and all train examples.
2. Given these distances, for each test example we find the k nearest examples and have them vote for the label

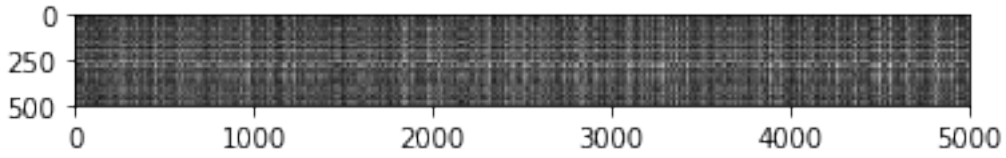
Lets begin with computing the distance matrix between all training and test examples. For example, if there are **Ntr** training examples and **Nte** test examples, this stage should result in a **Nte x Ntr** matrix where each element (i,j) is the distance between the i-th test and j-th train example.

First, open `cs682/classifiers/k_nearest_neighbor.py` and implement the function `compute_distances_two_loops` that uses a (very inefficient) double loop over all pairs of (test, train) examples and computes the distance matrix one element at a time.

```
In [7]: # Open cs682/classifiers/k_nearest_neighbor.py and implement
        # compute_distances_two_loops.
        #print(X_train)
        # Test your implementation:
        dists = classifier.compute_distances_two_loops(X_test)
        print(dists.shape)
```

```
(500, 5000)
```

```
In [8]: # We can visualize the distance matrix: each row is a single test example and
        # its distances to training examples
        plt.imshow(dists, interpolation='none')
        plt.show()
```



Inline Question #1: Notice the structured patterns in the distance matrix, where some rows or columns are visible brighter. (Note that with the default color scheme black indicates low distances while white indicates high distances.)

- What in the data is the cause behind the distinctly bright rows?
- What causes the columns?

Your Answer: The data that causes the distinctly bright rows are the data points in the train data that are far away from the test data. The columns are caused by data points on the test data that are far from the train data

```
In [9]: # Now implement the function predict_labels and run the code below:
        # We use k = 1 (which is Nearest Neighbor).
        y_test_pred = classifier.predict_labels(dists, k=1)

        # Compute and print the fraction of correctly predicted examples
        num_correct = np.sum(y_test_pred == y_test)
        accuracy = float(num_correct) / num_test
        print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
```

```
Got 137 / 500 correct => accuracy: 0.274000
```

You should expect to see approximately 27% accuracy. Now lets try out a larger `k`, say `k = 5`:

```
In [10]: y_test_pred = classifier.predict_labels(dists, k=5)
         num_correct = np.sum(y_test_pred == y_test)
         accuracy = float(num_correct) / num_test
         print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
```

```
Got 139 / 500 correct => accuracy: 0.278000
```

You should expect to see a slightly better performance than with `k = 1`.

Inline Question 2 We can also other distance metrics such as L1 distance. The performance of a Nearest Neighbor classifier that uses L1 distance will not change if (Select all that apply.): 1. The data is preprocessed by subtracting the mean. 2. The data is preprocessed by subtracting the mean and dividing by the standard deviation. 3. The coordinate axes for the data are rotated. 4. None of the above.

Your Answer: The data is preprocessed by subtracting the mean and dividing by the standard deviation

Your explanation: This is because l1 distance is by subtacting the distance between train and test data. hence, when the data is preprocessed, its posriton will change and hence the mean corresponding to them will also change. By dividing it by the standard devation, you are scaling down, hence it would effectively be the same.

```
In [11]: # Now lets speed up distance matrix computation by using partial vectorization
         # with one loop. Implement the function compute_distances_one_loop and run the
         # code below:
         dists_one = classifier.compute_distances_one_loop(X_test)

         # To ensure that our vectorized implementation is correct, we make sure that it
         # agrees with the naive implementation. There are many ways to decide whether
         # two matrices are similar; one of the simplest is the Frobenius norm. In case
         # you haven't seen it before, the Frobenius norm of two matrices is the square
         # root of the squared sum of differences of all elements; in other words, reshape
         # the matrices into vectors and compute the Euclidean distance between them.
         difference = np.linalg.norm(dists - dists_one, ord='fro')
         print('Difference was: %f' % (difference, ))
         if difference < 0.001:
```

```
            print('Good! The distance matrices are the same')
        else:
            print('Uh-oh! The distance matrices are different')
```

```
Difference was: 0.000000
Good! The distance matrices are the same
```

In [12]:
```
# Now implement the fully vectorized version inside compute_distances_no_loops
# and run the code
dists_two = classifier.compute_distances_no_loops(X_test)

# check that the distance matrix agrees with the one we computed before:
difference = np.linalg.norm(dists - dists_two, ord='fro')
print('Difference was: %f' % (difference, ))
if difference < 0.001:
    print('Good! The distance matrices are the same')
else:
    print('Uh-oh! The distance matrices are different')
```

```
Difference was: 0.000000
Good! The distance matrices are the same
```

In [13]:
```
# Let's compare how fast the implementations are
def time_function(f, *args):
    """
    Call a function f with args and return the time (in seconds) that it took to exec
    """
    import time
    tic = time.time()
    f(*args)
    toc = time.time()
    return toc - tic

two_loop_time = time_function(classifier.compute_distances_two_loops, X_test)
print('Two loop version took %f seconds' % two_loop_time)

one_loop_time = time_function(classifier.compute_distances_one_loop, X_test)
print('One loop version took %f seconds' % one_loop_time)

no_loop_time = time_function(classifier.compute_distances_no_loops, X_test)
print('No loop version took %f seconds' % no_loop_time)

# you should see significantly faster performance with the fully vectorized implement
```

```
Two loop version took 32.560608 seconds
One loop version took 36.555758 seconds
No loop version took 0.202962 seconds
```

1.0.1 Cross-validation

We have implemented the k-Nearest Neighbor classifier but we set the value k = 5 arbitrarily. We will now determine the best value of this hyperparameter with cross-validation.

In [14]:
```
num_folds = 5
k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50, 100]
#k_choices = [1, 3]
k_to_accuracies = {}
accuracy = np.zeros((10, 5), dtype=np.float)
X_train_folds = []
y_train_folds = []

X_train_folds = np.array_split(X_train,num_folds)
y_train_folds = np.array_split(y_train,num_folds)
#X_train_folds = np.array(X_train_folds)
#y_train_folds = np.array(y_train_folds)
n = X_train.shape[0]/num_folds

for i in range(len(k_choices)):
    obj = KNearestNeighbor()
    for j in range(num_folds):
        temp_train_x = np.concatenate((X_train_folds[:j]+X_train_folds[j+1:]))
        #print(temp_train_x.shape)
        temp_train_y = np.concatenate((y_train_folds[:j]+y_train_folds[j+1:]))
        temp_test_x = X_train_folds[j]
        temp_test_y = y_train_folds[j]
        obj.train(temp_train_x, temp_train_y)
        dists = obj.compute_distances_two_loops(temp_test_x)
        pred = obj.predict_labels(dists, k=k_choices[i])
        num_correct = np.sum(pred == temp_test_y)
        accuracy[i][j] = float(num_correct)/n
        k_to_accuracies[k_choices[i]] = accuracy[i]
################################################################################
# TODO:                                                                        #
# Split up the training data into folds. After splitting, X_train_folds and    #
# y_train_folds should each be lists of length num_folds, where                #
# y_train_folds[i] is the label vector for the points in X_train_folds[i].     #
# Hint: Look up the numpy array_split function.                                #
################################################################################
# Your code
################################################################################
#                                 END OF YOUR CODE                             #
################################################################################

# A dictionary holding the accuracies for different values of k that we find
# when running cross-validation. After running cross-validation,
# k_to_accuracies[k] should be a list of length num_folds giving the different
# accuracy values that we found when using that value of k.
```

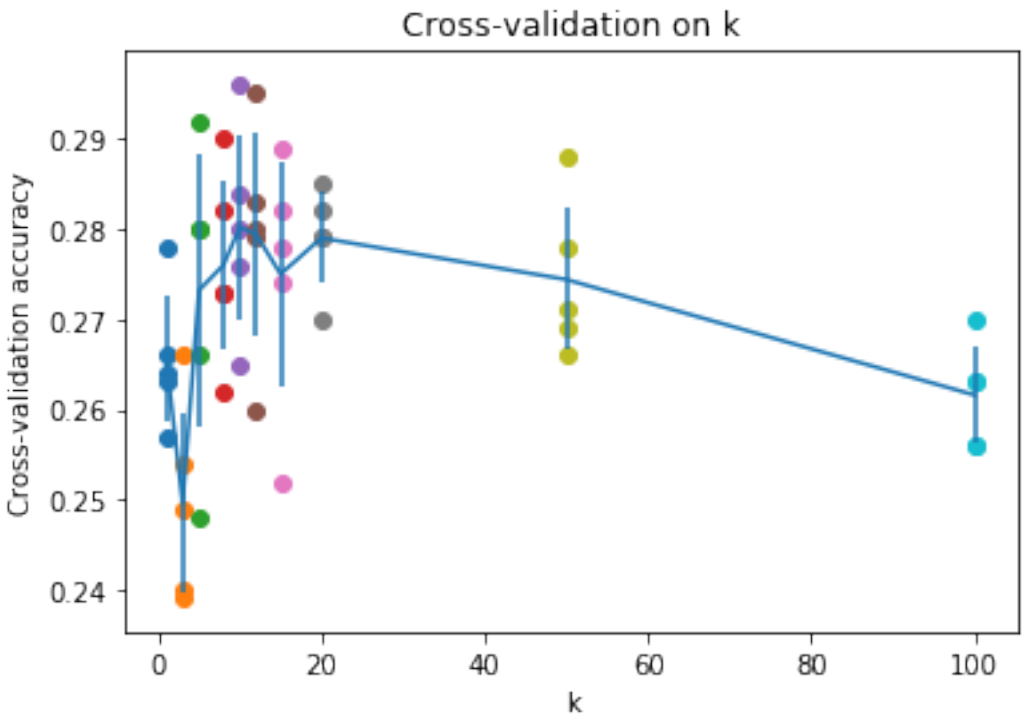
```
        ################################################################################
        # TODO:                                                                        #
        # Perform k-fold cross validation to find the best value of k. For each        #
        # possible value of k, run the k-nearest-neighbor algorithm num_folds times,   #
        # where in each case you use all but one of the folds as training data and the #
        # last fold as a validation set. Store the accuracies for all fold and all     #
        # values of k in the k_to_accuracies dictionary.                               #
        ################################################################################
        # Your code
        ################################################################################
        #                                END OF YOUR CODE                              #
        ################################################################################

        # Print out the computed accuracies
        for k in k_choices:
            for accuracy in k_to_accuracies[k]:
                print('k = %d, accuracy = %f' % (k, accuracy))
        #print(k_to_accuracies, accuracy)
```

```
k = 1, accuracy = 0.263000
k = 1, accuracy = 0.257000
k = 1, accuracy = 0.264000
k = 1, accuracy = 0.278000
k = 1, accuracy = 0.266000
k = 3, accuracy = 0.239000
k = 3, accuracy = 0.249000
k = 3, accuracy = 0.240000
k = 3, accuracy = 0.266000
k = 3, accuracy = 0.254000
k = 5, accuracy = 0.248000
k = 5, accuracy = 0.266000
k = 5, accuracy = 0.280000
k = 5, accuracy = 0.292000
k = 5, accuracy = 0.280000
k = 8, accuracy = 0.262000
k = 8, accuracy = 0.282000
k = 8, accuracy = 0.273000
k = 8, accuracy = 0.290000
k = 8, accuracy = 0.273000
k = 10, accuracy = 0.265000
k = 10, accuracy = 0.296000
k = 10, accuracy = 0.276000
k = 10, accuracy = 0.284000
k = 10, accuracy = 0.280000
k = 12, accuracy = 0.260000
```

```
k = 12, accuracy = 0.295000
k = 12, accuracy = 0.279000
k = 12, accuracy = 0.283000
k = 12, accuracy = 0.280000
k = 15, accuracy = 0.252000
k = 15, accuracy = 0.289000
k = 15, accuracy = 0.278000
k = 15, accuracy = 0.282000
k = 15, accuracy = 0.274000
k = 20, accuracy = 0.270000
k = 20, accuracy = 0.279000
k = 20, accuracy = 0.279000
k = 20, accuracy = 0.282000
k = 20, accuracy = 0.285000
k = 50, accuracy = 0.271000
k = 50, accuracy = 0.288000
k = 50, accuracy = 0.278000
k = 50, accuracy = 0.269000
k = 50, accuracy = 0.266000
k = 100, accuracy = 0.256000
k = 100, accuracy = 0.270000
k = 100, accuracy = 0.263000
k = 100, accuracy = 0.256000
k = 100, accuracy = 0.263000
```

