knn

August 30, 2021

[1]:
```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

# Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment1/'
FOLDERNAME ='cs231n/assignments/assignment1/'
assert FOLDERNAME is not None, "[!] Enter the foldername."

# Now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# This downloads the CIFAR-10 dataset to your Drive
# if it doesn't already exist.
%cd drive/My\ Drive/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/cs231n/assignments/assignment1/cs231n/datasets
/content/drive/My Drive/cs231n/assignments/assignment1
```

[2]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
```

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.

[9]:
```
# 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

# 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
```

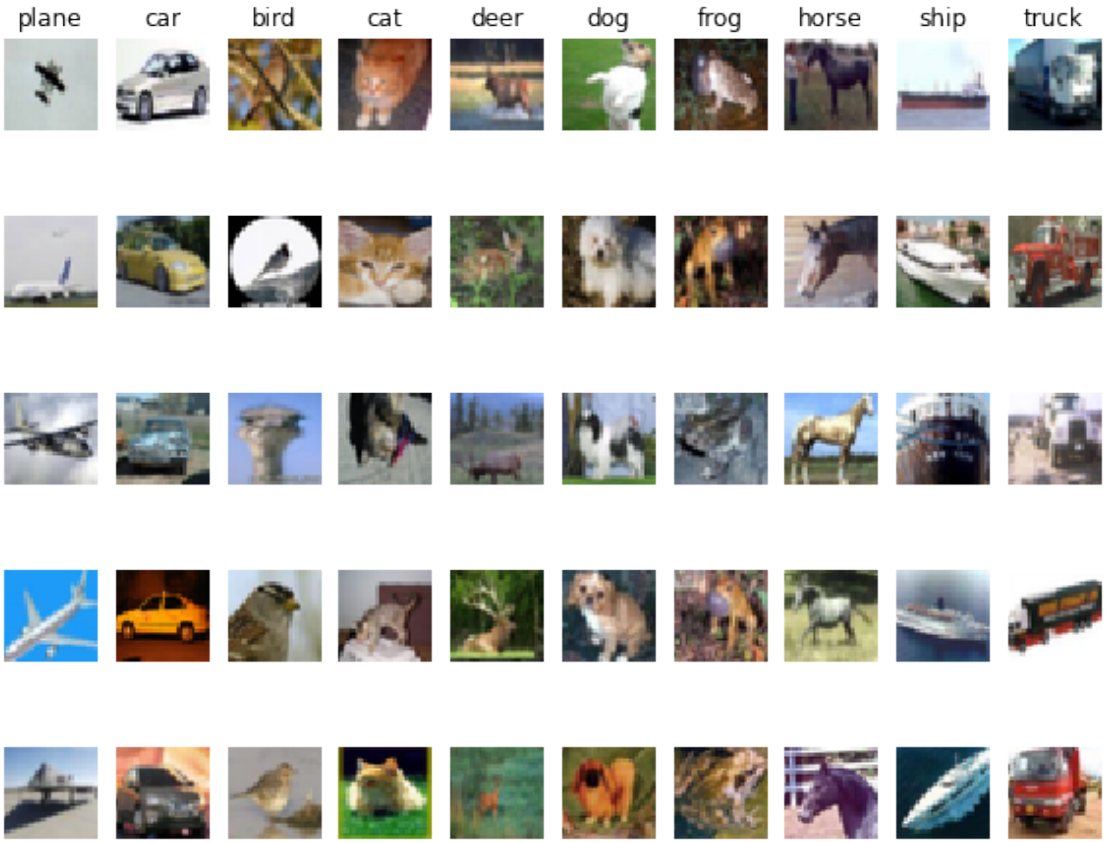
```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

[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␣
 ↪memory issue)
try:
   del X_train, y_train
   del X_test, y_test
   print('Clear previously loaded data.')
except:
   pass

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

```
# As a sanity check, we print out the size of the training and test data.
print(type(X_train),type(y_train))
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.
<class 'numpy.ndarray'> <class 'numpy.ndarray'>
Training data shape:  (50000, 32, 32, 3)
Training labels shape:  (50000,)
Test data shape:  (10000, 32, 32, 3)
Test labels shape:  (10000,)
```

[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', 'truck']
num_classes = len(classes)
samples_per_class = 5
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()
```



[12]:
```
# 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]

# 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)
```

[13]:
```
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.

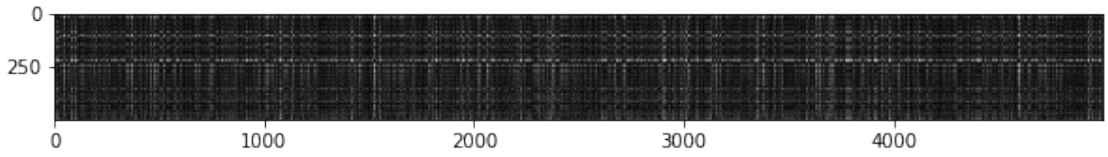
Note: For the three distance computations that we require you to implement in this notebook, you may not use the np.linalg.norm() function that numpy provides.

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.

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

[15]:
```
# 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?

YourAnswer : 1. either the test data is not included in any classes or the test data isn't similar with trained data probably because of its unique background color. 2. it means there is no test dats similar with traind data.

[16]:
```
# 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`:

[17]:
```
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 145 / 500 correct => accuracy: 0.290000
```

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

Inline Question 2

We can also use other distance metrics such as L1 distance. For pixel values $p_{ij}^{(k)}$ at location (i, j) of some image I_k,

the mean μ across all pixels over all images is

$$\mu = \frac{1}{nhw} \sum_{k=1}^{n} \sum_{i=1}^{h} \sum_{j=1}^{w} p_{ij}^{(k)}$$

And the pixel-wise mean μ_{ij} across all images is

$$\mu_{ij} = \frac{1}{n} \sum_{k=1}^{n} p_{ij}^{(k)}.$$

The general standard deviation σ and pixel-wise standard deviation σ_{ij} is defined similarly.

Which of the following preprocessing steps will not change the performance of a Nearest Neighbor classifier that uses L1 distance? Select all that apply. 1. Subtracting the mean μ ($\tilde{p}_{ij}^{(k)} = p_{ij}^{(k)} - \mu$.) 2. Subtracting the per pixel mean μ_{ij} ($\tilde{p}_{ij}^{(k)} = p_{ij}^{(k)} - \mu_{ij}$.) 3. Subtracting the mean μ and dividing by the standard deviation σ. 4. Subtracting the pixel-wise mean μ_{ij} and dividing by the pixel-wise standard deviation σ_{ij}. 5. Rotating the coordinate axes of the data.

YourAnswer : 1,3

YourExplanation :

For steps 2,4, it will change relative amount of distance because per pixel mean will be diffenet from all images. So, it will affect to L1 distance.

For step 5, performance will be changed because L1 distance is axes-dependence distance.

[18]:
```
# 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('One loop 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')
```

```
One loop difference was: 0.000000
Good! The distance matrices are the same
```

[19]:
```
# 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('No loop 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')
```

```
No loop difference was: 0.000000
Good! The distance matrices are the same
```

[20]:
```
# 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 execute.
    """
    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␣
 ↪implementation!

# NOTE: depending on what machine you're using,
# you might not see a speedup when you go from two loops to one loop,
# and might even see a slow-down.
```

```
Two loop version took 31.018522 seconds
One loop version took 40.713554 seconds
No loop version took 0.523067 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.

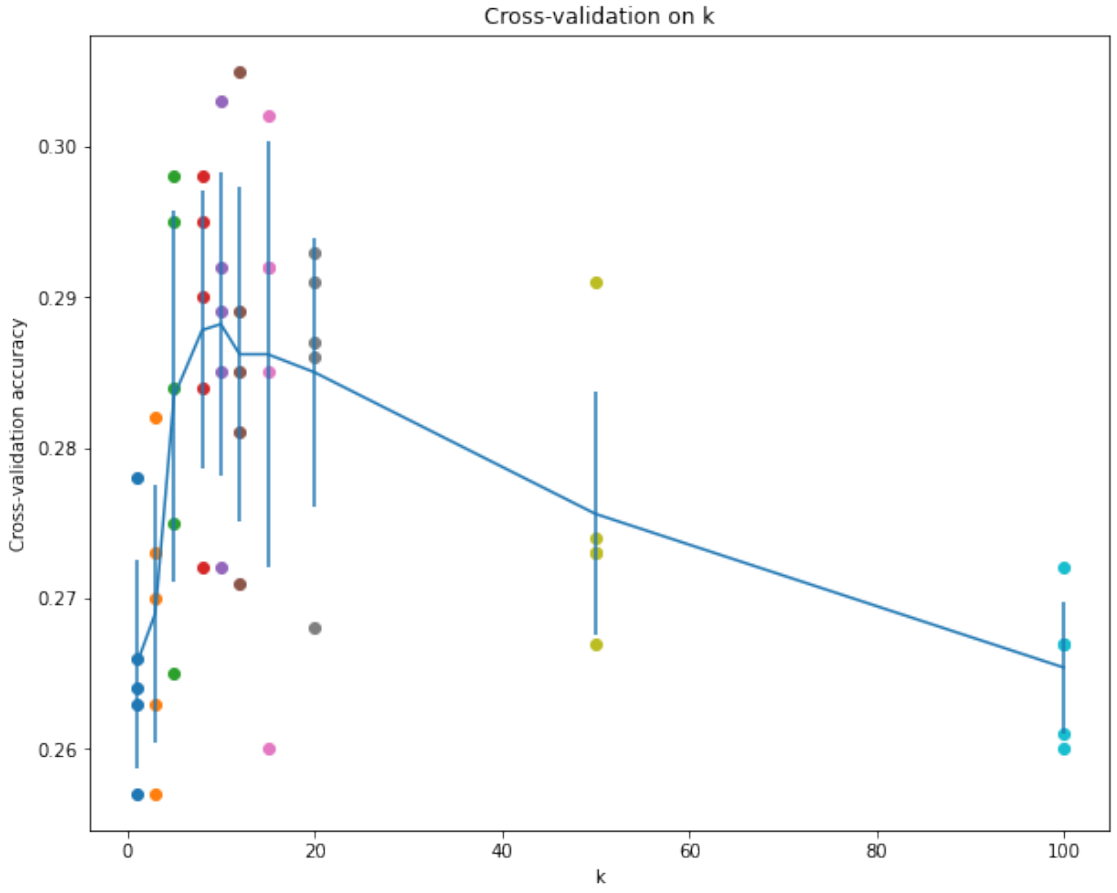
[68]:
```
num_folds = 5
k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50, 100]

