knn_My

May 20, 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.

My: Mount your Google Drive (two next cells):

```
In [3]: !apt-get install -y -qq software-properties-common python-software-properties module-i
        !add-apt-repository -y ppa:alessandro-strada/ppa 2>&1 > /dev/null
        !apt-get update -qq 2>&1 > /dev/null
        !apt-get -y install -qq google-drive-ocamlfuse fuse
        from google.colab import auth
        auth.authenticate_user()
        from oauth2client.client import GoogleCredentials
        creds = GoogleCredentials.get_application_default()
        import getpass
        !google-drive-ocamlfuse -headless -id={creds.client_id} -secret={creds.client_secret} 
        vcode = getpass.getpass()
        !echo {vcode} | google-drive-ocamlfuse -headless -id={creds.client_id} -secret={creds.

Please, open the following URL in a web browser: https://accounts.google.com/o/oauth2/auth?cli

ůůůůůůůůůů
Please, open the following URL in a web browser: https://accounts.google.com/o/oauth2/auth?cli
Please enter the verification code: Access token retrieved correctly.


In [0]: !mkdir -p drive
        !google-drive-ocamlfuse drive
```

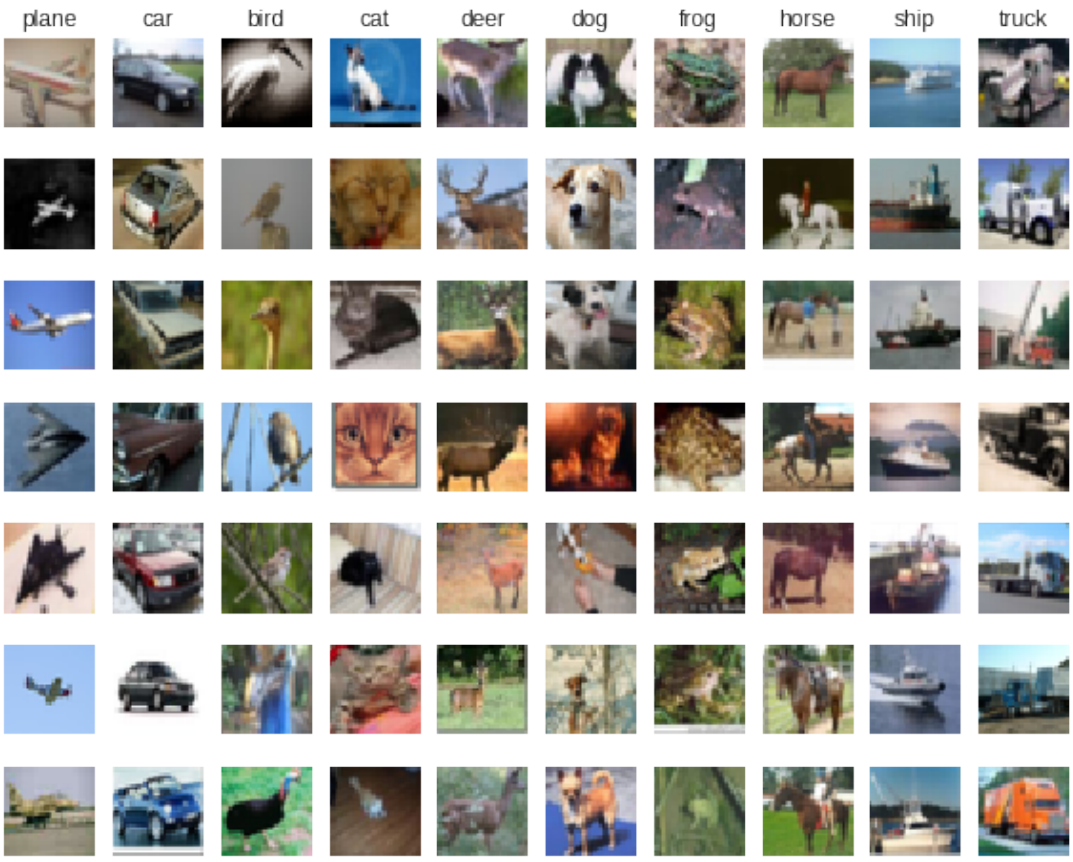
In [0]:
```
import os
os.chdir('drive/Colab Notebooks/cs231n/assignment1')
```

In [0]:
```
# Run some setup code for this notebook.

import random
import numpy as np
from cs231n.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 [10]:
```
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs231n/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)
```

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

In [11]:
```
# 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 [0]:
```
# 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 [13]:
```
# 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 [0]:
```
from cs231n.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

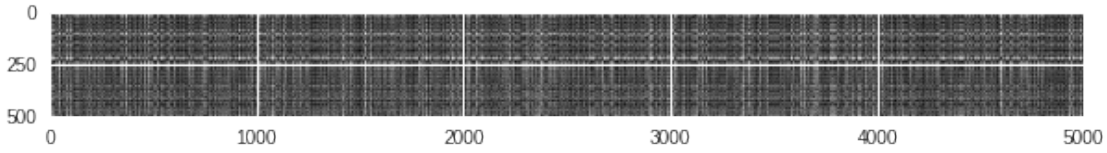
Lets begin with computing the distance matrix between all training and test examples. For example, if there are **Ntr** training examples and **Nte** test examples, this stage should result in a **Nte x Ntr** matrix where each element (i,j) is the distance between the i-th test and j-th train example.

First, open `cs231n/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 [15]:
```
# Open cs231n/classifiers/k_nearest_neighbor.py and implement
# compute_distances_two_loops.

# Test your implementation:
dists = classifier.compute_distances_two_loops(X_test)
print(dists.shape)
```

```
(500, 5000)
```

```
In [16]: # 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: *- I think that distinctly dright rows correspond to test images that very different from all train images - I think that distinctly dright the columns correspond to train images that very different from all test images*

```
In [17]: # 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 [18]: 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: 3

Your explanation: L1 distance will be change after rotate, because the line of equal distance is a square rhombus, not a circle.

In [19]:
```
# 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 [20]:
```
%%time
# 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
CPU times: user 1.21 s, sys: 57 ms, total: 1.27 s
Wall time: 685 ms
```

In [21]:
```
# 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 39.720455 seconds
One loop version took 39.818146 seconds
No loop version took 0.659355 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 [22]:
```
num_folds = 5
k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50, 100]