In [15]:
```
# plot the raw observations
for k in k_choices:
    accuracies = k_to_accuracies[k]
    plt.scatter([k] * len(accuracies), accuracies)

#plot the trend line with error bars that correspond to standard deviation
accuracies_mean = np.array([np.mean(v) for k,v in sorted(k_to_accuracies.items())])
accuracies_std = np.array([np.std(v) for k,v in sorted(k_to_accuracies.items())])
plt.errorbar(k_choices, accuracies_mean, yerr=accuracies_std)
plt.title('Cross-validation on k')
plt.xlabel('k')
plt.ylabel('Cross-validation accuracy')
plt.show()
```

```
The history saving thread hit an unexpected error (OperationalError('database is locked',)).His
```




```
In [27]: # Based on the cross-validation results above, choose the best value for k,
         # retrain the classifier using all the training data, and test it on the test
         # data. You should be able to get above 28% accuracy on the test data.
         best_k = 10

         classifier = KNearestNeighbor()
         classifier.train(X_train, y_train)
         y_test_pred = classifier.predict(X_test, k=best_k)

         # Compute and display the accuracy
         num_correct = np.sum(y_test_pred == y_test)
         accuracy = float(num_correct) / num_test
         print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
```

```
Got 141 / 500 correct => accuracy: 0.282000
```

Inline Question 3 Which of the following statements about k-Nearest Neighbor (k-NN) are true in a classification setting, and for all k? Select all that apply. 1. The training error of a 1-NN will always be better than that of 5-NN. 2. The test error of a 1-NN will always be better than that of a 5-NN. 3. The decision boundary of the k-NN classifier is linear. 4. The time needed to classify a test example with the k-NN classifier grows with the size of the training set. 5. None of the above.

Your Answer: The training error of 1-NN will always be better than that of 5-NN; The time needed to classify a test example with the k-NN classifier grows with the size of the training set

Your explanation: Since it uses only 1 closest neighbour, the error is often low when compared with 5NN. higher the number of values in the dataset, it takes more time because it has to compare each test point with with more train data.

svm

September 28, 2018

1 Multiclass Support Vector Machine 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.

In this exercise you will:

- implement a fully-vectorized **loss function** for the SVM
- implement the fully-vectorized expression for its **analytic gradient**
- **check your implementation** using 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 [10]:
```
# Run some setup code for this notebook.

import random
import numpy as np
from cs682.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

from __future__ import print_function

# This is a bit of magic to make matplotlib figures appear inline in the
# notebook rather than in a new window.
%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'

# Some more magic so that the notebook will reload external python modules;
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

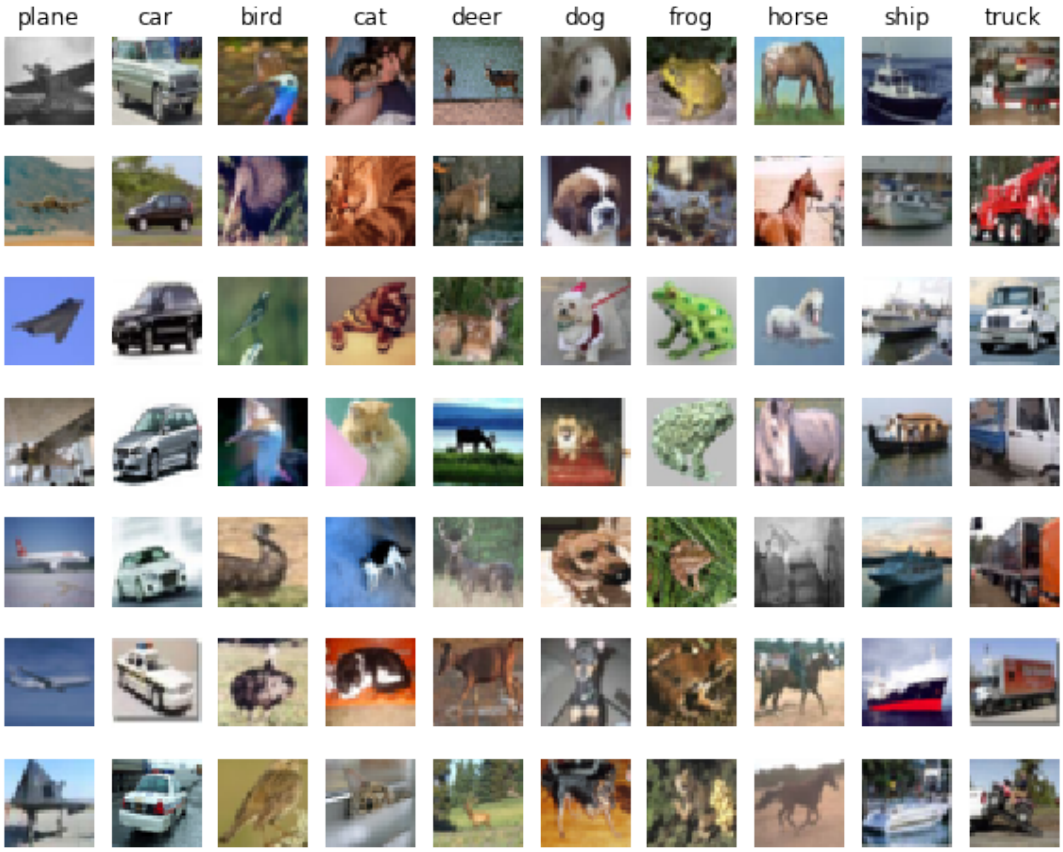
```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

1.1 CIFAR-10 Data Loading and Preprocessing

In [11]:
```
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs682/datasets/cifar-10-batches-py'

# Cleaning up variables to prevent loading data multiple times (which may cause memor
try:
   del X_train, y_train
   del X_test, y_test
   print('Clear previously loaded data.')
except:
   pass

X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

# As a sanity check, we print out the size of the training and test data.
print('Training data shape: ', X_train.shape)
print('Training labels shape: ', y_train.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

```
Clear previously loaded data.
Training data shape:  (50000, 32, 32, 3)
Training labels shape:  (50000,)
Test data shape:  (10000, 32, 32, 3)
Test labels shape:  (10000,)
```

In [12]:
```
# Visualize some examples from the dataset.
# We show a few examples of training images from each class.
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'tru
num_classes = len(classes)
samples_per_class = 7
for y, cls in enumerate(classes):
    idxs = np.flatnonzero(y_train == y)
    idxs = np.random.choice(idxs, samples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt_idx = i * num_classes + y + 1
        plt.subplot(samples_per_class, num_classes, plt_idx)
        plt.imshow(X_train[idx].astype('uint8'))
        plt.axis('off')
        if i == 0:
            plt.title(cls)
plt.show()
```



In [13]:
```
# Split the data into train, val, and test sets. In addition we will
# create a small development set as a subset of the training data;
# we can use this for development so our code runs faster.
num_training = 49000
num_validation = 1000
num_test = 1000
num_dev = 500

# Our validation set will be num_validation points from the original
# training set.
mask = range(num_training, num_training + num_validation)
X_val = X_train[mask]
y_val = y_train[mask]

# Our training set will be the first num_train points from the original
# training set.
mask = range(num_training)
X_train = X_train[mask]
y_train = y_train[mask]
```

```
        # We will also make a development set, which is a small subset of
        # the training set.
        mask = np.random.choice(num_training, num_dev, replace=False)
        X_dev = X_train[mask]
        y_dev = y_train[mask]

        # We use the first num_test points of the original test set as our
        # test set.
        mask = range(num_test)
        X_test = X_test[mask]
        y_test = y_test[mask]

        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)
```

```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

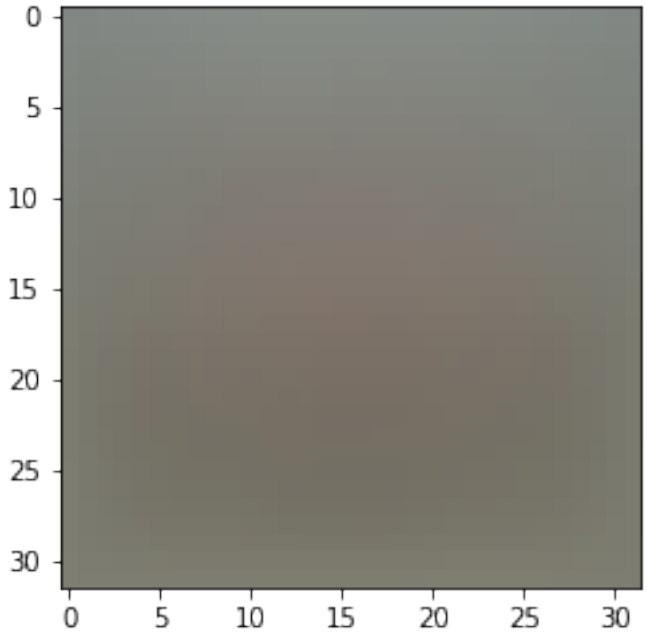
```
In [14]: # Preprocessing: reshape the image data into rows
         X_train = np.reshape(X_train, (X_train.shape[0], -1))
         X_val = np.reshape(X_val, (X_val.shape[0], -1))
         X_test = np.reshape(X_test, (X_test.shape[0], -1))
         X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

         # As a sanity check, print out the shapes of the data
         print('Training data shape: ', X_train.shape)
         print('Validation data shape: ', X_val.shape)
         print('Test data shape: ', X_test.shape)
         print('dev data shape: ', X_dev.shape)
```

```
Training data shape:  (49000, 3072)
Validation data shape:  (1000, 3072)
Test data shape:  (1000, 3072)
dev data shape:  (500, 3072)
```

```
In [15]: # Preprocessing: subtract the mean image
         # first: compute the image mean based on the training data
         mean_image = np.mean(X_train, axis=0)
```

```
print(mean_image[:10]) # print a few of the elements
plt.figure(figsize=(4,4))
plt.imshow(mean_image.reshape((32,32,3)).astype('uint8')) # visualize the mean image
plt.show()
```

```
[130.64189796 135.98173469 132.47391837 130.05569388 135.34804082
 131.75402041 130.96055102 136.14328571 132.47636735 131.48467347]
```