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.                               ␣
 ↪#
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
X_train_folds = np.array_split(X_train,num_folds)
y_train_folds = np.array_split(y_train,num_folds)
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

```
# 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.                               #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
for k in k_choices:
  res = [0]*num_folds
  for i in range(num_folds):
    X_training=np.concatenate([x for num,x in enumerate(X_train_folds) if num!=i])
    y_training=np.concatenate([y for num,y in enumerate(y_train_folds) if num!=i])
    classifirer = KNearestNeighbor()
    classifier.train(X_training, y_training)
    y_test_pred = classifier.predict(X_train_folds[i],k)
    num_correct = np.sum(y_test_pred == y_train_folds[i])
    res[i] = float(num_correct) / y_train_folds[i].shape[0]
  k_to_accuracies[k]=res
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

# 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 = 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 = 0.257000
k = 3, accuracy = 0.263000
k = 3, accuracy = 0.273000
k = 3, accuracy = 0.282000
k = 3, accuracy = 0.270000
k = 5, accuracy = 0.265000
k = 5, accuracy = 0.275000
k = 5, accuracy = 0.295000
k = 5, accuracy = 0.298000
k = 5, accuracy = 0.284000
k = 8, accuracy = 0.272000
k = 8, accuracy = 0.295000
k = 8, accuracy = 0.284000
k = 8, accuracy = 0.298000
k = 8, accuracy = 0.290000
k = 10, accuracy = 0.272000
k = 10, accuracy = 0.303000
k = 10, accuracy = 0.289000
k = 10, accuracy = 0.292000
k = 10, accuracy = 0.285000
k = 12, accuracy = 0.271000
k = 12, accuracy = 0.305000
k = 12, accuracy = 0.285000
k = 12, accuracy = 0.289000
k = 12, accuracy = 0.281000
k = 15, accuracy = 0.260000
k = 15, accuracy = 0.302000
k = 15, accuracy = 0.292000
k = 15, accuracy = 0.292000
k = 15, accuracy = 0.285000
k = 20, accuracy = 0.268000
k = 20, accuracy = 0.293000
k = 20, accuracy = 0.291000
k = 20, accuracy = 0.287000
k = 20, accuracy = 0.286000
k = 50, accuracy = 0.273000
k = 50, accuracy = 0.291000
k = 50, accuracy = 0.274000
k = 50, accuracy = 0.267000
k = 50, accuracy = 0.273000
k = 100, accuracy = 0.261000
k = 100, accuracy = 0.272000
k = 100, accuracy = 0.267000
k = 100, accuracy = 0.260000
k = 100, accuracy = 0.267000
```

[69]:
```
# 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()
```



[70]:
```
# 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 = 8

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 144 / 500 correct => accuracy: 0.288000
```

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 decision boundary of the k-NN classifier is linear. 2. The training error of a 1-NN will always be lower than or equal to that of 5-NN. 3. The test error of a 1-NN will always be lower than that of a 5-NN. 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.

YourAnswer : 2, 4

YourExplanation :

1 - X

Becuase we can't assert that arbitary images have same proportion of distances from the classes.

2 - O

For 1-NN, distance will be 0 because it will always find itself as nearest images. But 5-NN could find different images, so we can't assure that distance will be 0. So, training error of 1-NN is always be lower that that of 5-NN.

3 - X

k is hyper-parameter, so it depends on our datas.

4 - O

It is true because we have to compute more distances for bigger training set.

svm

August 30, 2021

[1]:
```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

# Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment1/'
FOLDERNAME = 'cs231n/assignments/assignment1/'
assert FOLDERNAME is not None, "[!] Enter the foldername."

# Now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# This downloads the CIFAR-10 dataset to your Drive
# if it doesn't already exist.
%cd drive/My\ Drive/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/cs231n/assignments/assignment1/cs231n/datasets
/content/drive/My Drive/cs231n/assignments/assignment1
```

[2]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
```

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

[3]:
```
# 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

# 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
```

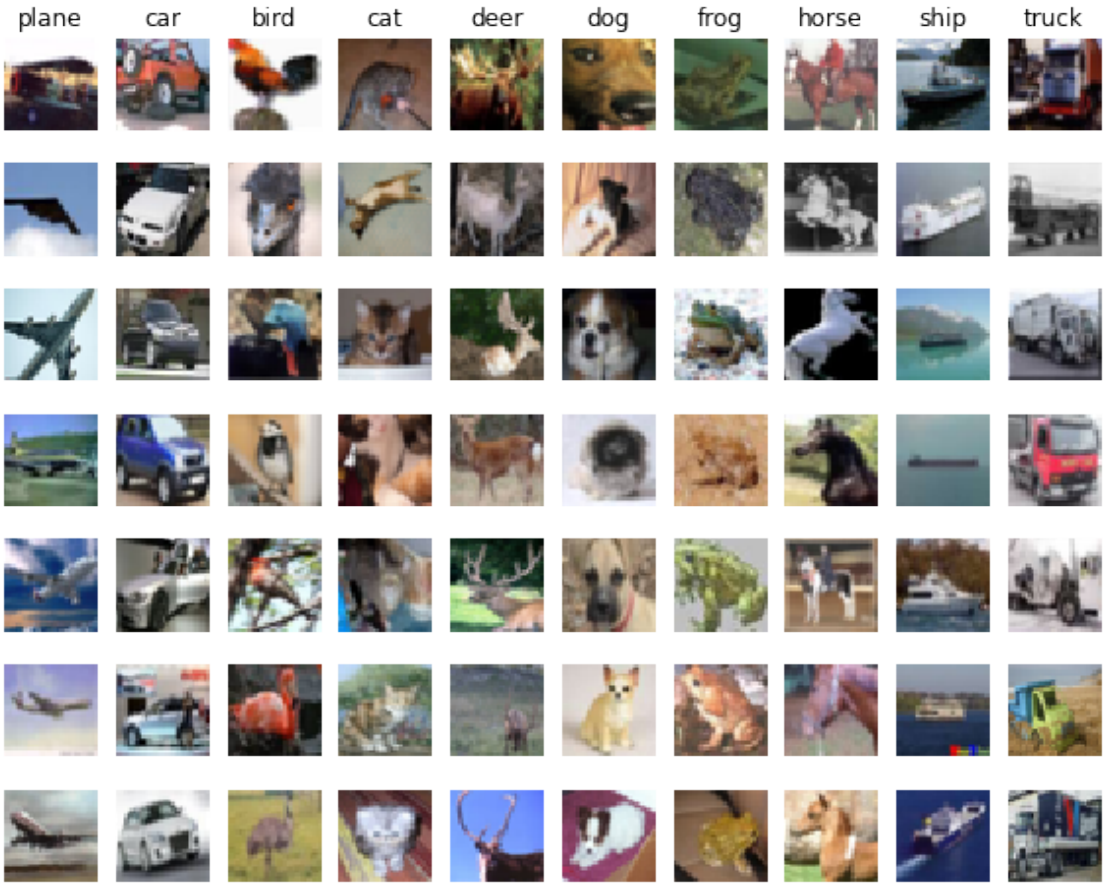
1.1 CIFAR-10 Data Loading and Preprocessing

[5]:
```
# 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 issue)
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,)
```

[6]:
```
# 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', 'truck']
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()
```



[7]:
```
# 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,)
```

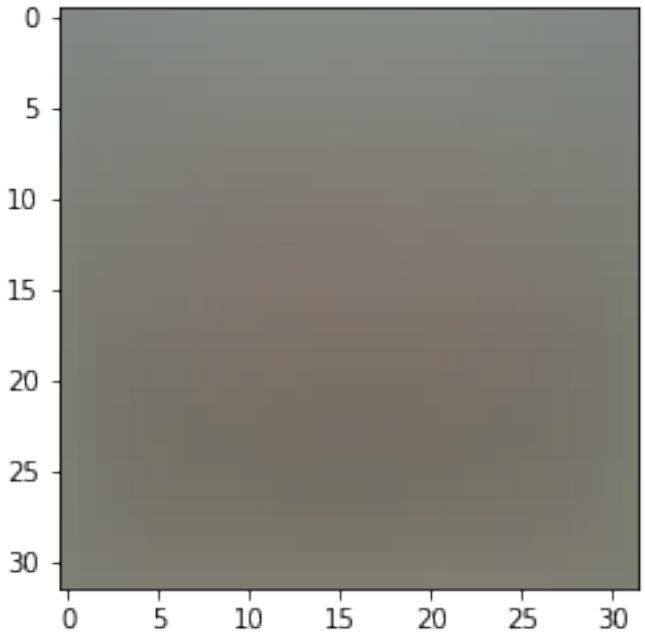
[8]:
```
# Preprocessing: reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_val = np.reshape(X_val, (X_val.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

# 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)
```

[9]:
```
# 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()

# 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

# 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)
```

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




```
(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 `svm_loss_naive` which uses for loops to evaluate the multiclass SVM loss function.

[10]:
```
# 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: 8.847756
```

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:

[11]:
```
# 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: -11.780734 analytic: -11.780734, relative error: 1.577587e-12
numerical: 25.528556 analytic: 25.528556, relative error: 7.098036e-12
numerical: 9.749872 analytic: 9.749872, relative error: 1.789498e-11
```

```
numerical: 8.295182 analytic: 8.295182, relative error: 2.306322e-12
numerical: -0.547659 analytic: -0.547659, relative error: 4.202489e-10
numerical: -2.418568 analytic: -2.418568, relative error: 4.183493e-11
numerical: 17.810653 analytic: 17.810653, relative error: 9.737190e-13
numerical: 16.419163 analytic: 16.419163, relative error: 1.747583e-11
numerical: 21.360998 analytic: 21.360998, relative error: 7.251954e-12
numerical: 21.960019 analytic: 21.960019, relative error: 2.292647e-11
numerical: 12.020374 analytic: 12.020374, relative error: 4.332506e-12
numerical: 22.988022 analytic: 22.988022, relative error: 3.641806e-12
numerical: 20.154404 analytic: 20.154404, relative error: 1.896981e-12
numerical: -39.906959 analytic: -39.906959, relative error: 1.660555e-11
numerical: -25.930947 analytic: -25.930947, relative error: 3.003941e-12
numerical: 14.455705 analytic: 14.455705, relative error: 1.272764e-11
numerical: -19.968771 analytic: -19.968771, relative error: 8.333290e-12
numerical: 16.612836 analytic: 16.612836, relative error: 3.646844e-12
numerical: 10.775713 analytic: 10.775713, relative error: 4.346594e-11
numerical: -12.709444 analytic: -12.709444, relative error: 1.246334e-11
```

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 :

Because we use max function for the loss function, so we can't calculate the gradient when s_j is similar with s_y[i]+1.

For example, let the score of correct class is 3 and score of the image that we want to calculate is 2. Then, analytic gradient is 0 because max(0,2-3+1)=0. But, numerical gradient is not 0 for any little step size.

So it causes dicrepancy between numerical and analytic gradient value.