X_train_folds = []
y_train_folds = []
################################################################################
# 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.                                #
################################################################################
X_train_folds = np.array_split(X_train, num_folds)
y_train_folds = np.array_split(y_train, num_folds)
################################################################################
#                                 END OF YOUR CODE                             #
################################################################################
```

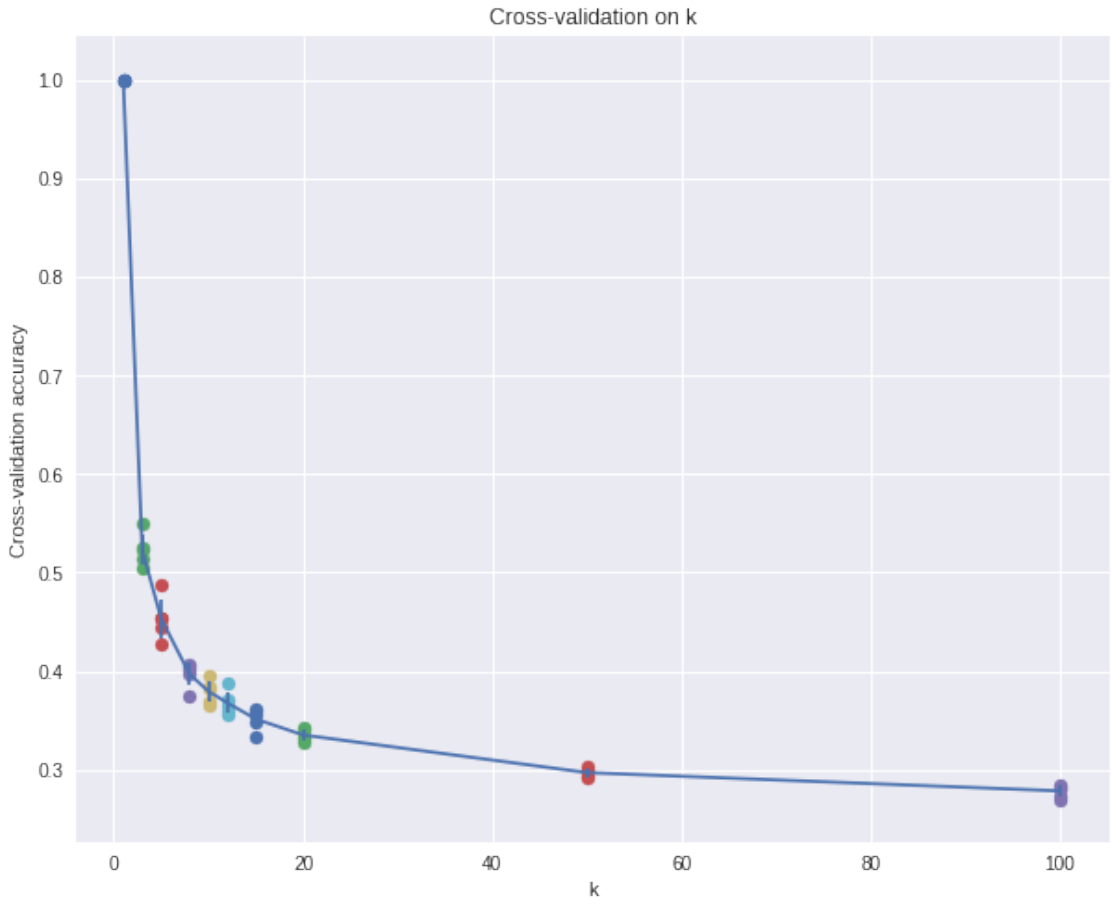
```
# A dictionary holding the accuracies for different values of k that we find
# when running cross-validation. After running cross-validation,
# k_to_accuracies[k] should be a list of length num_folds giving the different
# accuracy values that we found when using that value of k.
k_to_accuracies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


################################################################################
# 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.                               #
################################################################################

def list_merge_toarray(list_of_lists):
    _list = []
    for item in list_of_lists:
        _list.extend(item)
    return np.array(_list)

for k in k_choices:
    k_to_accuracies[k] = []
    for i in range(num_folds):
        classifier_cv = KNearestNeighbor()

        X_train_cv = X_train_folds.copy()
        y_train_cv = y_train_folds.copy()
        X_valid = X_train_cv.pop(i)
        y_valid = y_train_cv.pop(i)
        X_train_cv = list_merge_toarray(X_train_cv)
        y_train_cv = list_merge_toarray(y_train_cv)

        num_valid = len(y_valid)

        classifier_cv.train(X_train_cv, y_train_cv)
        y_valid_pred = classifier.predict(X_valid, k=k)
        num_correct = np.sum(y_valid_pred == y_valid)
        accuracy = float(num_correct) / num_valid
        k_to_accuracies[k].append(accuracy)
################################################################################
#                                END OF YOUR CODE                              #
################################################################################

# Print out the computed accuracies
for k in sorted(k_to_accuracies):
    for accuracy in k_to_accuracies[k]:
```

```
                print('k = %d, accuracy = %f' % (k, accuracy))
```

```
k = 1, accuracy = 1.000000
k = 1, accuracy = 1.000000
k = 1, accuracy = 1.000000
k = 1, accuracy = 1.000000
k = 1, accuracy = 1.000000
k = 3, accuracy = 0.504000
k = 3, accuracy = 0.514000
k = 3, accuracy = 0.525000
k = 3, accuracy = 0.523000
k = 3, accuracy = 0.550000
k = 5, accuracy = 0.428000
k = 5, accuracy = 0.445000
k = 5, accuracy = 0.453000
k = 5, accuracy = 0.453000
k = 5, accuracy = 0.488000
k = 8, accuracy = 0.375000
k = 8, accuracy = 0.407000
k = 8, accuracy = 0.397000
k = 8, accuracy = 0.402000
k = 8, accuracy = 0.404000
k = 10, accuracy = 0.366000
k = 10, accuracy = 0.396000
k = 10, accuracy = 0.369000
k = 10, accuracy = 0.382000
k = 10, accuracy = 0.385000
k = 12, accuracy = 0.356000
k = 12, accuracy = 0.387000
k = 12, accuracy = 0.360000
k = 12, accuracy = 0.365000
k = 12, accuracy = 0.371000
k = 15, accuracy = 0.333000
k = 15, accuracy = 0.359000
k = 15, accuracy = 0.355000
k = 15, accuracy = 0.349000
k = 15, accuracy = 0.361000
k = 20, accuracy = 0.331000
k = 20, accuracy = 0.343000
k = 20, accuracy = 0.333000
k = 20, accuracy = 0.328000
k = 20, accuracy = 0.341000
k = 50, accuracy = 0.291000
k = 50, accuracy = 0.300000
k = 50, accuracy = 0.304000
k = 50, accuracy = 0.297000
k = 50, accuracy = 0.293000
k = 100, accuracy = 0.270000
```

```
k = 100, accuracy = 0.283000
k = 100, accuracy = 0.274000
k = 100, accuracy = 0.281000
k = 100, accuracy = 0.284000
```

In [23]:
```
# 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()
```



In [24]:
```
# 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 = 1

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

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 c lassifier grows with the size of the training set. 5. None of the above.