In [16]:
```
# second: subtract the mean image from train and test data
X_train -= mean_image
X_val -= mean_image
X_test -= mean_image
X_dev -= mean_image
```

In [17]:
```
# third: append the bias dimension of ones (i.e. bias trick) so that our SVM
# only has to worry about optimizing a single weight matrix W.
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))])

print(X_train.shape, X_val.shape, X_test.shape, X_dev.shape)
```

```
(49000, 3073) (1000, 3073) (1000, 3073) (500, 3073)
```

1.2 SVM Classifier

Your code for this section will all be written inside **cs682/classifiers/linear_svm.py**.

As you can see, we have prefilled the function `svm_loss_naive` which uses for loops to evaluate the multiclass SVM loss function.

```
In [18]: # Evaluate the naive implementation of the loss we provided for you:
         from cs682.classifiers.linear_svm import svm_loss_naive
         import time

         # generate a random SVM weight matrix of small numbers
         W = np.random.randn(3073, 10) * 0.0001

         loss, grad = svm_loss_naive(W, X_dev, y_dev, 0.000005)
         print('loss: %f' % (loss, ))
```

```
loss: 8.990574
```

The `grad` returned from the function above is right now all zero. Derive and implement the gradient for the SVM cost function and implement it inline inside the function `svm_loss_naive`. You will find it helpful to interleave your new code inside the existing function.

To check that you have correctly implemented the gradient correctly, you can numerically estimate the gradient of the loss function and compare the numeric estimate to the gradient that you computed. We have provided code that does this for you:

```
In [19]: # Once you've implemented the gradient, recompute it with the code below
         # and gradient check it with the function we provided for you

         # Compute the loss and its gradient at W.
         loss, grad = svm_loss_naive(W, X_dev, y_dev, 0.0)

         # Numerically compute the gradient along several randomly chosen dimensions, and
         # compare them with your analytically computed gradient. The numbers should match
         # almost exactly along all dimensions.
         from cs682.gradient_check import grad_check_sparse
         f = lambda w: svm_loss_naive(w, X_dev, y_dev, 0.0)[0]
         grad_numerical = grad_check_sparse(f, W, grad)

         # do the gradient check once again with regularization turned on
         # you didn't forget the regularization gradient did you?
         loss, grad = svm_loss_naive(W, X_dev, y_dev, 5e1)
         f = lambda w: svm_loss_naive(w, X_dev, y_dev, 5e1)[0]
         grad_numerical = grad_check_sparse(f, W, grad)
```

```
numerical: -24.108611 analytic: -24.108611, relative error: 2.704705e-11
numerical: 7.813560 analytic: 7.813560, relative error: 9.103830e-12
numerical: -12.520158 analytic: -12.520158, relative error: 8.184262e-12
numerical: -2.031423 analytic: -2.031423, relative error: 1.282149e-10
```

```
numerical: 4.496740 analytic: 4.496740, relative error: 5.679604e-11
numerical: -24.770518 analytic: -24.770518, relative error: 1.904929e-11
numerical: -3.636857 analytic: -3.636857, relative error: 3.462981e-11
numerical: 4.101868 analytic: 4.101868, relative error: 2.708284e-11
numerical: 6.854217 analytic: 6.854217, relative error: 1.483525e-11
numerical: 8.042819 analytic: 8.042819, relative error: 1.071404e-12
numerical: -15.222171 analytic: -15.222443, relative error: 8.958126e-06
numerical: 14.872230 analytic: 14.876024, relative error: 1.275436e-04
numerical: -0.993121 analytic: -0.990314, relative error: 1.415678e-03
numerical: 0.150445 analytic: 0.153391, relative error: 9.696562e-03
numerical: -2.875891 analytic: -2.876576, relative error: 1.191767e-04
numerical: -3.042409 analytic: -3.037801, relative error: 7.578284e-04
numerical: 18.864060 analytic: 18.870379, relative error: 1.674558e-04
numerical: 7.001854 analytic: 7.008647, relative error: 4.848467e-04
numerical: 2.096751 analytic: 2.098283, relative error: 3.652597e-04
numerical: 13.471014 analytic: 13.468233, relative error: 1.032337e-04
```

1.2.1 Inline Question 1:

It is possible that once in a while a dimension in the gradcheck will not match exactly. What could such a discrepancy be caused by? Is it a reason for concern? What is a simple example in one dimension where a gradient check could fail? How would change the margin affect of the frequency of this happening? *Hint: the SVM loss function is not strictly speaking differentiable*

Your Answer: Yeah, it is possible. The discrepancy is can happen because the level of accuracy is not small enough. This can happen with with all non differenciable functions. Example, for the mod function, which is not differenciable at the origin, hence the grad check might not match for positive or negative levels of accuracies. It will not have a huge effect on the margin.

In [32]:
```
# Next implement the function svm_loss_vectorized; for now only compute the loss;
# we will implement the gradient in a moment.
tic = time.time()
loss_naive, grad_naive = svm_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Naive loss: %e computed in %fs' % (loss_naive, toc - tic))

from cs682.classifiers.linear_svm import svm_loss_vectorized
tic = time.time()
loss_vectorized, _ = svm_loss_vectorized(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Vectorized loss: %e computed in %fs' % (loss_vectorized, toc - tic))
# The losses should match but your vectorized implementation should be much faster.
print('difference: %f' % (loss_naive - loss_vectorized))
```

```
Naive loss: 8.990574e+00 computed in 0.124924s
Vectorized loss: 8.990574e+00 computed in 0.003596s
difference: -0.000000
```

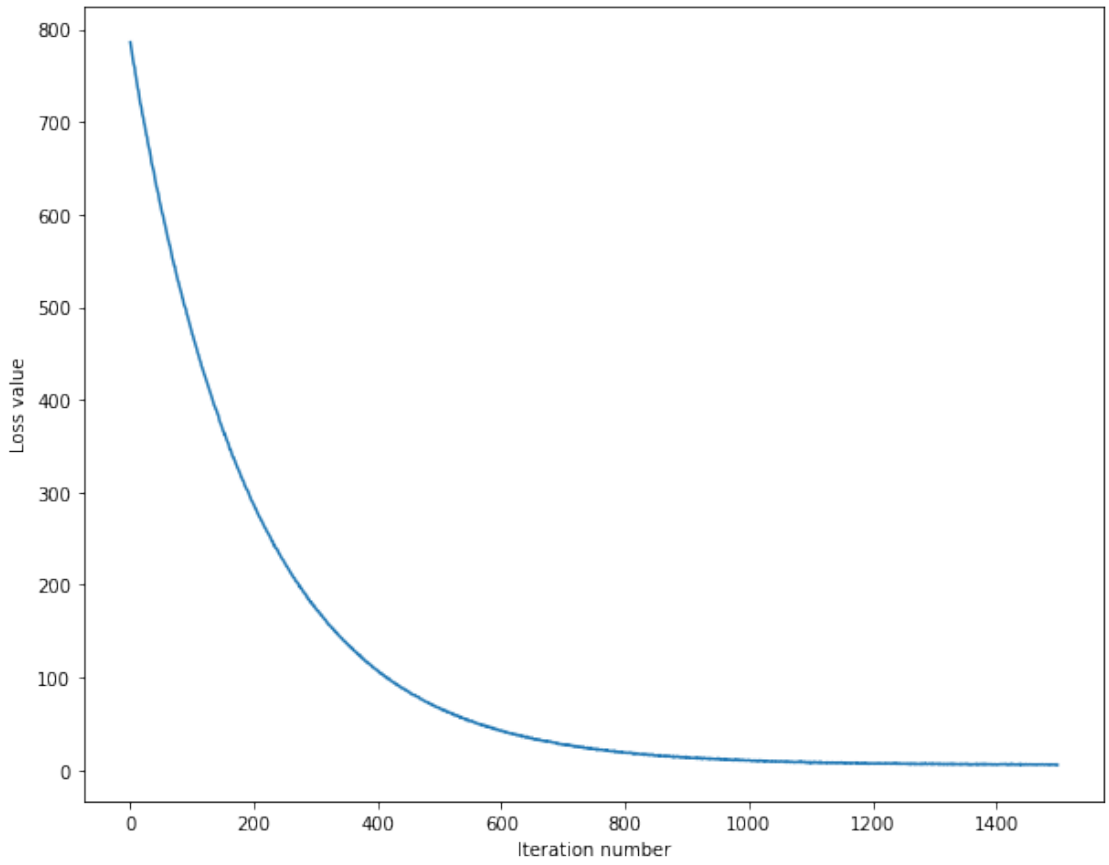
1.2.2 Stochastic Gradient Descent

We now have vectorized and efficient expressions for the loss, the gradient and our gradient matches the numerical gradient. We are therefore ready to do SGD to minimize the loss.