[12]:
```
# 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: 8.847756e+00 computed in 0.302192s
(500,)
Vectorized loss: 8.847756e+00 computed in 0.013804s
difference: -0.000000
```

[13]:
```
# 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 0.317342s
(500,)
Vectorized loss and gradient: computed in 0.012871s
difference: 0.000000
```

1.2.1 Stochastic Gradient Descent

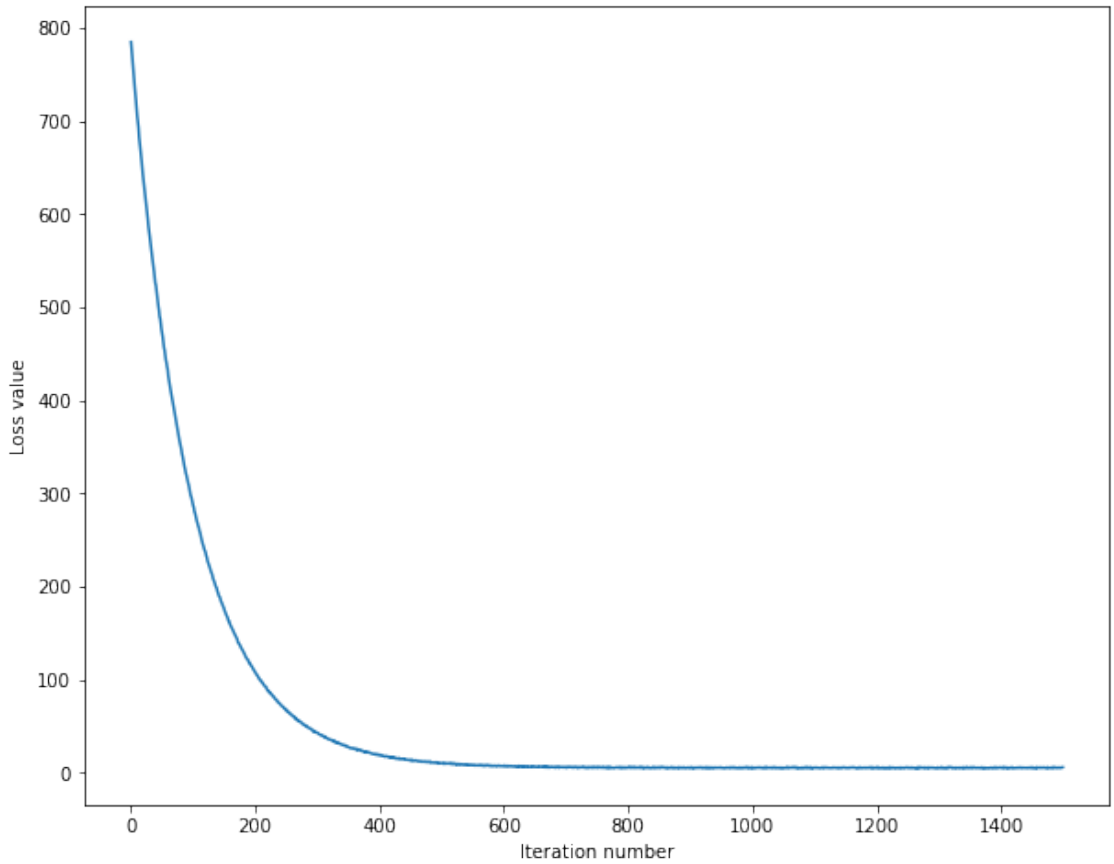
We now have vectorized and efficient expressions for the loss, the gradient and our gradient matches the numerical gradient. We are therefore ready to do SGD to minimize the loss. Your code for this part will be written inside `cs231n/classifiers/linear_classifier.py`.

[22]:
```
# 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 784.973088
iteration 100 / 1500: loss 286.860503
```

```
iteration 200 / 1500: loss 107.616416
iteration 300 / 1500: loss 42.924279
iteration 400 / 1500: loss 19.341109
iteration 500 / 1500: loss 10.183154
iteration 600 / 1500: loss 7.188049
iteration 700 / 1500: loss 5.507264
iteration 800 / 1500: loss 5.614595
iteration 900 / 1500: loss 5.882847
iteration 1000 / 1500: loss 5.607397
iteration 1100 / 1500: loss 5.484139
iteration 1200 / 1500: loss 5.796090
iteration 1300 / 1500: loss 5.244871
iteration 1400 / 1500: loss 5.521071
That took 8.991618s
```

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



[34]:
```
# 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: 0.369224
validation accuracy: 0.384000
```

[65]:
```
# 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 0.39 on the validation set.

# Note: you may see runtime/overflow warnings during hyper-parameter search.
# This may be caused by extreme values, and is not a bug.

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

# Provided as a reference. You may or may not want to change these hyperparameters
learning_rates = [1.25e-7, 1e-7, 0.75e-7]
regularization_strengths = [2.5e4, 2.75e4, 3e4]

# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
for l in learning_rates:
    for r in regularization_strengths:
        svm = LinearSVM()
        svm.train(X_train, y_train, l, r,
                      num_iters=1500, verbose=True)
        train_ac = np.mean(y_train == svm.predict(X_train))
        val_ac = np.mean(y_val == svm.predict(X_val))
        if val_ac > best_val:
          best_val, best_svm = val_ac, svm
        results[(l,r)] = (train_ac ,val_ac)
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

# 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 788.114971
iteration 100 / 1500: loss 223.552274
iteration 200 / 1500: loss 67.160283
iteration 300 / 1500: loss 22.895278
iteration 400 / 1500: loss 9.741601
iteration 500 / 1500: loss 6.827411
iteration 600 / 1500: loss 5.606782
iteration 700 / 1500: loss 5.113238
iteration 800 / 1500: loss 5.010355
iteration 900 / 1500: loss 5.364078
```

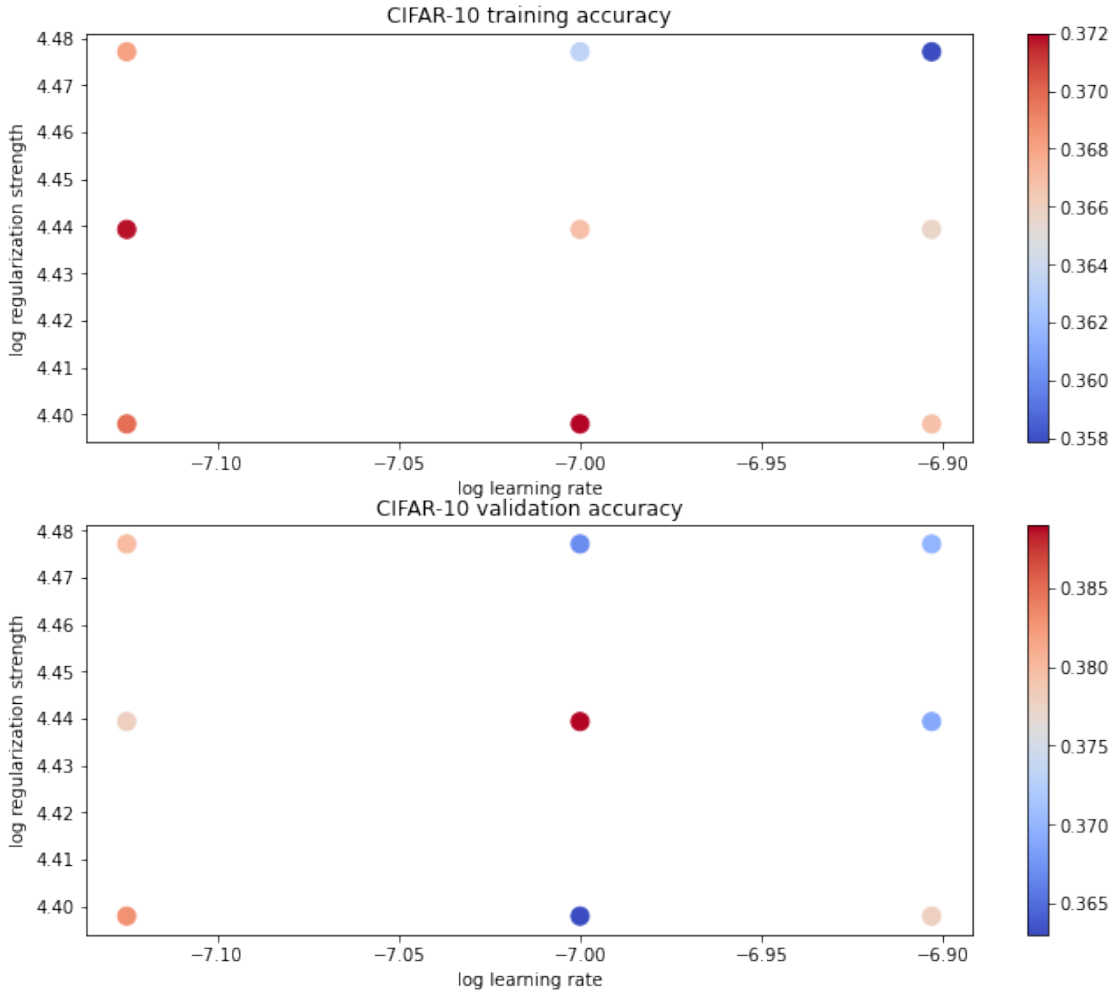
```
iteration 1000 / 1500: loss 5.115651
iteration 1100 / 1500: loss 4.950481
iteration 1200 / 1500: loss 5.107855
iteration 1300 / 1500: loss 4.874827
iteration 1400 / 1500: loss 5.352197
iteration 0 / 1500: loss 853.634472
iteration 100 / 1500: loss 214.703888
iteration 200 / 1500: loss 57.582088
iteration 300 / 1500: loss 18.661031
iteration 400 / 1500: loss 8.473411
iteration 500 / 1500: loss 5.877934
iteration 600 / 1500: loss 5.537184
iteration 700 / 1500: loss 5.519028
iteration 800 / 1500: loss 5.568531
iteration 900 / 1500: loss 5.546619
iteration 1000 / 1500: loss 5.037852
iteration 1100 / 1500: loss 5.693450
iteration 1200 / 1500: loss 5.572251
iteration 1300 / 1500: loss 5.150330
iteration 1400 / 1500: loss 4.962823
iteration 0 / 1500: loss 940.641253
iteration 100 / 1500: loss 209.002415
iteration 200 / 1500: loss 49.668442
iteration 300 / 1500: loss 15.113373
iteration 400 / 1500: loss 7.824332
iteration 500 / 1500: loss 6.086913
iteration 600 / 1500: loss 5.609299
iteration 700 / 1500: loss 5.433150
iteration 800 / 1500: loss 5.488670
iteration 900 / 1500: loss 5.354380
iteration 1000 / 1500: loss 5.492699
iteration 1100 / 1500: loss 5.245927
iteration 1200 / 1500: loss 5.478512
iteration 1300 / 1500: loss 5.391665
iteration 1400 / 1500: loss 5.538042
iteration 0 / 1500: loss 793.320524
iteration 100 / 1500: loss 288.041052
iteration 200 / 1500: loss 108.798471
iteration 300 / 1500: loss 42.399756
iteration 400 / 1500: loss 19.266375
iteration 500 / 1500: loss 10.311406
iteration 600 / 1500: loss 7.134507
iteration 700 / 1500: loss 5.779169
iteration 800 / 1500: loss 5.758518
iteration 900 / 1500: loss 5.976259
iteration 1000 / 1500: loss 5.658806
iteration 1100 / 1500: loss 5.716194
iteration 1200 / 1500: loss 5.780230
```

```
iteration 1300 / 1500: loss 5.280702
iteration 1400 / 1500: loss 5.009076
iteration 0 / 1500: loss 875.145103
iteration 100 / 1500: loss 287.874245
iteration 200 / 1500: loss 98.414427
iteration 300 / 1500: loss 36.170020
iteration 400 / 1500: loss 15.030083
iteration 500 / 1500: loss 8.801644
iteration 600 / 1500: loss 6.137325
iteration 700 / 1500: loss 5.906501
iteration 800 / 1500: loss 5.224420
iteration 900 / 1500: loss 5.548555
iteration 1000 / 1500: loss 5.282471
iteration 1100 / 1500: loss 5.527352
iteration 1200 / 1500: loss 5.802646
iteration 1300 / 1500: loss 5.540638
iteration 1400 / 1500: loss 5.226772
iteration 0 / 1500: loss 939.007419
iteration 100 / 1500: loss 281.821125
iteration 200 / 1500: loss 87.423546
iteration 300 / 1500: loss 29.487443
iteration 400 / 1500: loss 12.210920
iteration 500 / 1500: loss 7.111797
iteration 600 / 1500: loss 5.705578
iteration 700 / 1500: loss 5.187601
iteration 800 / 1500: loss 5.710288
iteration 900 / 1500: loss 5.603275
iteration 1000 / 1500: loss 5.608854
iteration 1100 / 1500: loss 5.279017
iteration 1200 / 1500: loss 5.024830
iteration 1300 / 1500: loss 4.986192
iteration 1400 / 1500: loss 5.557469
iteration 0 / 1500: loss 793.310508
iteration 100 / 1500: loss 368.838209
iteration 200 / 1500: loss 175.738339
iteration 300 / 1500: loss 85.208342
iteration 400 / 1500: loss 42.335263
iteration 500 / 1500: loss 22.597874
iteration 600 / 1500: loss 13.514496
iteration 700 / 1500: loss 9.183597
iteration 800 / 1500: loss 7.497890
iteration 900 / 1500: loss 6.359911
iteration 1000 / 1500: loss 6.212342
iteration 1100 / 1500: loss 5.637713
iteration 1200 / 1500: loss 5.308323
iteration 1300 / 1500: loss 5.338490
iteration 1400 / 1500: loss 5.293857
iteration 0 / 1500: loss 870.242929
```

```
iteration 100 / 1500: loss 379.849481
iteration 200 / 1500: loss 167.730369
iteration 300 / 1500: loss 75.648208
iteration 400 / 1500: loss 36.680059
iteration 500 / 1500: loss 18.701012
iteration 600 / 1500: loss 11.141353
iteration 700 / 1500: loss 8.231157
iteration 800 / 1500: loss 6.093937
iteration 900 / 1500: loss 5.749710
iteration 1000 / 1500: loss 5.782350
iteration 1100 / 1500: loss 5.748016
iteration 1200 / 1500: loss 5.411621
iteration 1300 / 1500: loss 5.284593
iteration 1400 / 1500: loss 5.177544
iteration 0 / 1500: loss 950.309734
iteration 100 / 1500: loss 385.993393
iteration 200 / 1500: loss 159.100879
iteration 300 / 1500: loss 66.656609
iteration 400 / 1500: loss 30.445444
iteration 500 / 1500: loss 15.249425
iteration 600 / 1500: loss 9.199866
iteration 700 / 1500: loss 6.885465
iteration 800 / 1500: loss 6.010110
iteration 900 / 1500: loss 5.511161
iteration 1000 / 1500: loss 5.708703
iteration 1100 / 1500: loss 5.060804
iteration 1200 / 1500: loss 5.713410
iteration 1300 / 1500: loss 5.486264
iteration 1400 / 1500: loss 5.253848
lr 7.500000e-08 reg 2.500000e+04 train accuracy: 0.369816 val accuracy: 0.383000
lr 7.500000e-08 reg 2.750000e+04 train accuracy: 0.371816 val accuracy: 0.378000
lr 7.500000e-08 reg 3.000000e+04 train accuracy: 0.368163 val accuracy: 0.380000
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.372000 val accuracy: 0.363000
lr 1.000000e-07 reg 2.750000e+04 train accuracy: 0.366898 val accuracy: 0.389000
lr 1.000000e-07 reg 3.000000e+04 train accuracy: 0.363469 val accuracy: 0.367000
lr 1.250000e-07 reg 2.500000e+04 train accuracy: 0.366837 val accuracy: 0.378000
lr 1.250000e-07 reg 2.750000e+04 train accuracy: 0.365673 val accuracy: 0.369000
lr 1.250000e-07 reg 3.000000e+04 train accuracy: 0.357837 val accuracy: 0.370000
best validation accuracy achieved during cross-validation: 0.389000
```