Your Answer: 4

Your explanation: because the classification is computing distance between test point and each training point and the time for it is grows with the grow size of the training set.

svm_My

May 20, 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 [0]:
```
# '''
# Running or Importing .py Files with Google Colab
# Run these codes first in order to install the necessary libraries and perform author

# '''

# !apt-get install -y -qq software-properties-common python-software-properties module
# !add-apt-repository -y ppa:alessandro-strada/ppa 2>&1 > /dev/null
# !apt-get update -qq 2>&1 > /dev/null
# !apt-get -y install -qq google-drive-ocamlfuse fuse
# from google.colab import auth
# auth.authenticate_user()
# from oauth2client.client import GoogleCredentials
# creds = GoogleCredentials.get_application_default()
# import getpass
# !google-drive-ocamlfuse -headless -id={creds.client_id} -secret={creds.client_secret
# vcode = getpass.getpass()
# !echo {vcode} | google-drive-ocamlfuse -headless -id={creds.client_id} -secret={cred
```

In [0]:
```
# '''
# mount your Google Drive:
# '''

# !mkdir -p drive
# !google-drive-ocamlfuse drive
```

In [0]:
```
# import os
# os.chdir('drive/Colab Notebooks/cs231n/assignment1')
```

In [0]:
```
# Run some setup code for this notebook.

import random
import numpy as np
from cs231n.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
```

1.1 CIFAR-10 Data Loading and Preprocessing

In [0]:
```
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs231n/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 [0]:
```
# 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', 'truc
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 [0]:
```
# 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,)
```

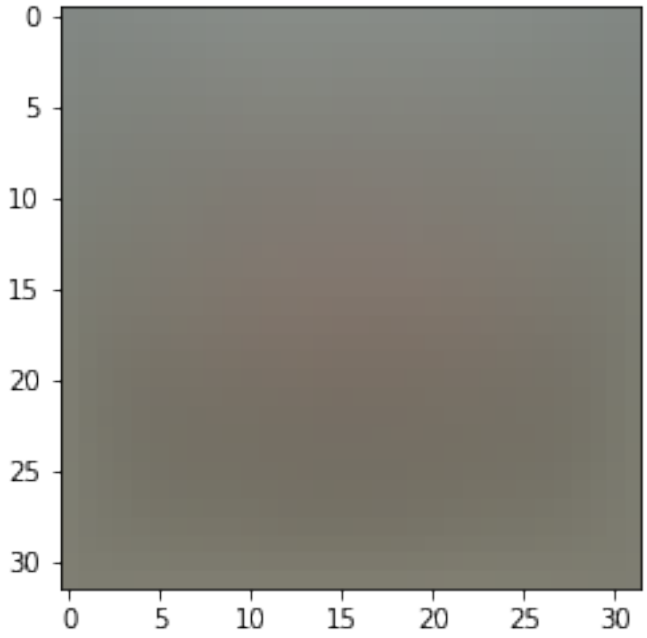
```
In [0]: # Preprocessing: reshape the image data into rows
        X_train = np.reshape(X_train, (X_train.shape[0], -1))
        X_val = np.reshape(X_val, (X_val.shape[0], -1))
        X_test = np.reshape(X_test, (X_test.shape[0], -1))
        X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))
```

```
        # 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 [0]:
```
        # 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 [0]:
```
        # 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 [0]: # 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 **cs231n/classifiers/linear_svm.py**.

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

```
In [0]: # Evaluate the naive implementation of the loss we provided for you:
        from cs231n.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: 8.654696
```

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 [0]: # 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 cs231n.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: -2.850507 analytic: -2.850507, relative error: 1.429385e-10
numerical: 30.089126 analytic: 30.089126, relative error: 3.097586e-12
numerical: -13.660684 analytic: -13.660684, relative error: 6.248406e-12
numerical: -6.615114 analytic: -6.615114, relative error: 3.943552e-11
numerical: 6.309124 analytic: 6.309124, relative error: 5.534922e-11
numerical: 68.875196 analytic: 68.875196, relative error: 2.135491e-13
numerical: 29.183127 analytic: 29.183127, relative error: 1.006961e-12
numerical: -19.837752 analytic: -19.837752, relative error: 2.175379e-11
numerical: 27.584583 analytic: 27.763042, relative error: 3.224333e-03
numerical: -26.786258 analytic: -26.786258, relative error: 2.795152e-12
numerical: -16.020844 analytic: -16.020844, relative error: 6.598785e-12
numerical: -4.693606 analytic: -4.582351, relative error: 1.199398e-02
numerical: -13.351179 analytic: -13.351179, relative error: 1.426016e-12
numerical: -1.327963 analytic: -1.327963, relative error: 1.776263e-10
numerical: 2.649903 analytic: 2.649903, relative error: 2.241271e-10
numerical: 18.285826 analytic: 18.285826, relative error: 1.581348e-11
numerical: -7.570040 analytic: -7.570040, relative error: 3.846919e-11
numerical: -18.052876 analytic: -18.101052, relative error: 1.332542e-03
numerical: -15.170802 analytic: -15.051329, relative error: 3.953162e-03
numerical: 6.233174 analytic: 6.138037, relative error: 7.690166e-03
```

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: *fill this in.*

The derivative $L(margin(delta))$ has a discontinuity of the first kind when $scores[j] - scores[y[i]] + 1 = 0$. The method can disperse near this point. It is necessary to replace the function by a similar differentiable.

```
In [0]: # 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 cs231n.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: 8.654696e+00 computed in 0.209710s
Vectorized loss: 8.654696e+00 computed in 0.000000s
difference: -0.000000
```

In [0]:
```
# Complete the implementation of svm_loss_vectorized, and compute the gradient
# of the loss function in a vectorized way.

# The naive implementation and the vectorized implementation should match, but
# the vectorized version should still be much faster.
tic = time.time()
_, grad_naive = svm_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Naive loss and gradient: computed in %fs' % (toc - tic))

tic = time.time()
_, grad_vectorized = svm_loss_vectorized(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Vectorized loss and gradient: computed in %fs' % (toc - tic))

# The loss is a single number, so it is easy to compare the values computed
# by the two implementations. The gradient on the other hand is a matrix, so
# we use the Frobenius norm to compare them.
difference = np.linalg.norm(grad_naive - grad_vectorized, ord='fro')
print('difference: %f' % difference)
```

```
Naive loss and gradient: computed in 0.187483s
Vectorized loss and gradient: computed in 0.000000s
difference: 0.000000
```