In [21]:
```
# In the file linear_classifier.py, implement SGD in the function
# LinearClassifier.train() and then run it with the code below.
from cs682.classifiers import LinearSVM
svm = LinearSVM()
tic = time.time()
loss_hist = svm.train(X_train, y_train, learning_rate=1e-7, reg=2.5e4,
                      num_iters=1500, verbose=True)
toc = time.time()
print('That took %fs' % (toc - tic))
```

```
iteration 0 / 1500: loss 786.205506
iteration 100 / 1500: loss 470.545603
iteration 200 / 1500: loss 285.992351
iteration 300 / 1500: loss 174.378581
iteration 400 / 1500: loss 107.180093
iteration 500 / 1500: loss 66.982101
iteration 600 / 1500: loss 42.480333
iteration 700 / 1500: loss 27.227735
iteration 800 / 1500: loss 18.838914
iteration 900 / 1500: loss 13.679232
iteration 1000 / 1500: loss 10.638101
iteration 1100 / 1500: loss 7.845622
iteration 1200 / 1500: loss 6.987930
iteration 1300 / 1500: loss 6.277957
iteration 1400 / 1500: loss 5.582518
That took 12.756051s
```

In [22]:
```
# A useful debugging strategy is to plot the loss as a function of
# iteration number:
plt.plot(loss_hist)
plt.xlabel('Iteration number')
plt.ylabel('Loss value')
plt.show()
```



In [14]:
```
# Write the LinearSVM.predict function and evaluate the performance on both the
# training and validation set
y_train_pred = svm.predict(X_train)
print('training accuracy: %f' % (np.mean(y_train == y_train_pred), ))
y_val_pred = svm.predict(X_val)
print('validation accuracy: %f' % (np.mean(y_val == y_val_pred), ))
```

```
training accuracy: 0.378714
validation accuracy: 0.385000
```

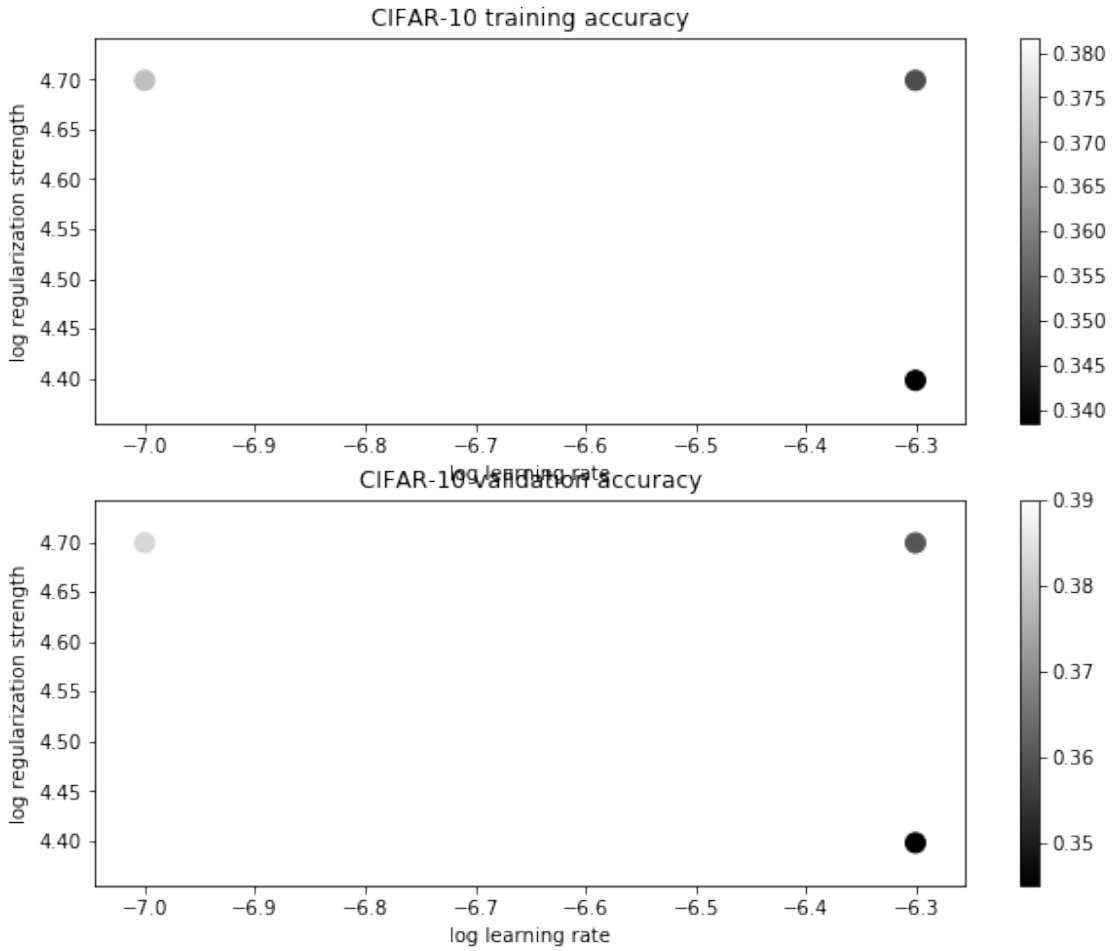
In [28]:
```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of about 0.4 on the validation set.
learning_rates = [1e-7, 5e-7]
regularization_strengths = [2.5e4, 5e4]

# results is dictionary mapping tuples of the form
# (learning_rate, regularization_strength) to tuples of the form
```

```
# (training_accuracy, validation_accuracy). The accuracy is simply the fraction
# of data points that are correctly classified.
test_accuracies =[]
results = {}
best_val = -1   # The highest validation accuracy that we have seen so far.
best_svm = None # The LinearSVM object that achieved the highest validation rate.
svm1 = LinearSVM()
for i in learning_rates:
    for j in regularization_strengths:
        loss_history = svm1.train(X_train, y_train, learning_rate=i, reg=j, num_iters=
        y_train_prediction = svm1.predict(X_train)
        accuracy_train = np.mean(y_train == y_train_prediction)
        y_val_prediction = svm1.predict(X_val)
        accuracy_validation = np.mean(y_val == y_val_prediction)
        y_test_pred = svm1.predict(X_test)
        #test_accuracy = np.mean(y_test == y_test_pred)
        #test_accuracies.append(test_accuracy)
        #print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
        results[(i,j)] = accuracy_train, accuracy_validation
        if accuracy_validation > best_val:
           best_val = accuracy_validation
           best_svm = svm1
################################################################################
# TODO:                                                                        #
# Write code that chooses the best hyperparameters by tuning on the validation #
# set. For each combination of hyperparameters, train a linear SVM on the      #
# training set, compute its accuracy on the training and validation sets, and  #
# store these numbers in the results dictionary. In addition, store the best   #
# validation accuracy in best_val and the LinearSVM object that achieves this  #
# accuracy in best_svm.                                                        #
#                                                                              #
# Hint: You should use a small value for num_iters as you develop your         #
# validation code so that the SVMs don't take much time to train; once you are #
# confident that your validation code works, you should rerun the validation   #
# code with a larger value for num_iters.                                      #
################################################################################
# Your code
################################################################################
#                              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 789.757246
iteration 100 / 1500: loss 469.809985
iteration 200 / 1500: loss 284.606484
iteration 300 / 1500: loss 173.803452
iteration 400 / 1500: loss 106.274001
iteration 500 / 1500: loss 65.857728
iteration 600 / 1500: loss 42.266374
iteration 700 / 1500: loss 27.464876
iteration 800 / 1500: loss 18.968869
iteration 900 / 1500: loss 12.858708
iteration 1000 / 1500: loss 10.190389
iteration 1100 / 1500: loss 8.758876
iteration 1200 / 1500: loss 7.240618
iteration 1300 / 1500: loss 6.274803
iteration 1400 / 1500: loss 6.261473
iteration 0 / 1500: loss 7.158331
iteration 100 / 1500: loss 6.205398
iteration 200 / 1500: loss 6.264308
iteration 300 / 1500: loss 5.280626
iteration 400 / 1500: loss 6.188082
iteration 500 / 1500: loss 5.715437
iteration 600 / 1500: loss 6.238145
iteration 700 / 1500: loss 5.628909
iteration 800 / 1500: loss 5.712633
iteration 900 / 1500: loss 5.561339
iteration 1000 / 1500: loss 5.819825
iteration 1100 / 1500: loss 5.666501
iteration 1200 / 1500: loss 5.946238
iteration 1300 / 1500: loss 6.116036
iteration 1400 / 1500: loss 5.623897
iteration 0 / 1500: loss 5.564203
iteration 100 / 1500: loss 6.003337
iteration 200 / 1500: loss 5.799800
iteration 300 / 1500: loss 6.032289
iteration 400 / 1500: loss 6.281438
iteration 500 / 1500: loss 5.563329
iteration 600 / 1500: loss 5.651315
iteration 700 / 1500: loss 6.033787
iteration 800 / 1500: loss 5.599360
iteration 900 / 1500: loss 5.710713
iteration 1000 / 1500: loss 5.737837
iteration 1100 / 1500: loss 6.367185
iteration 1200 / 1500: loss 6.165881
iteration 1300 / 1500: loss 6.173567
iteration 1400 / 1500: loss 5.797589
iteration 0 / 1500: loss 7.223745
iteration 100 / 1500: loss 6.066232
iteration 200 / 1500: loss 6.169951
```

```
iteration 300 / 1500: loss 6.311852
iteration 400 / 1500: loss 6.533727
iteration 500 / 1500: loss 5.604897
iteration 600 / 1500: loss 6.530148
iteration 700 / 1500: loss 6.613336
iteration 800 / 1500: loss 5.763047
iteration 900 / 1500: loss 6.905548
iteration 1000 / 1500: loss 6.369873
iteration 1100 / 1500: loss 6.338782
iteration 1200 / 1500: loss 6.237365
iteration 1300 / 1500: loss 6.724698
iteration 1400 / 1500: loss 6.159615
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.381673 val accuracy: 0.390000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.370898 val accuracy: 0.383000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.338469 val accuracy: 0.345000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.351327 val accuracy: 0.360000
best validation accuracy achieved during cross-validation: 0.390000
```

In [29]:
```
# Visualize the cross-validation results
import math
x_scatter = [math.log10(x[0]) for x in results]
y_scatter = [math.log10(x[1]) for x in results]

# plot training accuracy
marker_size = 100
colors = [results[x][0] for x in results]
plt.subplot(2, 1, 1)
plt.scatter(x_scatter, y_scatter, marker_size, c=colors)
plt.colorbar()
plt.xlabel('log learning rate')
plt.ylabel('log regularization strength')
plt.title('CIFAR-10 training accuracy')

# plot validation accuracy
colors = [results[x][1] for x in results] # default size of markers is 20
plt.subplot(2, 1, 2)
plt.scatter(x_scatter, y_scatter, marker_size, c=colors)
plt.colorbar()
plt.xlabel('log learning rate')
plt.ylabel('log regularization strength')
plt.title('CIFAR-10 validation accuracy')
plt.show()
```