[66]:
```python
# Visualize the cross-validation results
import math
import pdb

# pdb.set_trace()

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.tight_layout(pad=3)
plt.scatter(x_scatter, y_scatter, marker_size, c=colors, cmap=plt.cm.coolwarm)
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, cmap=plt.cm.coolwarm)
plt.colorbar()
plt.xlabel('log learning rate')
plt.ylabel('log regularization strength')
plt.title('CIFAR-10 validation accuracy')
plt.show()
```

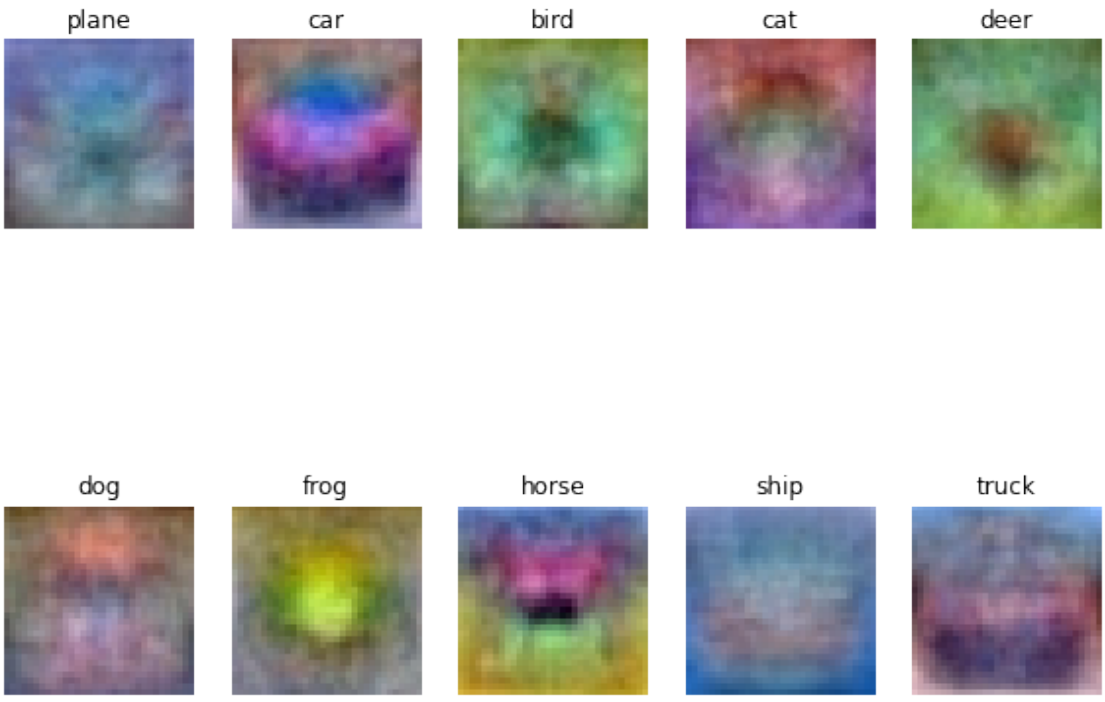


[67]:
```
# Evaluate the best svm on test set
y_test_pred = best_svm.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
```

```
linear SVM on raw pixels final test set accuracy: 0.357000
```

[68]:
```
# 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', 'truck']
```

```
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])
```



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 :

We can see the average value of image in visualized SVM weights. Due to score is the dot product of the samples and the weight, so weigth must be similar to the samples to get the high score.

softmax

August 30, 2021

[1]:
```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

# Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment1/'
FOLDERNAME = 'cs231n/assignments/assignment1/'
assert FOLDERNAME is not None, "[!] Enter the foldername."

# Now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# This downloads the CIFAR-10 dataset to your Drive
# if it doesn't already exist.
%cd drive/My\ Drive/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/cs231n/assignments/assignment1/cs231n/datasets
/content/drive/My Drive/cs231n/assignments/assignment1
```

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

[2]:
```
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading extenrnal modules
# see http://stackoverflow.com/questions/1907993/
 ↪autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

[3]:
```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000,␣
 ↪num_dev=500):
    """
    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'

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

    X_train, y_train, X_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


# 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`.

[13]:
```
# 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: 2.414284
sanity check: 2.302585
```

Inline Question 1

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

Your Answer :

Because we initialized W with small values, all the scores are similar to 0. So, each Li is very similar to -log(1/N). As a result, our loss function's value is 1/N* N *(-log(1/N)) = -log(1/N) (when we don't consider regulation). We know N is 10 in our datasets, we can expect our loss to be close to -log(0.1)

Futrthermore when we change W 10 times bigger, loss function value isn't close to -log(0.1)

[15]:
```
# 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: 3.578903 analytic: 3.578903, relative error: 1.202492e-09
numerical: 2.557805 analytic: 2.557805, relative error: 4.697598e-09
```

```
numerical: 2.507119 analytic: 2.507119, relative error: 2.749359e-08
numerical: -0.272929 analytic: -0.272929, relative error: 9.043610e-09
numerical: 5.028566 analytic: 5.028566, relative error: 3.966600e-09
numerical: -1.204780 analytic: -1.204780, relative error: 7.338380e-09
numerical: -0.149096 analytic: -0.149096, relative error: 1.661144e-07
numerical: 4.736987 analytic: 4.736987, relative error: 2.224166e-08
numerical: 0.943974 analytic: 0.943974, relative error: 3.547894e-08
numerical: 2.508121 analytic: 2.508121, relative error: 1.398725e-08
numerical: 0.106883 analytic: 0.106883, relative error: 1.233612e-07
numerical: 1.145273 analytic: 1.145273, relative error: 1.576722e-08
numerical: 1.833693 analytic: 1.833693, relative error: 2.323541e-09
numerical: 0.125882 analytic: 0.125882, relative error: 3.369794e-07
numerical: -1.324151 analytic: -1.324151, relative error: 3.729933e-08
numerical: 0.114384 analytic: 0.114384, relative error: 2.915620e-07
numerical: -0.365236 analytic: -0.365236, relative error: 2.798728e-08
numerical: -3.051455 analytic: -3.051456, relative error: 1.015053e-08
numerical: -0.487751 analytic: -0.487751, relative error: 2.692739e-08
numerical: -1.292873 analytic: -1.292874, relative error: 1.747619e-08
```

[45]:
```
# Now that we have a naive implementation of the softmax loss function and its gradient,
# implement a vectorized version in softmax_loss_vectorized.
# The two versions should compute the same results, but the vectorized version should be
# 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: 2.414284e+00 computed in 0.139150s
vectorized loss: 2.414284e+00 computed in 0.010219s
Loss difference: 0.000000
Gradient difference: 0.000000
```