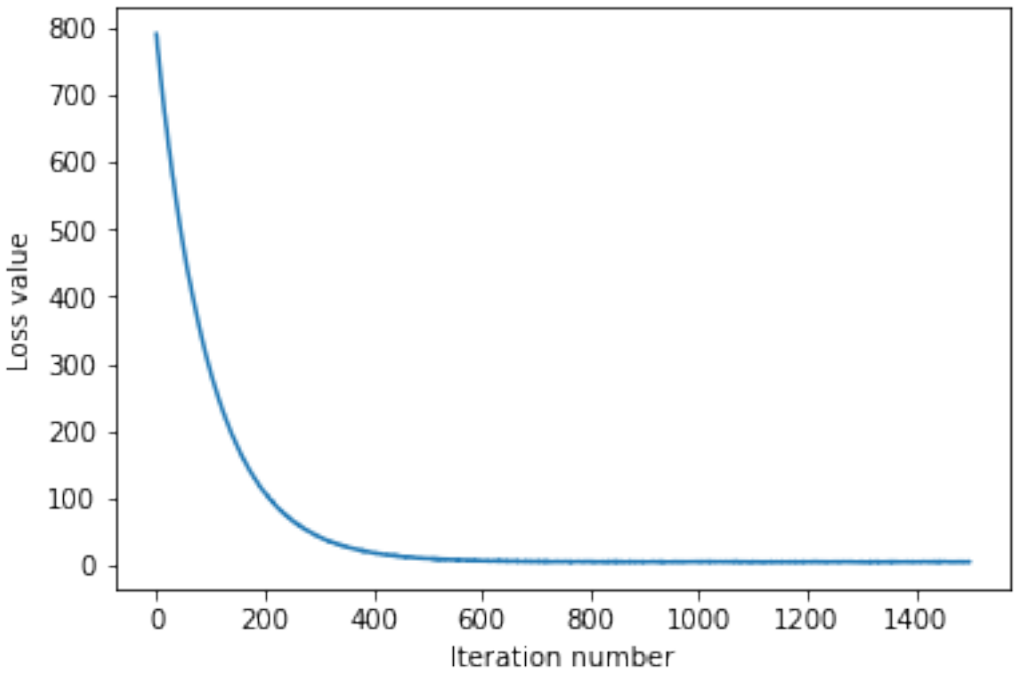
1.2.2 Stochastic Gradient Descent

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 [0]:
```
# In the file linear_classifier.py, implement SGD in the function
# LinearClassifier.train() and then run it with the code below.
from cs231n.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 790.202494
iteration 100 / 1500: loss 288.843432
iteration 200 / 1500: loss 108.079931
iteration 300 / 1500: loss 42.953444
iteration 400 / 1500: loss 19.107166
iteration 500 / 1500: loss 9.951262
iteration 600 / 1500: loss 7.005451
iteration 700 / 1500: loss 6.022529
iteration 800 / 1500: loss 5.460717
iteration 900 / 1500: loss 5.030977
iteration 1000 / 1500: loss 5.347833
iteration 1100 / 1500: loss 5.100142
iteration 1200 / 1500: loss 5.391687
iteration 1300 / 1500: loss 5.143432
iteration 1400 / 1500: loss 5.346606
That took 15.207583s
```

In [0]:
```
# 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 [0]: # 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: 0.373102
validation accuracy: 0.384000
```

```
In [0]: learning_rates = [1e-7, 5e-5]
        regularization_strengths = [2.5e4, 5e4]
        from itertools import product
        for param in product(learning_rates, regularization_strengths):
            print(param)
```

```
(1e-07, 25000.0)
(1e-07, 50000.0)
(5e-05, 25000.0)
(5e-05, 50000.0)
```

```
In [0]: %%time
        # 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-5]
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.
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.

################################################################################
# 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.                                      #
################################################################################
for lr in learning_rates:
    for reg in regularization_strengths:
        svm = LinearSVM()
        _ = svm.train(X_train, y_train, learning_rate=lr, reg=reg,
                      num_iters=1500)
        y_train_pred = svm.predict(X_train)
        training_accuracy = np.mean(y_train == y_train_pred)
        y_val_pred = svm.predict(X_val)
        validation_accuracy = np.mean(y_val == y_val_pred)
        results[((lr, reg))] = (training_accuracy, validation_accuracy)
        if validation_accuracy > best_val:
            best_val = validation_accuracy
            best_svm = svm
################################################################################
#                              END OF YOUR CODE                                #
################################################################################

# Print out results.
for lr, reg in sorted(results):
    train_accuracy, val_accuracy = results[(lr, reg)]
```

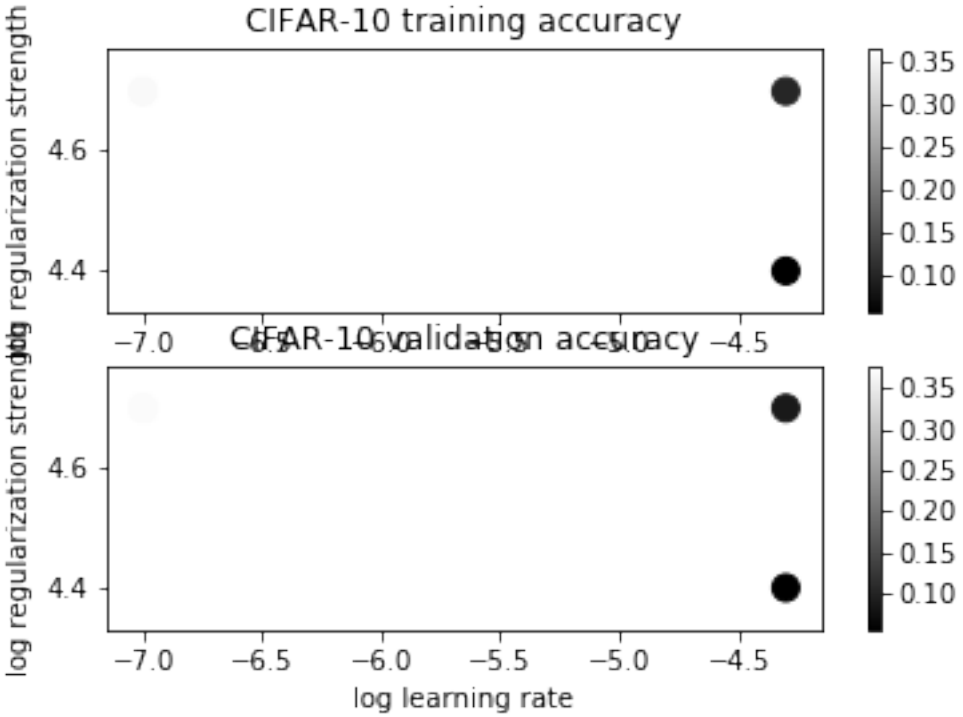
```
            print('lr %e reg %e train accuracy: %f val accuracy: %f' % (
                        lr, reg, train_accuracy, val_accuracy))

        print('best validation accuracy achieved during cross-validation: %f' % best_val)
```

```
C:\Users\dsher\Google \Colab Notebooks\cs231n\assignment1\cs231n\classifiers\linear_svm.py:94:
  loss += reg * np.sum(W * W)
C:\Users\dsher\Anaconda3\envs\cs231n\lib\site-packages\numpy\core\_methods.py:32: RuntimeWarni
  return umr_sum(a, axis, dtype, out, keepdims)
C:\Users\dsher\Google \Colab Notebooks\cs231n\assignment1\cs231n\classifiers\linear_svm.py:94:
  loss += reg * np.sum(W * W)
C:\Users\dsher\Google \Colab Notebooks\cs231n\assignment1\cs231n\classifiers\linear_svm.py:95:
  dW += 2 * reg * W
C:\Users\dsher\Google \Colab Notebooks\cs231n\assignment1\cs231n\classifiers\linear_svm.py:81:
  margins = np.maximum(0, scores - scores[np.arange(num_train), y].reshape(-1, 1) + delta)
C:\Users\dsher\Google \Colab Notebooks\cs231n\assignment1\cs231n\classifiers\linear_svm.py:84:
  mask[np.arange(num_train), y] = - ((margins > 0) * mask).sum(axis=1)
C:\Users\dsher\Google \Colab Notebooks\cs231n\assignment1\cs231n\classifiers\linear_svm.py:85:
  mask *= (margins > 0)
C:\Users\dsher\Google \Colab Notebooks\cs231n\assignment1\cs231n\classifiers\linear_classifier
  self.W -= learning_rate * grad
```

```
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.364592 val accuracy: 0.377000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.356122 val accuracy: 0.371000
lr 5.000000e-05 reg 2.500000e+04 train accuracy: 0.054143 val accuracy: 0.055000
lr 5.000000e-05 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
best validation accuracy achieved during cross-validation: 0.377000
Wall time: 43 s
```

```
In [0]: # 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()
```