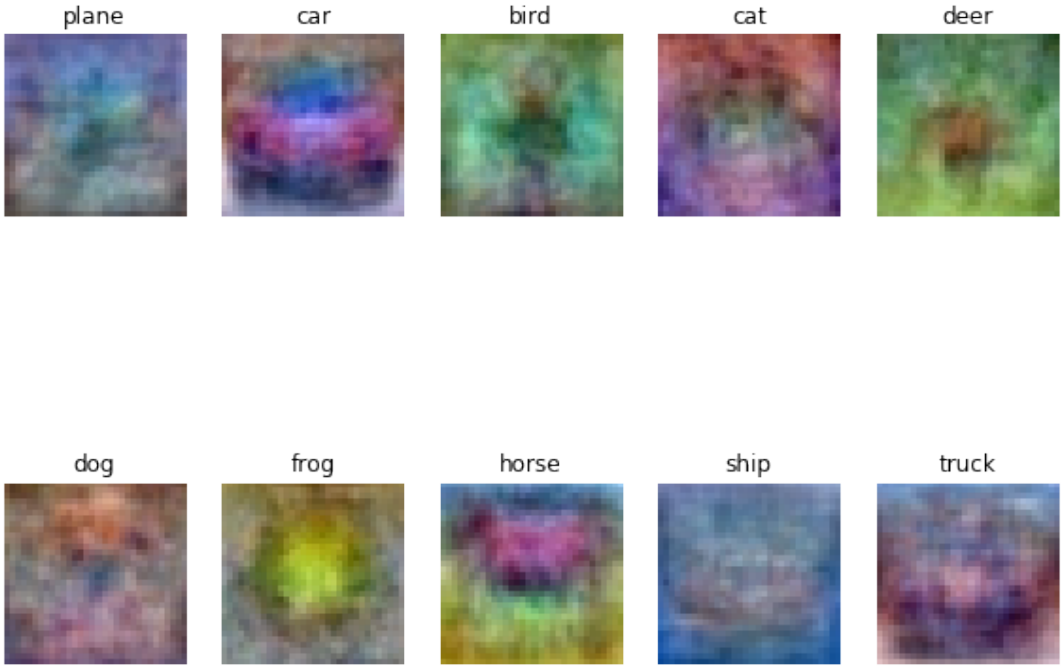


In [30]:
```
# Evaluate the best svm on test set

y_test_pred = best_svm.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
#svm2 = LinearSVM()

print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
```

```
linear SVM on raw pixels final test set accuracy: 0.360000
```

In [31]:
```
# Visualize the learned weights for each class.
# Depending on your choice of learning rate and regularization strength, these may
# or may not be nice to look at.
w = best_svm.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', 'tru
```

```
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])
```



1.2.3 Inline question 2:

Describe what your visualized SVM weights look like, and offer a brief explanation for why they look they way that they do.

Your answer: Theses svm weights are an average of the data looks. The plane has a light blue background because skies are blue and are a common background for planes. The structure of a car can be noticed for the car class. And a two headed horse structure for the horse class.

softmax

September 28, 2018

1 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 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 cs682.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
```

In [2]:
```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000, num_dev=5
    """
    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 = '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]
    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


# Cleaning up variables to prevent loading data multiple times (which may cause memory
try:
   del X_train, y_train
   del X_test, y_test
   print('Clear previously loaded data.')
except:
   pass
```

```
        # 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,)
```

1.1 Softmax Classifier

Your code for this section will all be written inside **cs682/classifiers/softmax.py**.

```
In [3]: # First implement the naive softmax loss function with nested loops.
        # Open the file cs682/classifiers/softmax.py and implement the
        # softmax_loss_naive function.

        from cs682.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: 2.355978
sanity check: 2.302585
```

1.2 Inline Question 1:

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

Your answer: The probability of each class 0.1, because there are 10 classes. Hence on an average, we can expect the loss to be -log(0.1)

In [4]:
```
# 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 cs682.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: 1.866588 analytic: 1.866588, relative error: 2.870578e-08
numerical: -0.777731 analytic: -0.777731, relative error: 4.913355e-08
numerical: 3.048122 analytic: 3.048122, relative error: 4.932950e-09
numerical: 0.378959 analytic: 0.378958, relative error: 1.656307e-07
numerical: -0.976891 analytic: -0.976891, relative error: 3.352302e-08
numerical: -1.707841 analytic: -1.707841, relative error: 1.244064e-08
numerical: -0.748268 analytic: -0.748268, relative error: 2.655352e-08
numerical: 0.381892 analytic: 0.381892, relative error: 1.218675e-07
numerical: 0.044736 analytic: 0.044736, relative error: 9.387103e-08
numerical: 0.417072 analytic: 0.417072, relative error: 3.609779e-09
numerical: 4.627194 analytic: 4.626463, relative error: 7.895356e-05
numerical: -0.706533 analytic: -0.707054, relative error: 3.688338e-04
numerical: 0.229622 analytic: 0.234915, relative error: 1.139498e-02
numerical: 0.643648 analytic: 0.642792, relative error: 6.650322e-04
numerical: -2.861869 analytic: -2.868403, relative error: 1.140199e-03
numerical: 1.028699 analytic: 1.023023, relative error: 2.766471e-03
numerical: 3.339658 analytic: 3.338466, relative error: 1.784581e-04
numerical: -1.248134 analytic: -1.256357, relative error: 3.283001e-03
numerical: -0.407527 analytic: -0.404488, relative error: 3.742294e-03
numerical: -0.759588 analytic: -0.763481, relative error: 2.555840e-03
```

In [5]:
```
# Now that we have a naive implementation of the softmax loss function and its gradien
# implement a vectorized version in softmax_loss_vectorized.
# The two versions should compute the same results, but the vectorized version should
# 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 cs682.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 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: 2.355978e+00 computed in 0.140424s
vectorized loss: 2.355978e+00 computed in 0.006468s
Loss difference: 0.000000
Gradient difference: 0.000000
```

In [6]:
```
        # Use the validation set to tune hyperparameters (regularization strength and
        # learning rate). You should experiment with different ranges for the learning
        # 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 cs682.classifiers import Softmax
        results = {}
        best_val = -1
        best_softmax = None
        learning_rates = [1e-7, 5e-7]
        regularization_strengths = [2.5e4, 5e4]

        softmax1 = Softmax()
        for i in learning_rates:
            for j in regularization_strengths:
                loss_history = softmax1.train(X_train, y_train, learning_rate=i, reg=j, num_ite
                y_train_prediction = softmax1.predict(X_train)
                accuracy_train = np.mean(y_val == y_train_prediction)
                y_val_prediction = softmax1.predict(X_val)
                accuracy_validation = np.mean(y_val == y_val_prediction)

                results[(i,j)] = accuracy_train, accuracy_validation
                if accuracy_validation > best_val:
                    best_val=accuracy_validation
                    best_softmax = softmax1
        ################################################################################
        # 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_softmax.                          #
        ################################################################################
        # Your code
        ################################################################################
```

```
        #                               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 761.533042
iteration 100 / 1500: loss 460.528496
iteration 200 / 1500: loss 279.488481
iteration 300 / 1500: loss 169.764818
iteration 400 / 1500: loss 103.437518
iteration 500 / 1500: loss 63.507726
iteration 600 / 1500: loss 39.245693
iteration 700 / 1500: loss 24.554745
iteration 800 / 1500: loss 15.727254
iteration 900 / 1500: loss 10.341743
iteration 1000 / 1500: loss 7.065417
iteration 1100 / 1500: loss 5.082982
iteration 1200 / 1500: loss 3.942867
iteration 1300 / 1500: loss 3.249523
iteration 1400 / 1500: loss 2.783591


/home/akhila/.local/lib/python3.6/site-packages/ipykernel_launcher.py:17: DeprecationWarning: c


iteration 0 / 1500: loss 3.140750
iteration 100 / 1500: loss 2.529964
iteration 200 / 1500: loss 2.324758
iteration 300 / 1500: loss 2.236533
iteration 400 / 1500: loss 2.181742
iteration 500 / 1500: loss 2.148546
iteration 600 / 1500: loss 2.224669
iteration 700 / 1500: loss 2.113984
iteration 800 / 1500: loss 2.085940
iteration 900 / 1500: loss 2.176372
iteration 1000 / 1500: loss 2.196403
iteration 1100 / 1500: loss 2.122962
iteration 1200 / 1500: loss 2.182554
iteration 1300 / 1500: loss 2.224660
iteration 1400 / 1500: loss 2.150173
iteration 0 / 1500: loss 2.113960
iteration 100 / 1500: loss 2.193093
```

```
iteration 200 / 1500: loss 2.140415
iteration 300 / 1500: loss 2.194579
iteration 400 / 1500: loss 2.105710
iteration 500 / 1500: loss 2.055973
iteration 600 / 1500: loss 2.085644
iteration 700 / 1500: loss 2.152067
iteration 800 / 1500: loss 2.146090
iteration 900 / 1500: loss 2.145500
iteration 1000 / 1500: loss 2.063160
iteration 1100 / 1500: loss 2.007901
iteration 1200 / 1500: loss 2.152323
iteration 1300 / 1500: loss 2.196219
iteration 1400 / 1500: loss 2.050577
iteration 0 / 1500: loss 2.344331
iteration 100 / 1500: loss 2.176658
iteration 200 / 1500: loss 2.134563
iteration 300 / 1500: loss 2.111681
iteration 400 / 1500: loss 2.143278
iteration 500 / 1500: loss 2.215188
iteration 600 / 1500: loss 2.184355
iteration 700 / 1500: loss 2.197539
iteration 800 / 1500: loss 2.186359
iteration 900 / 1500: loss 2.190039
iteration 1000 / 1500: loss 2.212629
iteration 1100 / 1500: loss 2.184437
iteration 1200 / 1500: loss 2.244856
iteration 1300 / 1500: loss 2.225584
iteration 1400 / 1500: loss 2.160992
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.000000 val accuracy: 0.358000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.000000 val accuracy: 0.342000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.000000 val accuracy: 0.356000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.000000 val accuracy: 0.336000
best validation accuracy achieved during cross-validation: 0.358000
```

```
In [7]: # 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: 0.328000
```

Inline Question - *True or False*

It's possible to add a new datapoint to a training set that would leave the SVM loss unchanged, but this is not the case with the Softmax classifier loss.

Your answer:True

Your explanation: In SVM, the decision boundary is not significantly effected by one training point. For softmax loss, the exponential of a small value can cause large changes in loss.

```
In [8]: # 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', 'tru
        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])
```

two_layer_net

September 28, 2018

1 Implementing a Neural Network

In this exercise we will develop a neural network with fully-connected layers to perform classification, and test it out on the CIFAR-10 dataset.

In [1]:
```
# A bit of setup

import numpy as np
import matplotlib.pyplot as plt

from cs682.classifiers.neural_net import TwoLayerNet

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 external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2

def rel_error(x, y):
    """ returns relative error """
    return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

We will use the class `TwoLayerNet` in the file `cs682/classifiers/neural_net.py` to represent instances of our network. The network parameters are stored in the instance variable `self.params` where keys are string parameter names and values are numpy arrays. Below, we initialize toy data and a toy model that we will use to develop your implementation.

In [2]:
```
# Create a small net and some toy data to check your implementations.
# Note that we set the random seed for repeatable experiments.

input_size = 4
```

```
hidden_size = 10
num_classes = 3
num_inputs = 5

def init_toy_model():
    np.random.seed(0)
    return TwoLayerNet(input_size, hidden_size, num_classes, std=1e-1)

def init_toy_data():
    np.random.seed(1)
    X = 10 * np.random.randn(num_inputs, input_size)
    y = np.array([0, 1, 2, 2, 1])
    return X, y

net = init_toy_model()
X, y = init_toy_data()
```

2 Forward pass: compute scores

Open the file `cs682/classifiers/neural_net.py` and look at the method `TwoLayerNet.loss`. This function is very similar to the loss functions you have written for the SVM and Softmax exercises: It takes the data and weights and computes the class scores, the loss, and the gradients on the parameters.