[60]:
```
# 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

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

# Provided as a reference. You may or may not want to change these hyperparameters
learning_rates = [5e-6, 1e-6, 5e-7]
regularization_strengths = [0.5e3, 1e3, 1.5e3]

# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
for l in learning_rates:
    for r in regularization_strengths:
        softmax = Softmax()
        softmax.train(X_train, y_train, l, r,
                      num_iters=2000, verbose=True)
        train_ac = np.mean(y_train == softmax.predict(X_train))
        val_ac = np.mean(y_val == softmax.predict(X_val))
        if val_ac > best_val:
          best_val, best_softmax = val_ac, softmax
        results[(l,r)] = (train_ac ,val_ac)
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

# 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 20.154744
iteration 100 / 1500: loss 7.654575
iteration 200 / 1500: loss 3.756029
iteration 300 / 1500: loss 2.704748
iteration 400 / 1500: loss 2.181024
iteration 500 / 1500: loss 2.008781
iteration 600 / 1500: loss 2.159320
iteration 700 / 1500: loss 1.944106
iteration 800 / 1500: loss 1.907556
iteration 900 / 1500: loss 1.851227
iteration 1000 / 1500: loss 1.775236
iteration 1100 / 1500: loss 1.902691
iteration 1200 / 1500: loss 1.951581
iteration 1300 / 1500: loss 1.901259
iteration 1400 / 1500: loss 1.987837
iteration 0 / 1500: loss 36.412425
iteration 100 / 1500: loss 5.957800
iteration 200 / 1500: loss 2.433534
iteration 300 / 1500: loss 1.997166
iteration 400 / 1500: loss 1.850647
iteration 500 / 1500: loss 2.034861
iteration 600 / 1500: loss 2.028856
iteration 700 / 1500: loss 1.883997
iteration 800 / 1500: loss 1.973304
iteration 900 / 1500: loss 1.900555
iteration 1000 / 1500: loss 2.223199
iteration 1100 / 1500: loss 1.951945
iteration 1200 / 1500: loss 1.959766
iteration 1300 / 1500: loss 1.977278
iteration 1400 / 1500: loss 1.887709
iteration 0 / 1500: loss 51.799640
iteration 100 / 1500: loss 3.972382
iteration 200 / 1500: loss 2.244665
iteration 300 / 1500: loss 1.925247
iteration 400 / 1500: loss 1.925781
iteration 500 / 1500: loss 1.970497
iteration 600 / 1500: loss 2.062422
iteration 700 / 1500: loss 2.117807
iteration 800 / 1500: loss 1.979888
iteration 900 / 1500: loss 1.886368
iteration 1000 / 1500: loss 2.415300
iteration 1100 / 1500: loss 1.951573
iteration 1200 / 1500: loss 2.101487
iteration 1300 / 1500: loss 2.061885
iteration 1400 / 1500: loss 1.957854
iteration 0 / 1500: loss 21.804186
iteration 100 / 1500: loss 15.191332
iteration 200 / 1500: loss 12.413625
```

```
iteration 300 / 1500: loss 10.329120
iteration 400 / 1500: loss 8.808767
iteration 500 / 1500: loss 7.464215
iteration 600 / 1500: loss 6.327647
iteration 700 / 1500: loss 5.469445
iteration 800 / 1500: loss 4.738428
iteration 900 / 1500: loss 4.292715
iteration 1000 / 1500: loss 3.774708
iteration 1100 / 1500: loss 3.365075
iteration 1200 / 1500: loss 3.157794
iteration 1300 / 1500: loss 2.914793
iteration 1400 / 1500: loss 2.670965
iteration 0 / 1500: loss 36.347329
iteration 100 / 1500: loss 22.791852
iteration 200 / 1500: loss 15.582603
iteration 300 / 1500: loss 10.922232
iteration 400 / 1500: loss 8.041158
iteration 500 / 1500: loss 5.933289
iteration 600 / 1500: loss 4.476594
iteration 700 / 1500: loss 3.555823
iteration 800 / 1500: loss 3.025759
iteration 900 / 1500: loss 2.560694
iteration 1000 / 1500: loss 2.254269
iteration 1100 / 1500: loss 2.166689
iteration 1200 / 1500: loss 2.041229
iteration 1300 / 1500: loss 2.011979
iteration 1400 / 1500: loss 1.940934
iteration 0 / 1500: loss 52.180839
iteration 100 / 1500: loss 27.359939
iteration 200 / 1500: loss 15.548226
iteration 300 / 1500: loss 9.323219
iteration 400 / 1500: loss 5.887010
iteration 500 / 1500: loss 4.080118
iteration 600 / 1500: loss 3.077334
iteration 700 / 1500: loss 2.467374
iteration 800 / 1500: loss 2.230121
iteration 900 / 1500: loss 1.996212
iteration 1000 / 1500: loss 2.127441
iteration 1100 / 1500: loss 2.025435
iteration 1200 / 1500: loss 1.926282
iteration 1300 / 1500: loss 1.852812
iteration 1400 / 1500: loss 1.894990
iteration 0 / 1500: loss 20.391352
iteration 100 / 1500: loss 16.764815
iteration 200 / 1500: loss 14.891487
iteration 300 / 1500: loss 13.622386
iteration 400 / 1500: loss 12.257478
iteration 500 / 1500: loss 11.211284
```

```
iteration 600 / 1500: loss 10.311616
iteration 700 / 1500: loss 9.405125
iteration 800 / 1500: loss 8.626100
iteration 900 / 1500: loss 7.944562
iteration 1000 / 1500: loss 7.348161
iteration 1100 / 1500: loss 6.805979
iteration 1200 / 1500: loss 6.380544
iteration 1300 / 1500: loss 5.835416
iteration 1400 / 1500: loss 5.390236
iteration 0 / 1500: loss 35.047511
iteration 100 / 1500: loss 27.610921
iteration 200 / 1500: loss 22.632089
iteration 300 / 1500: loss 18.708014
iteration 400 / 1500: loss 15.407279
iteration 500 / 1500: loss 12.908937
iteration 600 / 1500: loss 10.874313
iteration 700 / 1500: loss 9.176222
iteration 800 / 1500: loss 7.901540
iteration 900 / 1500: loss 6.748132
iteration 1000 / 1500: loss 5.850846
iteration 1100 / 1500: loss 5.231605
iteration 1200 / 1500: loss 4.594727
iteration 1300 / 1500: loss 4.049937
iteration 1400 / 1500: loss 3.670933
iteration 0 / 1500: loss 50.201231
iteration 100 / 1500: loss 35.761009
iteration 200 / 1500: loss 26.716294
iteration 300 / 1500: loss 20.055501
iteration 400 / 1500: loss 15.291259
iteration 500 / 1500: loss 11.713602
iteration 600 / 1500: loss 9.186582
iteration 700 / 1500: loss 7.183835
iteration 800 / 1500: loss 5.744672
iteration 900 / 1500: loss 4.803507
iteration 1000 / 1500: loss 3.955932
iteration 1100 / 1500: loss 3.420421
iteration 1200 / 1500: loss 3.025710
iteration 1300 / 1500: loss 2.705520
iteration 1400 / 1500: loss 2.533115
lr 5.000000e-07 reg 5.000000e+02 train accuracy: 0.365531 val accuracy: 0.348000
lr 5.000000e-07 reg 1.000000e+03 train accuracy: 0.384122 val accuracy: 0.391000
lr 5.000000e-07 reg 1.500000e+03 train accuracy: 0.390224 val accuracy: 0.391000
lr 1.000000e-06 reg 5.000000e+02 train accuracy: 0.402408 val accuracy: 0.400000
lr 1.000000e-06 reg 1.000000e+03 train accuracy: 0.400673 val accuracy: 0.397000
lr 1.000000e-06 reg 1.500000e+03 train accuracy: 0.394980 val accuracy: 0.382000
lr 5.000000e-06 reg 5.000000e+02 train accuracy: 0.356898 val accuracy: 0.370000
lr 5.000000e-06 reg 1.000000e+03 train accuracy: 0.365796 val accuracy: 0.346000
lr 5.000000e-06 reg 1.500000e+03 train accuracy: 0.307878 val accuracy: 0.310000
```

```
best validation accuracy achieved during cross-validation: 0.400000
```

[61]:
```
# 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: 0.384000
```

Inline Question 2 - *True or False*

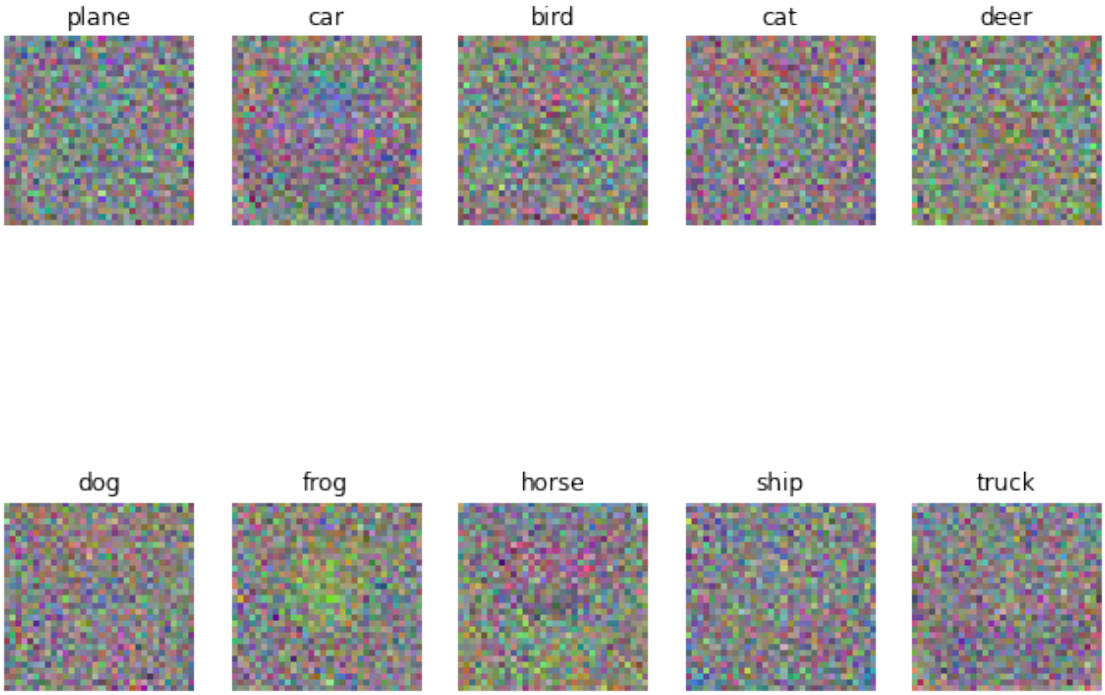
Suppose the overall training loss is defined as the sum of the per-datapoint loss over all training examples. It is possible to add a new datapoint to a training set that would leave the SVM loss unchanged, but this is not the case with the Softmax classifier loss.