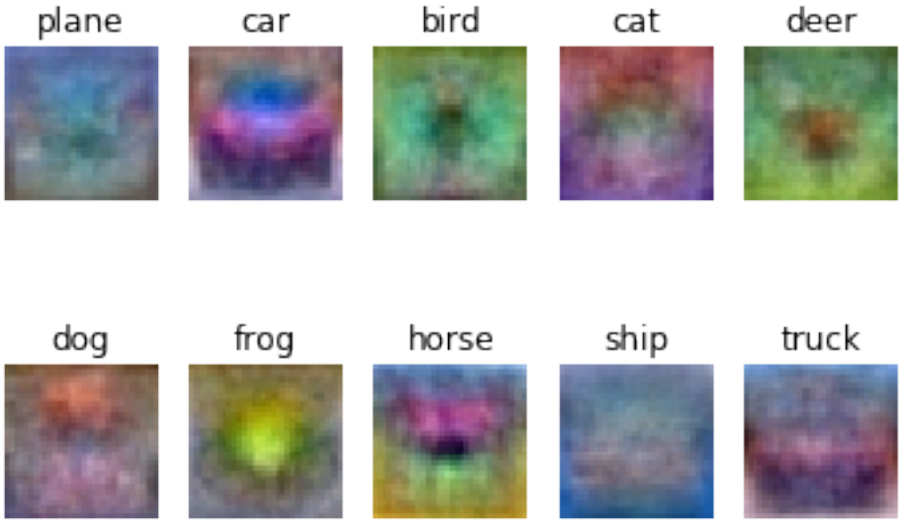


In [0]:
```
# Evaluate the best svm on test set
y_test_pred = best_svm.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
```

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

In [0]:
```
# 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:

The visualized SVM weights look like as averaged images corresponding to the names of the labels. This happens when the corresponding scores are maximized in the corresponding expressions and all others are minimized. And such a maximization is obtained with the maximum coincidence of weights and those values to which they are multiplied.

softmax_My

May 20, 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 cs231n.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 = 'cs231n/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 **cs231n/classifiers/softmax.py**.

```
In [53]: # First implement the naive softmax loss function with nested loops.
         # Open the file cs231n/classifiers/softmax.py and implement the
         # softmax_loss_naive function.

         from cs231n.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: 2.386702
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: *Because we have ten classes and the mean loss is neary log zero point one*

In [54]:
```
# 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 cs231n.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: -2.734919 analytic: -2.734919, relative error: 5.350510e-09
numerical: -5.990124 analytic: -5.990123, relative error: 1.198755e-08
numerical: 1.777149 analytic: 1.777149, relative error: 3.579210e-08
numerical: 0.664249 analytic: 0.664249, relative error: 1.582974e-07
numerical: 0.799516 analytic: 0.799516, relative error: 2.049939e-08
numerical: -1.759006 analytic: -1.759007, relative error: 3.245047e-08
numerical: -0.759323 analytic: -0.759323, relative error: 3.629384e-08
numerical: -0.540509 analytic: -0.540510, relative error: 1.156137e-07
numerical: 3.532213 analytic: 3.532213, relative error: 2.589805e-08
numerical: -0.488441 analytic: -0.488441, relative error: 1.130461e-08
numerical: 3.756660 analytic: 3.756659, relative error: 1.600612e-08
numerical: 0.517707 analytic: 0.517707, relative error: 3.524529e-08
numerical: -1.701453 analytic: -1.701453, relative error: 4.004619e-08
numerical: 0.354527 analytic: 0.354527, relative error: 4.805298e-08
numerical: 7.358082 analytic: 7.358082, relative error: 1.659233e-08
numerical: -2.551213 analytic: -2.551213, relative error: 1.033921e-08
numerical: -6.029541 analytic: -6.029541, relative error: 8.506012e-09
numerical: 1.945461 analytic: 1.945461, relative error: 5.640307e-09
numerical: -0.043188 analytic: -0.043188, relative error: 8.622491e-07
numerical: 2.218568 analytic: 2.218568, relative error: 5.251569e-09
```

In [62]:
```
# Now that we have a naive implementation of the softmax loss function and its gradie
# 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 cs231n.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: 2.386702e+00 computed in 0.320817s
vectorized loss: 2.386702e+00 computed in 0.007019s
Loss difference: 0.000000
Gradient difference: 0.000000
```

In [48]:
```
        # 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 cs231n.classifiers import Softmax
        results = {}
        best_val = -1
        best_softmax = None
        learning_rates = [1e-7, 5e-7]
        regularization_strengths = [2.5e4, 5e4]

        ################################################################################
        # 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.                          #
        ################################################################################
        for lr in learning_rates:
            for reg in regularization_strengths:
                softmax = Softmax()
                _ = softmax.train(X_train, y_train, learning_rate=lr, reg=reg,
                            num_iters=1500)
                y_train_pred = softmax.predict(X_train)
                training_accuracy = np.mean(y_train == y_train_pred)
                y_val_pred = softmax.predict(X_val)
                validation_accuracy = np.mean(y_val == y_val_pred)
                results[((lr, reg))] = (training_accuracy, validation_accuracy)
                if validation_accuracy > best_val:
                    best_val = validation_accuracy
                    best_softmax = softmax
        ################################################################################
        #                              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-07 reg 2.500000e+04 train accuracy: 0.325102 val accuracy: 0.343000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.315531 val accuracy: 0.326000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.317653 val accuracy: 0.334000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.307633 val accuracy: 0.317000
best validation accuracy achieved during cross-validation: 0.343000
```

```
In [49]: # 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: 0.338000
```