Implement the first part of the forward pass which uses the weights and biases to compute the scores for all inputs.

```
In [3]: scores = net.loss(X)
        print('Your scores:')
        print(scores)
        print()
        print('correct scores:')
        correct_scores = np.asarray([
          [-0.81233741, -1.27654624, -0.70335995],
          [-0.17129677, -1.18803311, -0.47310444],
          [-0.51590475, -1.01354314, -0.8504215 ],
          [-0.15419291, -0.48629638, -0.52901952],
          [-0.00618733, -0.12435261, -0.15226949]])
        print(correct_scores)
        print()

        # The difference should be very small. We get < 1e-7
        print('Difference between your scores and correct scores:')
        print(np.sum(np.abs(scores - correct_scores)))
```

```
Your scores:
[[-0.81233741 -1.27654624 -0.70335995]
 [-0.17129677 -1.18803311 -0.47310444]
 [-0.51590475 -1.01354314 -0.8504215 ]
```

```
 [-0.15419291 -0.48629638 -0.52901952]
 [-0.00618733 -0.12435261 -0.15226949]]

correct scores:
[[-0.81233741 -1.27654624 -0.70335995]
 [-0.17129677 -1.18803311 -0.47310444]
 [-0.51590475 -1.01354314 -0.8504215 ]
 [-0.15419291 -0.48629638 -0.52901952]
 [-0.00618733 -0.12435261 -0.15226949]]

Difference between your scores and correct scores:
3.6802720496109664e-08
```

3 Forward pass: compute loss

In the same function, implement the second part that computes the data and regularizaion loss.

```
In [4]: loss, _ = net.loss(X, y, reg=0.05)
        correct_loss = 1.30378789133

        # should be very small, we get < 1e-12
        print('Difference between your loss and correct loss:')
        print(np.sum(np.abs(loss - correct_loss)))
```

```
Difference between your loss and correct loss:
1.794120407794253e-13
```

4 Backward pass

Implement the rest of the function. This will compute the gradient of the loss with respect to the variables `W1`, `b1`, `W2`, and `b2`. Now that you (hopefully!) have a correctly implemented forward pass, you can debug your backward pass using a numeric gradient check:

```
In [5]: from cs682.gradient_check import eval_numerical_gradient

        # Use numeric gradient checking to check your implementation of the backward pass.
        # If your implementation is correct, the difference between the numeric and
        # analytic gradients should be less than 1e-8 for each of W1, W2, b1, and b2.

        loss, grads = net.loss(X, y, reg=0.05)

        # these should all be less than 1e-8 or so
        for param_name in grads:
            f = lambda W: net.loss(X, y, reg=0.05)[0]
            param_grad_num = eval_numerical_gradient(f, net.params[param_name], verbose=False)
            print('%s max relative error: %e' % (param_name, rel_error(param_grad_num, grads[pa
```

```
W1 max relative error: 3.561318e-09
W2 max relative error: 3.440708e-09
b1 max relative error: 2.738422e-09
b2 max relative error: 3.865028e-11
```

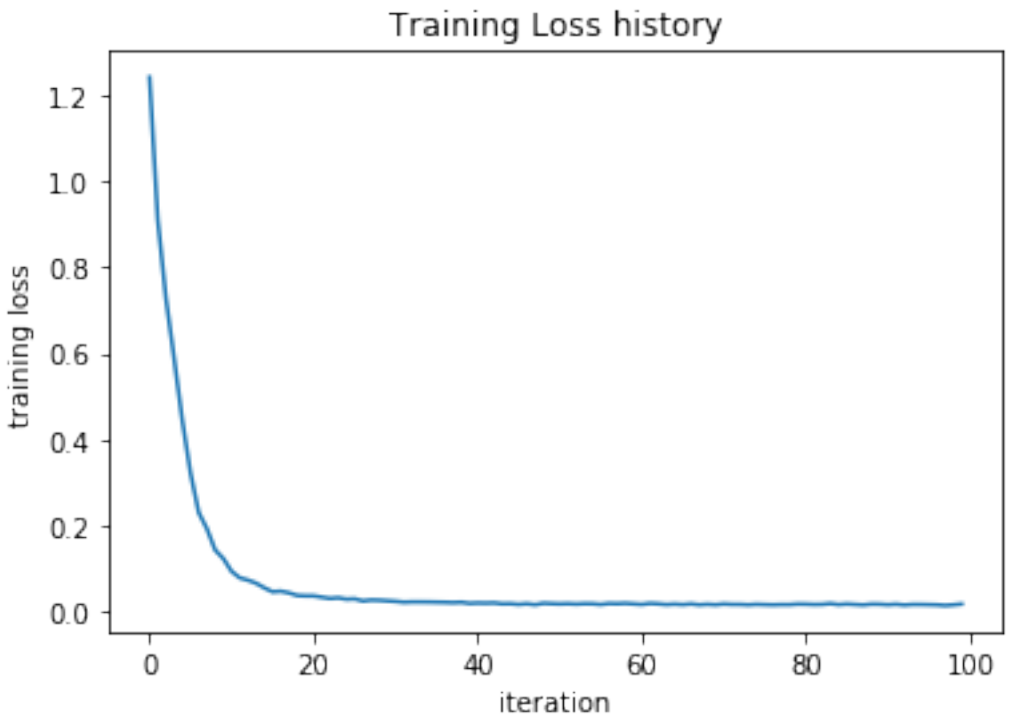
5 Train the network

To train the network we will use stochastic gradient descent (SGD), similar to the SVM and Softmax classifiers. Look at the function `TwoLayerNet.train` and fill in the missing sections to implement the training procedure. This should be very similar to the training procedure you used for the SVM and Softmax classifiers. You will also have to implement `TwoLayerNet.predict`, as the training process periodically performs prediction to keep track of accuracy over time while the network trains.

Once you have implemented the method, run the code below to train a two-layer network on toy data. You should achieve a training loss less than 0.2.

```
In [6]: net = init_toy_model()
        stats = net.train(X, y, X, y,
                    learning_rate=1e-1, reg=5e-6,
                    num_iters=100, verbose=False)

        print('Final training loss: ', stats['loss_history'][-1])

        # plot the loss history
        plt.plot(stats['loss_history'])
        plt.xlabel('iteration')
        plt.ylabel('training loss')
        plt.title('Training Loss history')
        plt.show()
```

```
Final training loss:  0.017149607938732023
```



6 Load the data

Now that you have implemented a two-layer network that passes gradient checks and works on toy data, it's time to load up our favorite CIFAR-10 data so we can use it to train a classifier on a real dataset.

```
In [7]: from cs682.data_utils import load_CIFAR10

        def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000):
            """
            Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
            it for the two-layer neural net 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 = '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]

    # 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

    # Reshape data to rows
    X_train = X_train.reshape(num_training, -1)
    X_val = X_val.reshape(num_validation, -1)
    X_test = X_test.reshape(num_test, -1)

    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
try:
   del X_train, y_train
   del X_test, y_test
   print('Clear previously loaded data.')
except:
   pass

# Invoke the above function to get our data.
X_train, y_train, X_val, y_val, X_test, y_test = 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)
```

```
Train data shape:  (49000, 3072)
Train labels shape:  (49000,)
Validation data shape:  (1000, 3072)
Validation labels shape:  (1000,)
Test data shape:  (1000, 3072)
Test labels shape:  (1000,)
```

7 Train a network

To train our network we will use SGD. In addition, we will adjust the learning rate with an exponential learning rate schedule as optimization proceeds; after each epoch, we will reduce the learning rate by multiplying it by a decay rate.

In [8]:
```
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
net = TwoLayerNet(input_size, hidden_size, num_classes)

# Train the network
stats = net.train(X_train, y_train, X_val, y_val,
            num_iters=1000, batch_size=200,
            learning_rate=1e-4, learning_rate_decay=0.95,
            reg=0.25, verbose=True)

# Predict on the validation set
val_acc = (net.predict(X_val) == y_val).mean()
print('Validation accuracy: ', val_acc)
```

```
iteration 0 / 1000: loss 2.302954
iteration 100 / 1000: loss 2.302550
iteration 200 / 1000: loss 2.297648
iteration 300 / 1000: loss 2.259602
iteration 400 / 1000: loss 2.204170
iteration 500 / 1000: loss 2.118565
iteration 600 / 1000: loss 2.051535
iteration 700 / 1000: loss 1.988466
iteration 800 / 1000: loss 2.006591
iteration 900 / 1000: loss 1.951473
Validation accuracy:  0.287
```

8 Debug the training

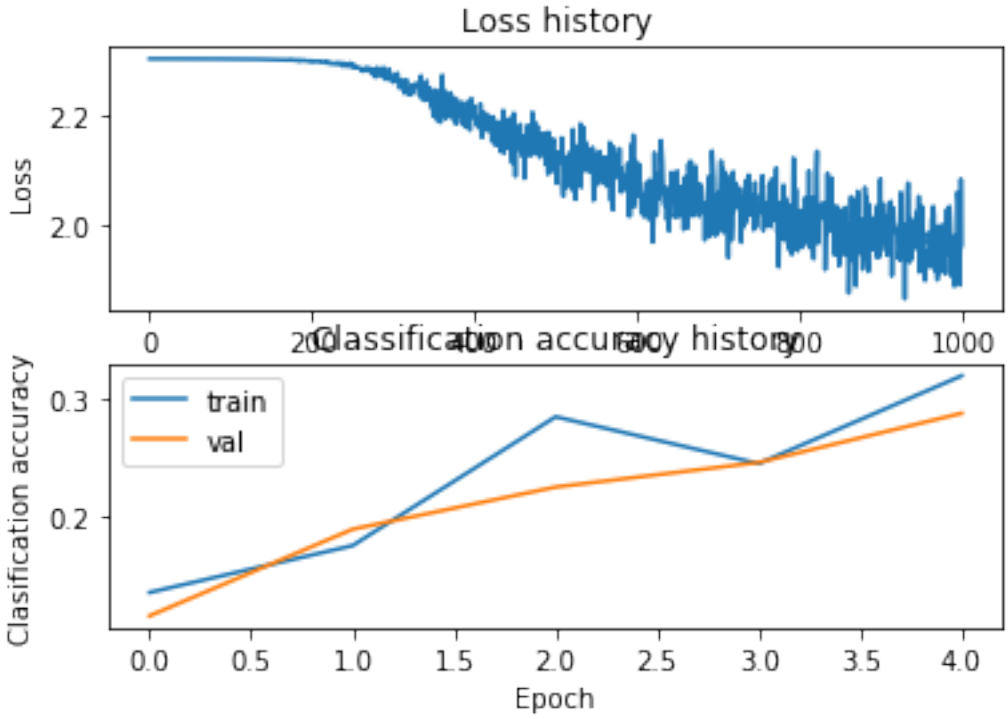
With the default parameters we provided above, you should get a validation accuracy of about 0.29 on the validation set. This isn't very good.