YourAnswer : True

YourExplanation :

The statement is true because when we put new datapoint to training set, SVM loss doesn't change beacuse SVM never developes itself when classifier's performance exceed some value. But softmax classifier loss will change becuase this classifier developes itself until the probability reach 1.

[63]:
```
# 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', 'truck']
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

August 30, 2021

[1]:
```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

# Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment1/'
FOLDERNAME = 'cs231n/assignments/assignment1/'
assert FOLDERNAME is not None, "[!] Enter the foldername."

# Now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# This downloads the CIFAR-10 dataset to your Drive
# if it doesn't already exist.
%cd drive/My\ Drive/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/cs231n/assignments/assignment1/cs231n/datasets
/content/drive/My Drive/cs231n/assignments/assignment1
```

1 Fully-Connected Neural Nets

In this exercise we will implement fully-connected networks using a modular approach. For each layer we will implement a `forward` and a `backward` function. The `forward` function will receive inputs, weights, and other parameters and will return both an output and a `cache` object storing data needed for the backward pass, like this:

```
def layer_forward(x, w):
  """ Receive inputs x and weights w """
  # Do some computations ...
  z = # ... some intermediate value
```

```
  # Do some more computations ...
  out = # the output

  cache = (x, w, z, out) # Values we need to compute gradients

  return out, cache
```

The backward pass will receive upstream derivatives and the `cache` object, and will return gradients with respect to the inputs and weights, like this:

```
def layer_backward(dout, cache):
  """
  Receive dout (derivative of loss with respect to outputs) and cache,
  and compute derivative with respect to inputs.
  """
  # Unpack cache values
  x, w, z, out = cache

  # Use values in cache to compute derivatives
  dx = # Derivative of loss with respect to x
  dw = # Derivative of loss with respect to w

  return dx, dw
```

After implementing a bunch of layers this way, we will be able to easily combine them to build classifiers with different architectures.

[12]:
```
# As usual, a bit of setup
from __future__ import print_function
import time
import numpy as np
import matplotlib.pyplot as plt
from cs231n.classifiers.fc_net import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient,␣
 ↪eval_numerical_gradient_array
from cs231n.solver import Solver

%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))))
```

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

[11]:
```
# Load the (preprocessed) CIFAR10 data.

data = get_CIFAR10_data()
for k, v in list(data.items()):
  print(('%s: ' % k, v.shape))
```

```
('X_train: ', (49000, 3, 32, 32))
('y_train: ', (49000,))
('X_val: ', (1000, 3, 32, 32))
('y_val: ', (1000,))
('X_test: ', (1000, 3, 32, 32))
('y_test: ', (1000,))
```

2 Affine layer: forward

Open the file `cs231n/layers.py` and implement the `affine_forward` function.

Once you are done you can test your implementaion by running the following:

[3]:
```
# Test the affine_forward function

num_inputs = 2
input_shape = (4, 5, 6)
output_dim = 3

input_size = num_inputs * np.prod(input_shape)
weight_size = output_dim * np.prod(input_shape)

x = np.linspace(-0.1, 0.5, num=input_size).reshape(num_inputs, *input_shape)
w = np.linspace(-0.2, 0.3, num=weight_size).reshape(np.prod(input_shape),␣
 ↪output_dim)
b = np.linspace(-0.3, 0.1, num=output_dim)

out, _ = affine_forward(x, w, b)
correct_out = np.array([[ 1.49834967,  1.70660132,  1.91485297],
                        [ 3.25553199,  3.5141327,   3.77273342]])

# Compare your output with ours. The error should be around e-9 or less.
print('Testing affine_forward function:')
print('difference: ', rel_error(out, correct_out))
```

```
Testing affine_forward function:
difference:  9.769849468192957e-10
```

3 Affine layer: backward

Now implement the `affine_backward` function and test your implementation using numeric gradient checking.

[4]:
```
# Test the affine_backward function
np.random.seed(231)
x = np.random.randn(10, 2, 3)
w = np.random.randn(6, 5)
b = np.random.randn(5)
dout = np.random.randn(10, 5)

dx_num = eval_numerical_gradient_array(lambda x: affine_forward(x, w, b)[0], x,␣
 ↪dout)
dw_num = eval_numerical_gradient_array(lambda w: affine_forward(x, w, b)[0], w,␣
 ↪dout)
db_num = eval_numerical_gradient_array(lambda b: affine_forward(x, w, b)[0], b,␣
 ↪dout)

_, cache = affine_forward(x, w, b)
dx, dw, db = affine_backward(dout, cache)

# The error should be around e-10 or less
print('Testing affine_backward function:')
print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

```
Testing affine_backward function:
dx error:  5.399100368651805e-11
dw error:  9.904211865398145e-11
db error:  2.4122867568119087e-11
```

4 ReLU activation: forward

Implement the forward pass for the ReLU activation function in the `relu_forward` function and test your implementation using the following:

[5]:
```
# Test the relu_forward function

x = np.linspace(-0.5, 0.5, num=12).reshape(3, 4)

out, _ = relu_forward(x)
correct_out = np.array([[ 0.,          0.,          0.,          0.,        ],
                        [ 0.,          0.,          0.04545455,  0.13636364,],
```

```
                        [ 0.22727273,  0.31818182,  0.40909091,  0.5,       ]])

# Compare your output with ours. The error should be on the order of e-8
print('Testing relu_forward function:')
print('difference: ', rel_error(out, correct_out))
```

```
Testing relu_forward function:
difference:  4.999999798022158e-08
```

5 ReLU activation: backward

Now implement the backward pass for the ReLU activation function in the `relu_backward` function and test your implementation using numeric gradient checking:

[6]:
```
np.random.seed(231)
x = np.random.randn(10, 10)
dout = np.random.randn(*x.shape)

dx_num = eval_numerical_gradient_array(lambda x: relu_forward(x)[0], x, dout)

_, cache = relu_forward(x)
dx = relu_backward(dout, cache)

# The error should be on the order of e-12
print('Testing relu_backward function:')
print('dx error: ', rel_error(dx_num, dx))
```

```
Testing relu_backward function:
dx error:  3.2756349136310288e-12
```

5.1 Inline Question 1:

We've only asked you to implement ReLU, but there are a number of different activation functions that one could use in neural networks, each with its pros and cons. In particular, an issue commonly seen with activation functions is getting zero (or close to zero) gradient flow during backpropagation. Which of the following activation functions have this problem? If you consider these functions in the one dimensional case, what types of input would lead to this behaviour? 1. Sigmoid 2. ReLU 3. Leaky ReLU

5.2 Answer:

1 - If the input value is too big or too small, result after pass the Sigmoid is close to either 1 or -1. Gradient is near 0 at thos saturated position, so this case give 0 upstream gradient during backpropagation.

2 - For positive input, Relu don't give 0 gradient. But for negative and 0 input, 0 gradient flows to the next backpropagation step.

3 - Because of the small value slope in negative area, this activation functino get never saturated. So, this activation func. doesn't make that issue.

6 "Sandwich" layers

There are some common patterns of layers that are frequently used in neural nets. For example, affine layers are frequently followed by a ReLU nonlinearity. To make these common patterns easy, we define several convenience layers in the file `cs231n/layer_utils.py`.

For now take a look at the `affine_relu_forward` and `affine_relu_backward` functions, and run the following to numerically gradient check the backward pass:

[7]:
```
from cs231n.layer_utils import affine_relu_forward, affine_relu_backward
np.random.seed(231)
x = np.random.randn(2, 3, 4)
w = np.random.randn(12, 10)
b = np.random.randn(10)
dout = np.random.randn(2, 10)

out, cache = affine_relu_forward(x, w, b)
dx, dw, db = affine_relu_backward(dout, cache)

dx_num = eval_numerical_gradient_array(lambda x: affine_relu_forward(x, w,␣
 ↪b)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: affine_relu_forward(x, w,␣
 ↪b)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: affine_relu_forward(x, w,␣
 ↪b)[0], b, dout)

# Relative error should be around e-10 or less
print('Testing affine_relu_forward and affine_relu_backward:')
print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

```
Testing affine_relu_forward and affine_relu_backward:
dx error:  2.299579177309368e-11
dw error:  8.162011105764925e-11
db error:  7.826724021458994e-12
```

7 Loss layers: Softmax and SVM

Now implement the loss and gradient for softmax and SVM in the `softmax_loss` and `svm_loss` function in `cs231n/layers.py`. These should be similar to what you implemented in `cs231n/classifiers/softmax.py` and `cs231n/classifiers/linear_svm.py`.

You can make sure that the implementations are correct by running the following:

[8]:
```
np.random.seed(231)
num_classes, num_inputs = 10, 50
x = 0.001 * np.random.randn(num_inputs, num_classes)
y = np.random.randint(num_classes, size=num_inputs)
```

```
dx_num = eval_numerical_gradient(lambda x: svm_loss(x, y)[0], x, verbose=False)
loss, dx = svm_loss(x, y)

# Test sum_loss function. Loss should be around 9 and dx error should be around␣
 ↪the order of e-9
print('Testing svm_loss:')
print('loss: ', loss)
print('dx error: ', rel_error(dx_num, dx))

dx_num = eval_numerical_gradient(lambda x: softmax_loss(x, y)[0], x,␣
 ↪verbose=False)
loss, dx = softmax_loss(x, y)

# Test softmax_loss function. Loss should be close to 2.3 and dx error should␣
 ↪be around e-8
print('\nTesting softmax_loss:')
print('loss: ', loss)
print('dx error: ', rel_error(dx_num, dx))
```

```
Testing svm_loss:
loss:  8.999602749096233
dx error:  1.4021566006651672e-09

Testing softmax_loss:
loss:  2.302545844500738
dx error:  9.483503037636722e-09
```

8 Two-layer network

Open the file `cs231n/classifiers/fc_net.py` and complete the implementation of the `TwoLayerNet` class. Read through it to make sure you understand the API. You can run the cell below to test your implementation.

[9]:
```
np.random.seed(231)
N, D, H, C = 3, 5, 50, 7
X = np.random.randn(N, D)
y = np.random.randint(C, size=N)

std = 1e-3
model = TwoLayerNet(input_dim=D, hidden_dim=H, num_classes=C, weight_scale=std)

print('Testing initialization ... ')
W1_std = abs(model.params['W1'].std() - std)
b1 = model.params['b1']
W2_std = abs(model.params['W2'].std() - std)
b2 = model.params['b2']
assert W1_std < std / 10, 'First layer weights do not seem right'
```