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: It's True if the datapoint margins is negativ for any uncorrect class score (*or:* if the correct class score is greater that any uncorrect class score plus delta).

```
In [50]: # 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])
```


plane
car
bird
cat
deer
dog
frog
horse
ship
truck

two_layer_net_My

May 20, 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 [2]:
```
# A bit of setup

import numpy as np
import matplotlib.pyplot as plt

from cs231n.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 `cs231n/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 [93]:
```
# 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 `cs231n/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 [94]: 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 [95]: 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:
1.7985612998927536e-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 [96]: from cs231n.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[p
```

```
W2 max relative error: 3.440708e-09
b2 max relative error: 3.865091e-11
W1 max relative error: 3.561318e-09
b1 max relative error: 1.555471e-09
```

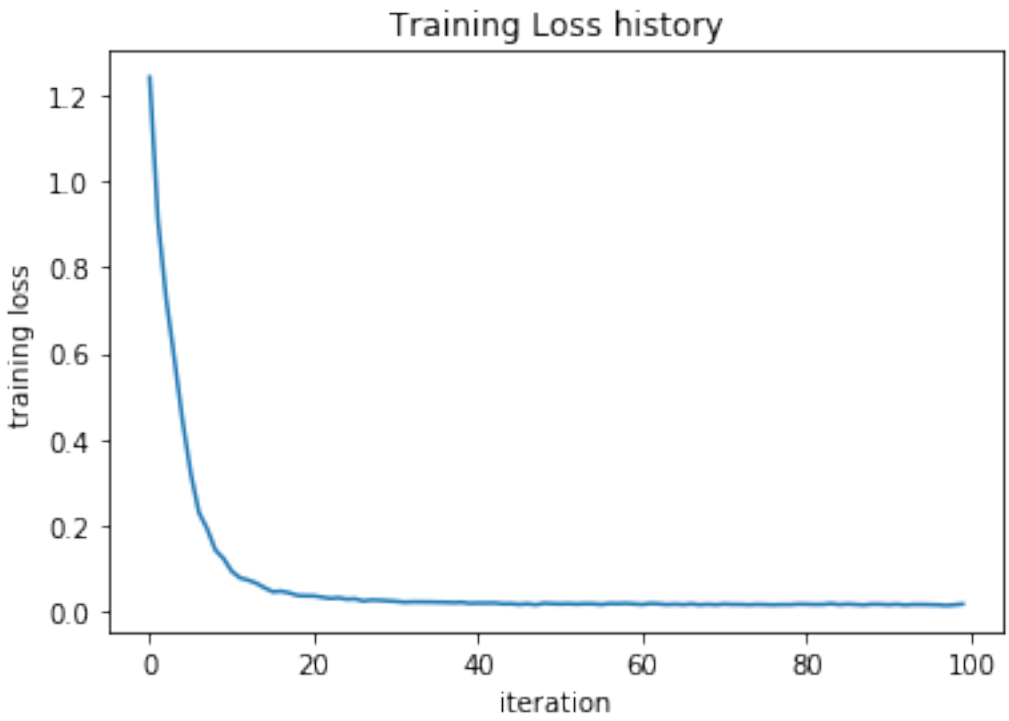
5 Train the network

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 [97]: 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:  0.017149612521020395
```



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 [98]:
```
from cs231n.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 = 'cs231n/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 memor
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)
```

```
Clear previously loaded data.
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 [99]: 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

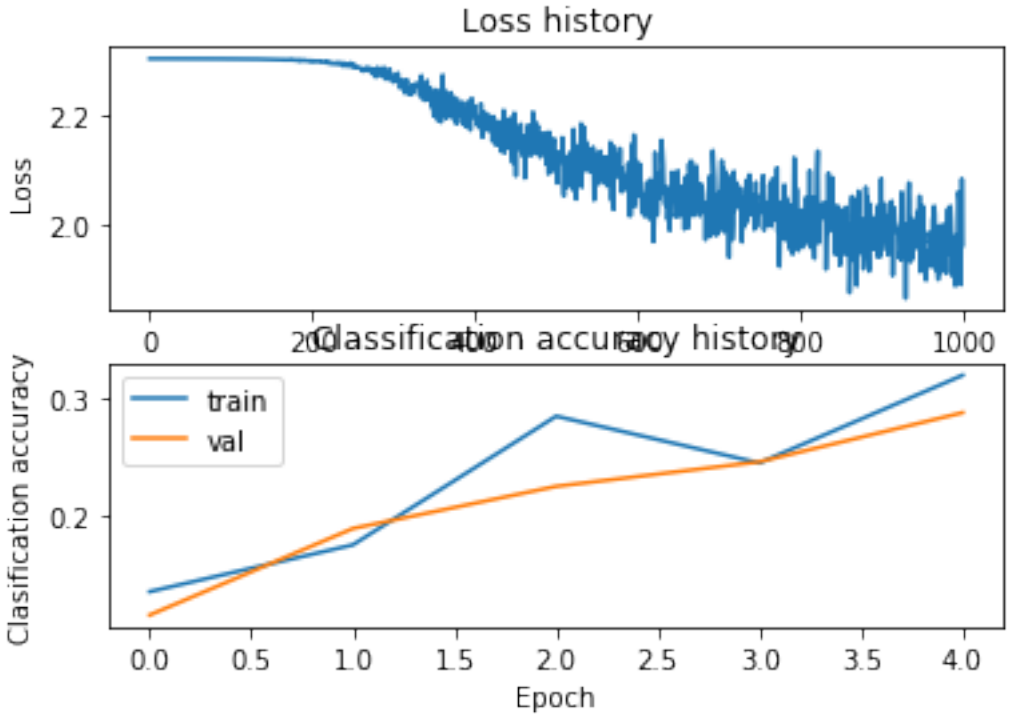
With the default parameters we provided above, you should get a validation accuracy of about 0.29 on the validation set. This isn't very good.

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.

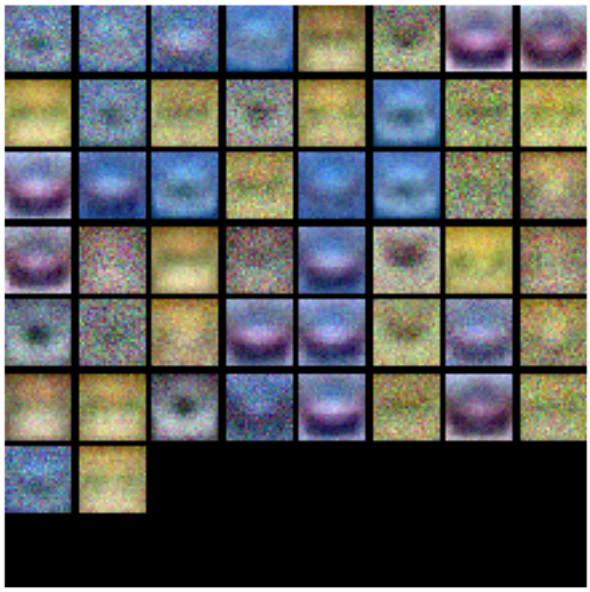
Another strategy is to visualize the weights that were learned in the first layer of the network. In most neural networks trained on visual data, the first layer weights typically show some visible structure when visualized.