One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

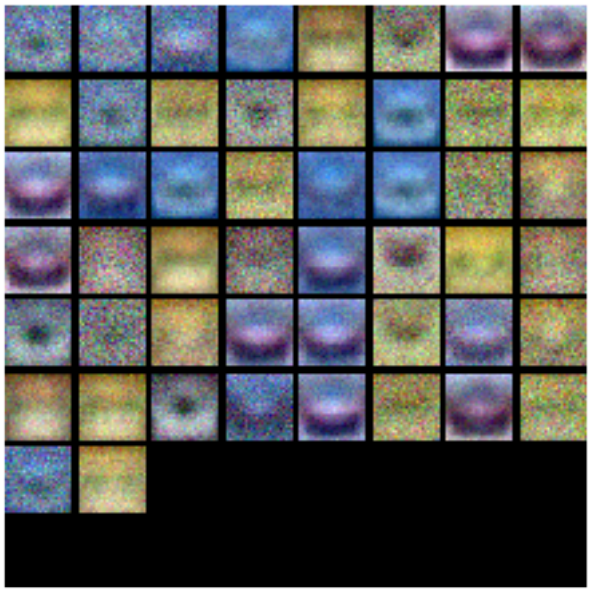
Another strategy is to visualize the weights that were learned in the first layer of the network. In most neural networks trained on visual data, the first layer weights typically show some visible structure when visualized.

In [9]:
```
# Plot the loss function and train / validation accuracies
plt.subplot(2, 1, 1)
plt.plot(stats['loss_history'])
plt.title('Loss history')
```

```
plt.xlabel('Iteration')
plt.ylabel('Loss')

plt.subplot(2, 1, 2)
plt.plot(stats['train_acc_history'], label='train')
plt.plot(stats['val_acc_history'], label='val')
plt.title('Classification accuracy history')
plt.xlabel('Epoch')
plt.ylabel('Clasification accuracy')
plt.legend()
plt.show()
```



In [10]:
```
from cs682.vis_utils import visualize_grid

# Visualize the weights of the network

def show_net_weights(net):
    W1 = net.params['W1']
    W1 = W1.reshape(32, 32, 3, -1).transpose(3, 0, 1, 2)
    plt.imshow(visualize_grid(W1, padding=3).astype('uint8'))
    plt.gca().axis('off')
    plt.show()

show_net_weights(net)
```



9 Tune your hyperparameters

What's wrong?. Looking at the visualizations above, we see that the loss is decreasing more or less linearly, which seems to suggest that the learning rate may be too low. Moreover, there is no gap between the training and validation accuracy, suggesting that the model we used has low capacity, and that we should increase its size. On the other hand, with a very large model we would expect to see more overfitting, which would manifest itself as a very large gap between the training and validation accuracy.

Tuning. Tuning the hyperparameters and developing intuition for how they affect the final performance is a large part of using Neural Networks, so we want you to get a lot of practice. Below, you should experiment with different values of the various hyperparameters, including hidden layer size, learning rate, numer of training epochs, and regularization strength. You might also consider tuning the learning rate decay, but you should be able to get good performance using the default value.

Approximate results. You should be aim to achieve a classification accuracy of greater than 48% on the validation set. Our best network gets over 52% on the validation set.

Experiment: You goal in this exercise is to get as good of a result on CIFAR-10 as you can, with a fully-connected Neural Network. Feel free implement your own techniques (e.g. PCA to reduce dimensionality, or adding dropout, or adding features to the solver, etc.).

```
In [11]: #old_settings = np.seterr(all='ignore')
         best_net = None # store the best model into this
         learning_rates = [1e-3, 6e-3,4e-4,  7e-4]
         regularization = [0.5, 0.3, 0.6, 0.8]
         hidden_layer = [100, 20, 40, 200, 400]
```

```
best_accuracy = -1
#training_epochs = []
parameters = {}
for i in learning_rates:
    for j in regularization:
        for k in hidden_layer:
            net1 = TwoLayerNet(32*32*3, k, 10)
            stats = net.train(X_train, y_train, X_val, y_val, num_iters=1000, batch_s
                              learning_rate_decay=0.95, reg=j, verbose=False)
            #print(stats)
            temp = stats['val_acc_history'][-1]
            print('learning rates: %e, regularization: %e, hidden: %e, accuracy is: %
                    % (i, j, k, temp))
            parameters[(i, j, k)] = temp

################################################################################
# TODO: Tune hyperparameters using the validation set. Store your best trained  #
# model in best_net.                                                            #
#                                                                               #
# To help debug your network, it may help to use visualizations similar to the  #
# ones we used above; these visualizations will have significant qualitative    #
# differences from the ones we saw above for the poorly tuned network.          #
#                                                                               #
# Tweaking hyperparameters by hand can be fun, but you might find it useful to  #
# write code to sweep through possible combinations of hyperparameters          #
# automatically like we did on the previous exercises.                          #
################################################################################
# Your code
################################################################################
#                              END OF YOUR CODE                                 #
################################################################################
maximum =[]
maximum =max(parameters, key = parameters.get)
best_net = TwoLayerNet(32*32*3, maximum[2], 10)
stats = best_net.train(X_train, y_train, X_val, y_val, num_iters=1000, batch_size=200
                       learning_rate_decay=0.95, reg=maximum[1], verbose=False)
```

```
learning rates: 1.000000e-03, regularization: 5.000000e-01, hidden: 1.000000e+02, accuracy is:
learning rates: 1.000000e-03, regularization: 5.000000e-01, hidden: 2.000000e+01, accuracy is:
learning rates: 1.000000e-03, regularization: 5.000000e-01, hidden: 4.000000e+01, accuracy is:
learning rates: 1.000000e-03, regularization: 5.000000e-01, hidden: 2.000000e+02, accuracy is:
learning rates: 1.000000e-03, regularization: 5.000000e-01, hidden: 4.000000e+02, accuracy is:
learning rates: 1.000000e-03, regularization: 3.000000e-01, hidden: 1.000000e+02, accuracy is:
learning rates: 1.000000e-03, regularization: 3.000000e-01, hidden: 2.000000e+01, accuracy is:
learning rates: 1.000000e-03, regularization: 3.000000e-01, hidden: 4.000000e+01, accuracy is:
learning rates: 1.000000e-03, regularization: 3.000000e-01, hidden: 2.000000e+02, accuracy is:
learning rates: 1.000000e-03, regularization: 3.000000e-01, hidden: 4.000000e+02, accuracy is:
learning rates: 1.000000e-03, regularization: 6.000000e-01, hidden: 1.000000e+02, accuracy is:
```

```
learning rates: 1.000000e-03, regularization: 6.000000e-01, hidden: 2.000000e+01, accuracy is:
learning rates: 1.000000e-03, regularization: 6.000000e-01, hidden: 4.000000e+01, accuracy is:
learning rates: 1.000000e-03, regularization: 6.000000e-01, hidden: 2.000000e+02, accuracy is:
learning rates: 1.000000e-03, regularization: 6.000000e-01, hidden: 4.000000e+02, accuracy is:
learning rates: 1.000000e-03, regularization: 8.000000e-01, hidden: 1.000000e+02, accuracy is:
learning rates: 1.000000e-03, regularization: 8.000000e-01, hidden: 2.000000e+01, accuracy is:
learning rates: 1.000000e-03, regularization: 8.000000e-01, hidden: 4.000000e+01, accuracy is:
learning rates: 1.000000e-03, regularization: 8.000000e-01, hidden: 2.000000e+02, accuracy is:
learning rates: 1.000000e-03, regularization: 8.000000e-01, hidden: 4.000000e+02, accuracy is:


/home/akhila/Desktop/COLLEGE_STUFF/Neural networks/assignment1/cs682/classifiers/neural_net.py
  temp_loss = np.sum(-np.log(soft_score[range(num_train), y]))
/home/akhila/Desktop/COLLEGE_STUFF/Neural networks/assignment1/cs682/classifiers/neural_net.py
  soft_score = np.true_divide(numerator,denominator)


learning rates: 6.000000e-03, regularization: 5.000000e-01, hidden: 1.000000e+02, accuracy is:
learning rates: 6.000000e-03, regularization: 5.000000e-01, hidden: 2.000000e+01, accuracy is:
learning rates: 6.000000e-03, regularization: 5.000000e-01, hidden: 4.000000e+01, accuracy is:
learning rates: 6.000000e-03, regularization: 5.000000e-01, hidden: 2.000000e+02, accuracy is:
learning rates: 6.000000e-03, regularization: 5.000000e-01, hidden: 4.000000e+02, accuracy is:
learning rates: 6.000000e-03, regularization: 3.000000e-01, hidden: 1.000000e+02, accuracy is:
learning rates: 6.000000e-03, regularization: 3.000000e-01, hidden: 2.000000e+01, accuracy is:
learning rates: 6.000000e-03, regularization: 3.000000e-01, hidden: 4.000000e+01, accuracy is:
learning rates: 6.000000e-03, regularization: 3.000000e-01, hidden: 2.000000e+02, accuracy is:
learning rates: 6.000000e-03, regularization: 3.000000e-01, hidden: 4.000000e+02, accuracy is:
learning rates: 6.000000e-03, regularization: 6.000000e-01, hidden: 1.000000e+02, accuracy is:
learning rates: 6.000000e-03, regularization: 6.000000e-01, hidden: 2.000000e+01, accuracy is:
learning rates: 6.000000e-03, regularization: 6.000000e-01, hidden: 4.000000e+01, accuracy is:
learning rates: 6.000000e-03, regularization: 6.000000e-01, hidden: 2.000000e+02, accuracy is:
learning rates: 6.000000e-03, regularization: 6.000000e-01, hidden: 4.000000e+02, accuracy is:
learning rates: 6.000000e-03, regularization: 8.000000e-01, hidden: 1.000000e+02, accuracy is:
learning rates: 6.000000e-03, regularization: 8.000000e-01, hidden: 2.000000e+01, accuracy is:
learning rates: 6.000000e-03, regularization: 8.000000e-01, hidden: 4.000000e+01, accuracy is:
learning rates: 6.000000e-03, regularization: 8.000000e-01, hidden: 2.000000e+02, accuracy is:
learning rates: 6.000000e-03, regularization: 8.000000e-01, hidden: 4.000000e+02, accuracy is:
learning rates: 4.000000e-04, regularization: 5.000000e-01, hidden: 1.000000e+02, accuracy is:
learning rates: 4.000000e-04, regularization: 5.000000e-01, hidden: 2.000000e+01, accuracy is:
learning rates: 4.000000e-04, regularization: 5.000000e-01, hidden: 4.000000e+01, accuracy is:
learning rates: 4.000000e-04, regularization: 5.000000e-01, hidden: 2.000000e+02, accuracy is:
learning rates: 4.000000e-04, regularization: 5.000000e-01, hidden: 4.000000e+02, accuracy is:
learning rates: 4.000000e-04, regularization: 3.000000e-01, hidden: 1.000000e+02, accuracy is:
learning rates: 4.000000e-04, regularization: 3.000000e-01, hidden: 2.000000e+01, accuracy is:
learning rates: 4.000000e-04, regularization: 3.000000e-01, hidden: 4.000000e+01, accuracy is:
learning rates: 4.000000e-04, regularization: 3.000000e-01, hidden: 2.000000e+02, accuracy is:
learning rates: 4.000000e-04, regularization: 3.000000e-01, hidden: 4.000000e+02, accuracy is:
learning rates: 4.000000e-04, regularization: 6.000000e-01, hidden: 1.000000e+02, accuracy is:
```

```
learning rates: 4.000000e-04, regularization: 6.000000e-01, hidden: 2.000000e+01, accuracy is:
learning rates: 4.000000e-04, regularization: 6.000000e-01, hidden: 4.000000e+01, accuracy is:
learning rates: 4.000000e-04, regularization: 6.000000e-01, hidden: 2.000000e+02, accuracy is:
learning rates: 4.000000e-04, regularization: 6.000000e-01, hidden: 4.000000e+02, accuracy is:
learning rates: 4.000000e-04, regularization: 8.000000e-01, hidden: 1.000000e+02, accuracy is:
learning rates: 4.000000e-04, regularization: 8.000000e-01, hidden: 2.000000e+01, accuracy is:
learning rates: 4.000000e-04, regularization: 8.000000e-01, hidden: 4.000000e+01, accuracy is:
learning rates: 4.000000e-04, regularization: 8.000000e-01, hidden: 2.000000e+02, accuracy is:
learning rates: 4.000000e-04, regularization: 8.000000e-01, hidden: 4.000000e+02, accuracy is:
learning rates: 7.000000e-04, regularization: 5.000000e-01, hidden: 1.000000e+02, accuracy is:
learning rates: 7.000000e-04, regularization: 5.000000e-01, hidden: 2.000000e+01, accuracy is:
learning rates: 7.000000e-04, regularization: 5.000000e-01, hidden: 4.000000e+01, accuracy is:
learning rates: 7.000000e-04, regularization: 5.000000e-01, hidden: 2.000000e+02, accuracy is:
learning rates: 7.000000e-04, regularization: 5.000000e-01, hidden: 4.000000e+02, accuracy is:
learning rates: 7.000000e-04, regularization: 3.000000e-01, hidden: 1.000000e+02, accuracy is:
learning rates: 7.000000e-04, regularization: 3.000000e-01, hidden: 2.000000e+01, accuracy is:
learning rates: 7.000000e-04, regularization: 3.000000e-01, hidden: 4.000000e+01, accuracy is:
learning rates: 7.000000e-04, regularization: 3.000000e-01, hidden: 2.000000e+02, accuracy is:
learning rates: 7.000000e-04, regularization: 3.000000e-01, hidden: 4.000000e+02, accuracy is:
learning rates: 7.000000e-04, regularization: 6.000000e-01, hidden: 1.000000e+02, accuracy is:
learning rates: 7.000000e-04, regularization: 6.000000e-01, hidden: 2.000000e+01, accuracy is:
learning rates: 7.000000e-04, regularization: 6.000000e-01, hidden: 4.000000e+01, accuracy is:
learning rates: 7.000000e-04, regularization: 6.000000e-01, hidden: 2.000000e+02, accuracy is:
learning rates: 7.000000e-04, regularization: 6.000000e-01, hidden: 4.000000e+02, accuracy is:
learning rates: 7.000000e-04, regularization: 8.000000e-01, hidden: 1.000000e+02, accuracy is:
learning rates: 7.000000e-04, regularization: 8.000000e-01, hidden: 2.000000e+01, accuracy is:
learning rates: 7.000000e-04, regularization: 8.000000e-01, hidden: 4.000000e+01, accuracy is:
learning rates: 7.000000e-04, regularization: 8.000000e-01, hidden: 2.000000e+02, accuracy is:
learning rates: 7.000000e-04, regularization: 8.000000e-01, hidden: 4.000000e+02, accuracy is:
```