```
assert np.all(b1 == 0), 'First layer biases do not seem right'
assert W2_std < std / 10, 'Second layer weights do not seem right'
assert np.all(b2 == 0), 'Second layer biases do not seem right'

print('Testing test-time forward pass ... ')
model.params['W1'] = np.linspace(-0.7, 0.3, num=D*H).reshape(D, H)
model.params['b1'] = np.linspace(-0.1, 0.9, num=H)
model.params['W2'] = np.linspace(-0.3, 0.4, num=H*C).reshape(H, C)
model.params['b2'] = np.linspace(-0.9, 0.1, num=C)
X = np.linspace(-5.5, 4.5, num=N*D).reshape(D, N).T
scores = model.loss(X)
correct_scores = np.asarray(
  [[11.53165108,  12.2917344,   13.05181771,  13.81190102,  14.57198434, 15.
 ↪33206765,  16.09215096],
   [12.05769098,  12.74614105,  13.43459113,  14.1230412,   14.81149128, 15.
 ↪49994135,  16.18839143],
   [12.58373087,  13.20054771,  13.81736455,  14.43418138,  15.05099822, 15.
 ↪66781506,  16.2846319 ]])
scores_diff = np.abs(scores - correct_scores).sum()
assert scores_diff < 1e-6, 'Problem with test-time forward pass'
print('Testing training loss (no regularization)')
y = np.asarray([0, 5, 1])
loss, grads = model.loss(X, y)
correct_loss = 3.4702243556
assert abs(loss - correct_loss) < 1e-10, 'Problem with training-time loss'

model.reg = 1.0
loss, grads = model.loss(X, y)
correct_loss = 26.5948426952
assert abs(loss - correct_loss) < 1e-10, 'Problem with regularization loss'

# Errors should be around e-7 or less
for reg in [0.0, 0.7]:
  print('Running numeric gradient check with reg = ', reg)
  model.reg = reg
  loss, grads = model.loss(X, y)

  for name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    grad_num = eval_numerical_gradient(f, model.params[name], verbose=False)
    print('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
```

```
Testing initialization ...
Testing test-time forward pass ...
Testing training loss (no regularization)
Running numeric gradient check with reg =  0.0
W1 relative error: 1.83e-08
```

```
W2 relative error: 3.31e-10
b1 relative error: 9.83e-09
b2 relative error: 4.33e-10
Running numeric gradient check with reg =  0.7
W1 relative error: 2.53e-07
W2 relative error: 2.85e-08
b1 relative error: 1.56e-08
b2 relative error: 7.76e-10
```

9 Solver

Open the file `cs231n/solver.py` and read through it to familiarize yourself with the API. You also need to imeplement the `sgd` function in `cs231n/optim.py`. After doing so, use a `Solver` instance to train a `TwoLayerNet` that achieves about 36% accuracy on the validation set.

[40]:
```
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
model = TwoLayerNet(input_size, hidden_size, num_classes,reg=0.5)
solver = None

################################################################################
# TODO: Use a Solver instance to train a TwoLayerNet that achieves about 36% #
# accuracy on the validation set.                                              #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
solver = Solver(model, data,
                    update_rule='sgd',
                    optim_config={
                      'learning_rate': 5e-3,
                    },
                    lr_decay=0.95,
                    num_epochs=10, batch_size=100,
                    print_every=100)
solver.train()
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
################################################################################
#                              END OF YOUR CODE                                #
################################################################################
```

```
(Iteration 1 / 4900) loss: 2.340973
(Epoch 0 / 10) train acc: 0.179000; val_acc: 0.170000

/content/drive/My Drive/cs231n/assignments/assignment1/cs231n/layers.py:182:
RuntimeWarning: divide by zero encountered in log
  loss = np.sum(-1*np.log(prob[np.arange(num_inputs),y]))
/content/drive/My Drive/cs231n/assignments/assignment1/cs231n/layers.py:179:
RuntimeWarning: overflow encountered in exp
```

```
  x=np.exp(x)
/content/drive/My Drive/cs231n/assignments/assignment1/cs231n/layers.py:181:
RuntimeWarning: invalid value encountered in true_divide
  prob = x / sums

(Iteration 101 / 4900) loss: nan
(Iteration 201 / 4900) loss: nan
(Iteration 301 / 4900) loss: nan
(Iteration 401 / 4900) loss: nan
(Epoch 1 / 10) train acc: 0.102000; val_acc: 0.087000
(Iteration 501 / 4900) loss: nan
(Iteration 601 / 4900) loss: nan
(Iteration 701 / 4900) loss: nan
(Iteration 801 / 4900) loss: nan
(Iteration 901 / 4900) loss: nan
(Epoch 2 / 10) train acc: 0.106000; val_acc: 0.087000
(Iteration 1001 / 4900) loss: nan
(Iteration 1101 / 4900) loss: nan
(Iteration 1201 / 4900) loss: nan
(Iteration 1301 / 4900) loss: nan
(Iteration 1401 / 4900) loss: nan
(Epoch 3 / 10) train acc: 0.103000; val_acc: 0.087000
(Iteration 1501 / 4900) loss: nan
(Iteration 1601 / 4900) loss: nan
(Iteration 1701 / 4900) loss: nan
(Iteration 1801 / 4900) loss: nan
(Iteration 1901 / 4900) loss: nan
(Epoch 4 / 10) train acc: 0.104000; val_acc: 0.087000
(Iteration 2001 / 4900) loss: nan
(Iteration 2101 / 4900) loss: nan
(Iteration 2201 / 4900) loss: nan
(Iteration 2301 / 4900) loss: nan
(Iteration 2401 / 4900) loss: nan
(Epoch 5 / 10) train acc: 0.114000; val_acc: 0.087000
(Iteration 2501 / 4900) loss: nan
(Iteration 2601 / 4900) loss: nan
(Iteration 2701 / 4900) loss: nan
(Iteration 2801 / 4900) loss: nan
(Iteration 2901 / 4900) loss: nan
(Epoch 6 / 10) train acc: 0.102000; val_acc: 0.087000
(Iteration 3001 / 4900) loss: nan
(Iteration 3101 / 4900) loss: nan
(Iteration 3201 / 4900) loss: nan
(Iteration 3301 / 4900) loss: nan
(Iteration 3401 / 4900) loss: nan
(Epoch 7 / 10) train acc: 0.085000; val_acc: 0.087000
(Iteration 3501 / 4900) loss: nan
(Iteration 3601 / 4900) loss: nan
```

```
(Iteration 3701 / 4900) loss: nan
(Iteration 3801 / 4900) loss: nan
(Iteration 3901 / 4900) loss: nan
(Epoch 8 / 10) train acc: 0.086000; val_acc: 0.087000
(Iteration 4001 / 4900) loss: nan
(Iteration 4101 / 4900) loss: nan
(Iteration 4201 / 4900) loss: nan
(Iteration 4301 / 4900) loss: nan
(Iteration 4401 / 4900) loss: nan
(Epoch 9 / 10) train acc: 0.102000; val_acc: 0.087000
(Iteration 4501 / 4900) loss: nan
(Iteration 4601 / 4900) loss: nan
(Iteration 4701 / 4900) loss: nan
(Iteration 4801 / 4900) loss: nan
(Epoch 10 / 10) train acc: 0.097000; val_acc: 0.087000
```

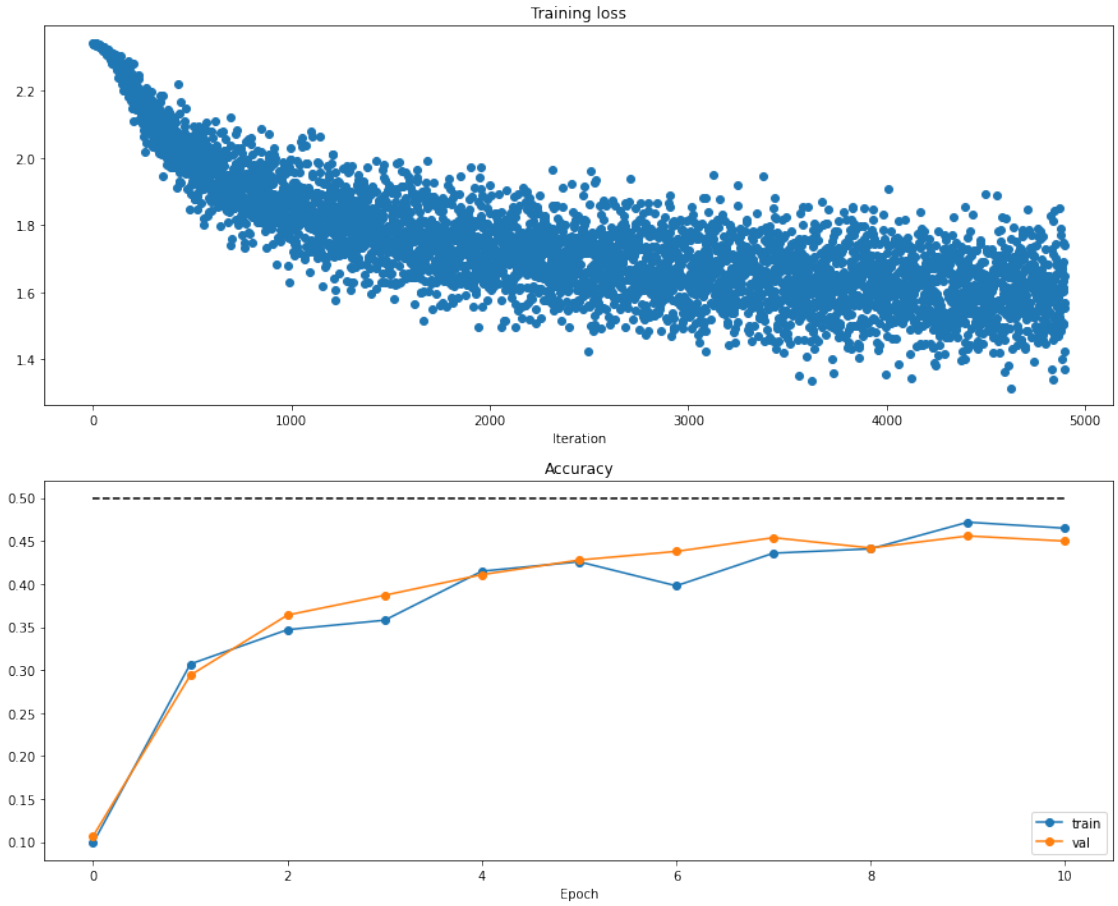
10 Debug the training

With the default parameters we provided above, you should get a validation accuracy of about 0.36 on the validation set. This isn't very good.

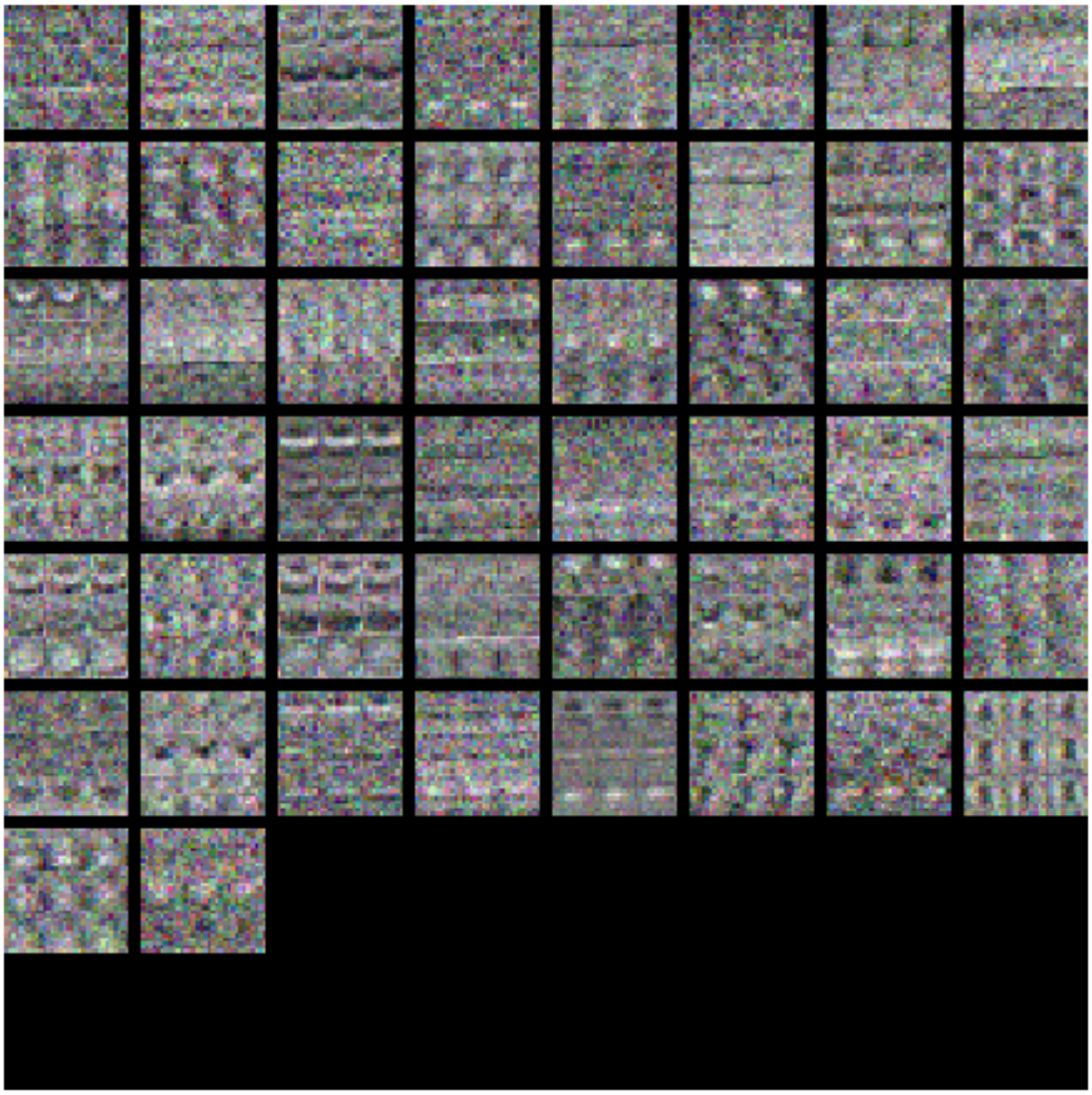
One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

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.

[38]:
```
# Run this cell to visualize training loss and train / val accuracy