```
In [100]: # 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 [101]:
```
from cs231n.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 [142]: %%time
          best_net = None # store the best model into this
          ################################################################################
          # 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.                          #
#################################################################################
results = {}
best_val = -1

# the constant parameters
input_size = 32 * 32 * 3
num_classes = 10
batch_size = 200

# the tuning parameters
hidden_size = [40, 50, 60]
num_iters = [1000]
learning_rate = [1e-3, 1e-4, 1e-5]
learning_rate_decay = [0.9, 0.95, 0.97]
reg = [0.2, 0.25, 0.3]


params = product(hidden_size, num_iters, learning_rate, learning_rate_decay, reg)
# total = 1
# for paramlist in [hidden_size, num_iters, learning_rate, learning_rate_decay, reg]
#     total *=len(paramlist)

for param in params:
    net = TwoLayerNet(input_size, param[0], num_classes)

    # Train the network
    stats = net.train(X_train, y_train, X_val, y_val,
                num_iters=param[1], batch_size=batch_size,
                learning_rate=param[2], learning_rate_decay=param[3],
                reg=param[4], verbose=False)

    # Predict on the train set
    y_train_pred = net.predict(X_train)
    training_accuracy = np.mean(y_train == y_train_pred)
    # Predict on the validation set
    y_val_pred = net.predict(X_val)
    validation_accuracy = np.mean(y_val == y_val_pred)
    results[param] = (training_accuracy, validation_accuracy)
    if validation_accuracy > best_val:
        best_val = validation_accuracy
```

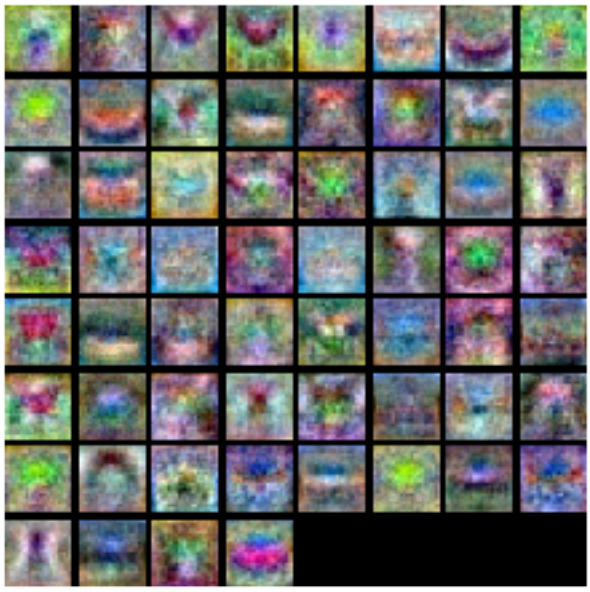
```
                best_net = net
                best_param = param
        ################################################################################
        #                                END OF YOUR CODE                              #
        ################################################################################
```

```
Wall time: 23min 13s
```

```
In [143]: print('best validation accuracy achieved during cross-validation: %f' % best_val)
          print('best hyperparameters')
          for i, prm in enumerate(['hidden_size', 'num_iters', 'learning_rate', 'learning_rate_
              print(prm + ': %f' % best_param[i])
```

```
best validation accuracy achieved during cross-validation: 0.490000
best hyperparameters
hidden_size: 60.000000
num_iters: 1000.000000
learning_rate: 0.001000
learning_rate_decay: 0.950000
reg: 0.300000
```

```
In [144]: # 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 [145]: test_acc = (best_net.predict(X_test) == y_test).mean()
          print('Test accuracy: ', test_acc)
```

```
Test accuracy:  0.474
```

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: 1, 2, 3

Your explanation:

1. A larger dataset is allways better 2. More complex network allows for more details 3. Increase the regularization strength is remedy against overfitting

features_My_colab

May 20, 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 [2]: !apt-get install -y -qq software-properties-common python-software-properties module-in
        !add-apt-repository -y ppa:alessandro-strada/ppa 2>&1 > /dev/null
        !apt-get update -qq 2>&1 > /dev/null
        !apt-get -y install -qq google-drive-ocamlfuse fuse
        from google.colab import auth
        auth.authenticate_user()
        from oauth2client.client import GoogleCredentials
        creds = GoogleCredentials.get_application_default()
        import getpass
        !google-drive-ocamlfuse -headless -id={creds.client_id} -secret={creds.client_secret} <
        vcode = getpass.getpass()
        !echo {vcode} | google-drive-ocamlfuse -headless -id={creds.client_id} -secret={creds.c

Please, open the following URL in a web browser: https://accounts.google.com/o/oauth2/auth?clie

ůůůůůůůůůů
Please, open the following URL in a web browser: https://accounts.google.com/o/oauth2/auth?clie
Please enter the verification code: Access token retrieved correctly.


In [0]: !mkdir -p drive
        !google-drive-ocamlfuse drive

In [0]: import os
        os.chdir('drive/Colab Notebooks/cs231n/assignment1')
```

```
In [0]: import random
        import numpy as np
        from cs231n.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
```

1.1 Load data

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

```
In [0]: from cs231n.features import color_histogram_hsv, hog_feature

        def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000):
            # Load the raw CIFAR-10 data
            cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

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

            # Subsample the data
            mask = 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()
```

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 [7]:
```
from cs231n.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.