In [12]:
```
# visualize the weights of the best network

show_net_weights(best_net)
```

10 Run on the test set

When you are done experimenting, you should evaluate your final trained network on the test set; you should get above 48%.

```
In [13]: test_acc = (best_net.predict(X_test) == y_test).mean()
         print('Test accuracy: ', test_acc)

Test accuracy:  0.448
```

Inline Question

Now that you have trained a Neural Network classifier, you may find that your testing accuracy is much lower than the training accuracy. In what ways can we decrease this gap? Select all that apply. 1. Train on a larger dataset. 2. Add more hidden units. 3. Increase the regularization strength. 4. None of the above.

Your answer: Train on a larger dataset, add more hidden units and increase regularization strength.

Your explanation: Training on a larger set helps build a better model. Adding more hidden units can learn a more complex model to fit the data better. Increasing the regularizarion strength will build a more general model

features

September 28, 2018

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 [6]: import random
        import numpy as np
        from cs682.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
```

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

1.1 Load data

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

```
In [7]: 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
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.
```

1.2 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 [8]: 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
```

1.3 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 [9]:
```
# Use the validation set to tune the learning rate and regularization strength

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

svm1 = LinearSVM()
for i in learning_rates:
    for j in regularization_strengths:
        loss_history = svm1.train(X_train_feats, y_train, learning_rate=i, reg=j, num_
        y_train_predict = svm1.predict(X_train_feats)
```

```
            train_accuracy = np.mean(y_train_predict == y_train)
            y_validation_predict = svm1.predict(X_val_feats)
            val_accuracy = np.mean(y_validation_predict == y_val)
            results[(i, j)] = (train_accuracy, val_accuracy)

            if val_accuracy > best_val:
                best_val = val_accuracy
                best_svm = svm1
        ################################################################################
        # 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.       #
        ################################################################################
        ################################################################################
        #                              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)
```

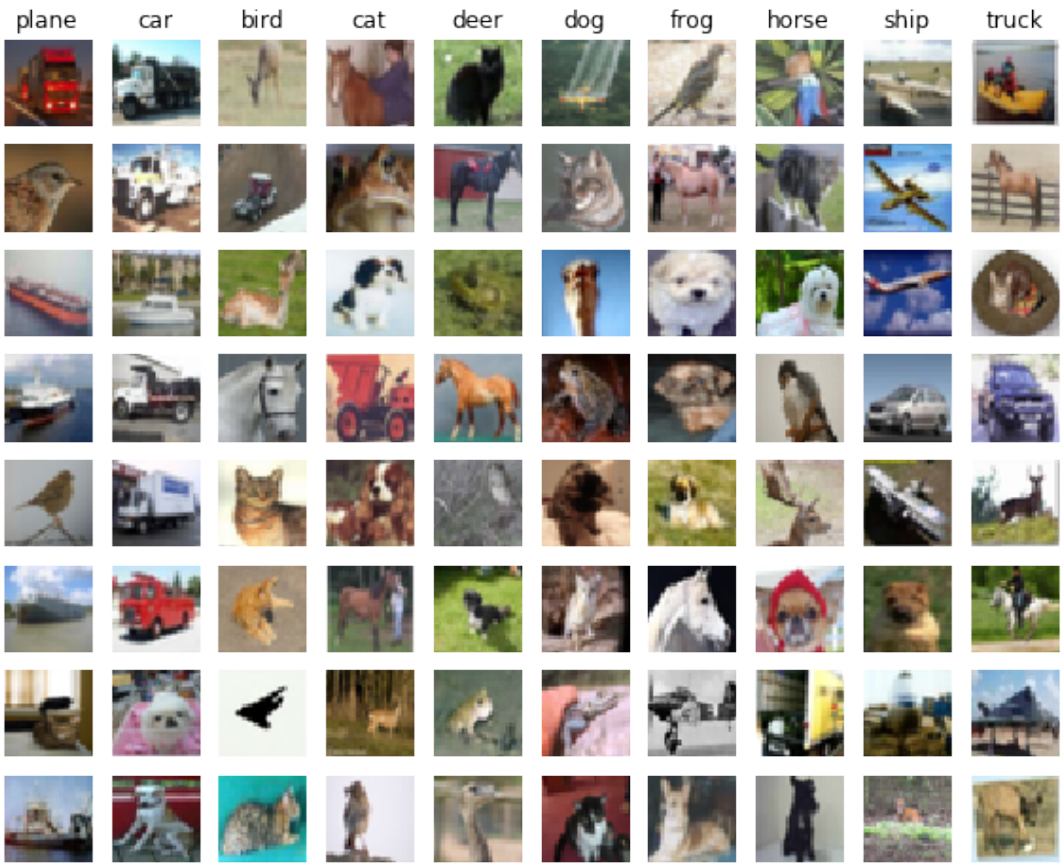
```
lr 1.000000e-09 reg 5.000000e+04 train accuracy: 0.090122 val accuracy: 0.083000
lr 1.000000e-09 reg 5.000000e+05 train accuracy: 0.090429 val accuracy: 0.083000
lr 1.000000e-09 reg 5.000000e+06 train accuracy: 0.188204 val accuracy: 0.193000
lr 1.000000e-08 reg 5.000000e+04 train accuracy: 0.415367 val accuracy: 0.417000
lr 1.000000e-08 reg 5.000000e+05 train accuracy: 0.415776 val accuracy: 0.413000
lr 1.000000e-08 reg 5.000000e+06 train accuracy: 0.404551 val accuracy: 0.395000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.415694 val accuracy: 0.420000
lr 1.000000e-07 reg 5.000000e+05 train accuracy: 0.408673 val accuracy: 0.390000
lr 1.000000e-07 reg 5.000000e+06 train accuracy: 0.370102 val accuracy: 0.357000
best validation accuracy achieved during cross-validation: 0.420000
```

In [10]:
```
# 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)
```

```
0.36
```

In [11]:
```
# 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', 'tru
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()
```



1.3.1 Inline question 1:

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

Yeah, they make sense at some level. Objects close to to each other share similar features and can be misclassified. Like cat is tagged as a dog, and a truck as a car.

1.4 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 [12]:
```
# 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 [13]:
```
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

learning_rates = [1]
regularization = [4e-4]
hidden_layer = [700]
best_accuracy = -1
#training_epochs = []
#parameters = {}
for i in learning_rates:
    for j in regularization:
        for k in hidden_layer:

            stats = net.train(X_train_feats, y_train, X_val_feats, y_val, num_iters=1
                              learning_rate_decay=0.95, reg=j, verbose=False)
```

```
                    #print(stats)
                    temp = stats['val_acc_history'][-1]
                    print('learning rates: %e, regularization: %e, hidden: %e, accuracy is: %
                            % (i, j, k, temp))
                    if temp > best_accuracy:
                        best_accuracy = temp
                        best_net = 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.                                              #
        ################################################################################
        # Your code
        ################################################################################
        #                              END OF YOUR CODE                                #
        ################################################################################
```

```
learning rates: 1.000000e+00, regularization: 4.000000e-04, hidden: 7.000000e+02, accuracy is:
```

In [14]:
```
# 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)
```

```
0.559
```