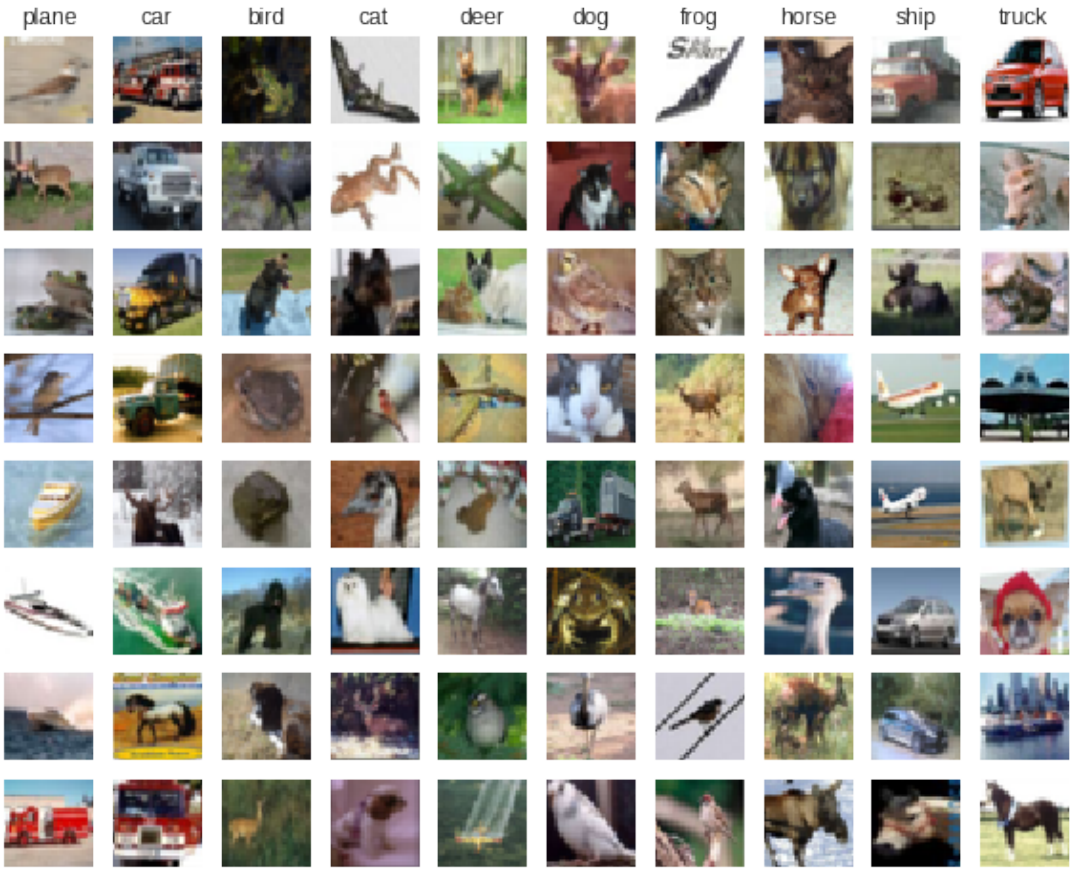
In [8]:
```
# Use the validation set to tune the learning rate and regularization strength

from cs231n.classifiers.linear_classifier import LinearSVM

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

results = {}
best_val = -1
best_svm = None

################################################################################
# TODO:                                                                        #
# Use the validation set to set the learning rate and regularization strength. #
# This should be identical to the validation that you did for the SVM; save    #
# the best trained classifer in best_svm. You might also want to play          #
# with different numbers of bins in the color histogram. If you are careful    #
# you should be able to get accuracy of near 0.44 on the validation set.       #
################################################################################
for lr in learning_rates:
    for reg in regularization_strengths:
        svm = LinearSVM()
        _ = svm.train(X_train_feats, y_train, learning_rate=lr, reg=reg,
                      num_iters=1500)
        y_train_pred = svm.predict(X_train_feats)
        training_accuracy = np.mean(y_train == y_train_pred)
        y_val_pred = svm.predict(X_val_feats)
        validation_accuracy = np.mean(y_val == y_val_pred)
        results[((lr, reg))] = (training_accuracy, validation_accuracy)
        if validation_accuracy > best_val:
            best_val = validation_accuracy
            best_svm = svm
################################################################################
#                              END OF YOUR CODE                                #
################################################################################

# Print out results.
for lr, reg in sorted(results):
    train_accuracy, val_accuracy = results[(lr, reg)]
    print('lr %e reg %e train accuracy: %f val accuracy: %f' % (
                lr, reg, train_accuracy, val_accuracy))

print('best validation accuracy achieved during cross-validation: %f' % best_val)
```

```
lr 1.000000e-09 reg 5.000000e+04 train accuracy: 0.082816 val accuracy: 0.082000
lr 1.000000e-09 reg 5.000000e+05 train accuracy: 0.091551 val accuracy: 0.091000
lr 1.000000e-09 reg 5.000000e+06 train accuracy: 0.413490 val accuracy: 0.416000
lr 1.000000e-08 reg 5.000000e+04 train accuracy: 0.102388 val accuracy: 0.099000
lr 1.000000e-08 reg 5.000000e+05 train accuracy: 0.412531 val accuracy: 0.415000
lr 1.000000e-08 reg 5.000000e+06 train accuracy: 0.397000 val accuracy: 0.410000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.416204 val accuracy: 0.423000
lr 1.000000e-07 reg 5.000000e+05 train accuracy: 0.409612 val accuracy: 0.399000
lr 1.000000e-07 reg 5.000000e+06 train accuracy: 0.323204 val accuracy: 0.309000
best validation accuracy achieved during cross-validation: 0.423000
```

In [9]:
```
# 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)
```

```
0.423
```

In [10]:
```
# 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?

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 [11]:
```
# 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 [0]:
```
from itertools import product
```

In [18]:
```
%%time
from cs231n.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

################################################################################
# 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.                                              #
################################################################################
results = {}
best_val = -1

# the constant parameters
batch_size = 200
hidden_size = [hidden_dim]
num_iters = [1000]

# the tuning parameters
learning_rate = [1, 0.7, 0.5, 0.2, 0.1, 0.01]
learning_rate_decay = [0.9, 0.95, 0.97, 0.99, 1, 1.1]
reg = [1e-1, 1e-2, 1e-3, 1e-4, 1e-5, 1e-6]


params = product(hidden_size, num_iters, learning_rate, learning_rate_decay, reg)

for param in params:
    net = TwoLayerNet(input_dim, hidden_dim, num_classes)
    # Train the network
    stats = net.train(X_train_feats, y_train, X_val_feats, y_val,
                num_iters=param[1], batch_size=batch_size,
                learning_rate=param[2], learning_rate_decay=param[3],
                reg=param[4], verbose=False)
```

```
            # Predict on the train set
            y_train_pred = net.predict(X_train_feats)
            training_accuracy = np.mean(y_train == y_train_pred)
            # Predict on the validation set
            y_val_pred = net.predict(X_val_feats)
            validation_accuracy = np.mean(y_val == y_val_pred)
            results[param] = (training_accuracy, validation_accuracy)
            if validation_accuracy > best_val:
                best_val = validation_accuracy
                best_net = net
                best_param = param
        ################################################################################
        #                              END OF YOUR CODE                                #
        ################################################################################
```

```
CPU times: user 45min 39s, sys: 12min 15s, total: 57min 55s
Wall time: 29min 13s
```

```
In [19]: print('best validation accuracy achieved during cross-validation: %f' % best_val)
         print('best hyperparameters')
         for i, prm in enumerate(['hidden_size', 'num_iters', 'learning_rate', 'learning_rate_
             print(prm + ': %f' % best_param[i])
```

```
best validation accuracy achieved during cross-validation: 0.591000
best hyperparameters
hidden_size: 500.000000
num_iters: 1000.000000
learning_rate: 0.700000
learning_rate_decay: 0.900000
reg: 0.000001
```

```
In [20]: # 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)
```

```
0.568
```