plt.subplot(2, 1, 1)
plt.title('Training loss')
plt.plot(solver.loss_history, 'o')
plt.xlabel('Iteration')

plt.subplot(2, 1, 2)
plt.title('Accuracy')
plt.plot(solver.train_acc_history, '-o', label='train')
plt.plot(solver.val_acc_history, '-o', label='val')
plt.plot([0.5] * len(solver.val_acc_history), 'k--')
plt.xlabel('Epoch')
plt.legend(loc='lower right')
plt.gcf().set_size_inches(15, 12)
plt.show()
```



[32]:
```
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(model)
```



11 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 (52% could serve as a reference), 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.).

[42]:
```
best_model = None


################################################################################
# TODO: Tune hyperparameters using the validation set. Store your best trained␣
 ↪#
# model in best_model.                                                         ␣
 ↪#
#                                                                              ␣
 ↪#
# 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 thexs previous exercises.                       ␣
 ↪  #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
best_acc = 0
results = {}
hidden_layer_size = [100]
learning_rates = [1e-3,5e-4,1e-4]
epoches  = [10]
regularizations = [0,0.5]
decays = [1]

for h in hidden_layer_size:
```

```
  for l in learning_rates:
    for e in epoches:
      for r in regularizations:
        for d in decays:
          model = TwoLayerNet(input_size, h, num_classes,reg=r)
          solver = Solver(model, data,
                      update_rule='sgd',
                      optim_config={
                        'learning_rate': l,
                      },
                      lr_decay=d,
                      num_epochs=e, batch_size=100,
                      print_every = 100
                      )
          solver.train()
          val_acc = solver.val_acc_history[-1]
          train_acc = solver.train_acc_history[-1]
          if val_acc > best_acc:
              best_acc  = val_acc
              best_model = model
          results[(h,l,e,r,d)] = (train_acc,val_acc)

for hid, lr, epo, reg, dec in sorted(results):
    train_acc, val_acc = results[(hid,lr,epo,reg,dec)]
    print('hid %d lr %e epo %d reg %e dec %f-> train accuracy: %f val accuracy:␣
 ↪%f' % (
                hid, lr, epo, reg, dec, train_acc, val_acc))
print('best validation accuracy achieved during cross-validation: %f' %␣
 ↪best_acc)
# We could get the best accuracy when hid 300 lr 1e-03 epo 15 reg 0 dec 0.95
# train accuracy: 0.721000 val accuracy: 0.536000
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
################################################################################
#                              END OF YOUR CODE                                ␣
 ↪#
################################################################################
```

```
(Iteration 1 / 4900) loss: 2.301393
(Epoch 0 / 10) train acc: 0.126000; val_acc: 0.120000
(Iteration 101 / 4900) loss: 1.774806
(Iteration 201 / 4900) loss: 1.816545
(Iteration 301 / 4900) loss: 1.666705
(Iteration 401 / 4900) loss: 1.505059
(Epoch 1 / 10) train acc: 0.424000; val_acc: 0.427000
(Iteration 501 / 4900) loss: 1.566510
(Iteration 601 / 4900) loss: 1.561661
(Iteration 701 / 4900) loss: 1.600879
```

```
(Iteration 801 / 4900) loss: 1.293911
(Iteration 901 / 4900) loss: 1.490607
(Epoch 2 / 10) train acc: 0.479000; val_acc: 0.457000
(Iteration 1001 / 4900) loss: 1.512685
(Iteration 1101 / 4900) loss: 1.419139
(Iteration 1201 / 4900) loss: 1.460812
(Iteration 1301 / 4900) loss: 1.281671
(Iteration 1401 / 4900) loss: 1.522814
(Epoch 3 / 10) train acc: 0.515000; val_acc: 0.466000
(Iteration 1501 / 4900) loss: 1.284662
(Iteration 1601 / 4900) loss: 1.374847
(Iteration 1701 / 4900) loss: 1.522880
(Iteration 1801 / 4900) loss: 1.297244
(Iteration 1901 / 4900) loss: 1.247457
(Epoch 4 / 10) train acc: 0.517000; val_acc: 0.488000
(Iteration 2001 / 4900) loss: 1.278034
(Iteration 2101 / 4900) loss: 1.549081
(Iteration 2201 / 4900) loss: 1.396613
(Iteration 2301 / 4900) loss: 1.360628
(Iteration 2401 / 4900) loss: 1.227445
(Epoch 5 / 10) train acc: 0.581000; val_acc: 0.490000
(Iteration 2501 / 4900) loss: 1.124105
(Iteration 2601 / 4900) loss: 1.198423
(Iteration 2701 / 4900) loss: 1.522234
(Iteration 2801 / 4900) loss: 1.154771
(Iteration 2901 / 4900) loss: 1.075773
(Epoch 6 / 10) train acc: 0.566000; val_acc: 0.490000
(Iteration 3001 / 4900) loss: 1.393159
(Iteration 3101 / 4900) loss: 1.278010
(Iteration 3201 / 4900) loss: 1.388739
(Iteration 3301 / 4900) loss: 1.208783
(Iteration 3401 / 4900) loss: 1.469021
(Epoch 7 / 10) train acc: 0.558000; val_acc: 0.491000
(Iteration 3501 / 4900) loss: 1.254009
(Iteration 3601 / 4900) loss: 1.174626
(Iteration 3701 / 4900) loss: 1.208723
(Iteration 3801 / 4900) loss: 1.365438
(Iteration 3901 / 4900) loss: 1.570771
(Epoch 8 / 10) train acc: 0.598000; val_acc: 0.494000
(Iteration 4001 / 4900) loss: 1.107854
(Iteration 4101 / 4900) loss: 1.273925
(Iteration 4201 / 4900) loss: 1.340819
(Iteration 4301 / 4900) loss: 1.246728
(Iteration 4401 / 4900) loss: 1.271102
(Epoch 9 / 10) train acc: 0.597000; val_acc: 0.483000
(Iteration 4501 / 4900) loss: 1.295681
(Iteration 4601 / 4900) loss: 0.986943
(Iteration 4701 / 4900) loss: 1.502703
```

```
(Iteration 4801 / 4900) loss: 1.131034
(Epoch 10 / 10) train acc: 0.571000; val_acc: 0.481000
(Iteration 1 / 4900) loss: 2.378170
(Epoch 0 / 10) train acc: 0.121000; val_acc: 0.112000
(Iteration 101 / 4900) loss: 1.887703
(Iteration 201 / 4900) loss: 1.891420
(Iteration 301 / 4900) loss: 1.811609
(Iteration 401 / 4900) loss: 1.557350
(Epoch 1 / 10) train acc: 0.441000; val_acc: 0.462000
(Iteration 501 / 4900) loss: 1.681168
(Iteration 601 / 4900) loss: 1.616226
(Iteration 701 / 4900) loss: 1.716890
(Iteration 801 / 4900) loss: 1.627443
(Iteration 901 / 4900) loss: 1.541937
(Epoch 2 / 10) train acc: 0.483000; val_acc: 0.452000
(Iteration 1001 / 4900) loss: 1.622673
(Iteration 1101 / 4900) loss: 1.517546
(Iteration 1201 / 4900) loss: 1.541471
(Iteration 1301 / 4900) loss: 1.458646
(Iteration 1401 / 4900) loss: 1.623044
(Epoch 3 / 10) train acc: 0.494000; val_acc: 0.466000
(Iteration 1501 / 4900) loss: 1.694804
(Iteration 1601 / 4900) loss: 1.613119
(Iteration 1701 / 4900) loss: 1.448438
(Iteration 1801 / 4900) loss: 1.380659
(Iteration 1901 / 4900) loss: 1.415924
(Epoch 4 / 10) train acc: 0.535000; val_acc: 0.492000
(Iteration 2001 / 4900) loss: 1.606718
(Iteration 2101 / 4900) loss: 1.609152
(Iteration 2201 / 4900) loss: 1.343624
(Iteration 2301 / 4900) loss: 1.533613
(Iteration 2401 / 4900) loss: 1.505489
(Epoch 5 / 10) train acc: 0.506000; val_acc: 0.484000
(Iteration 2501 / 4900) loss: 1.430438
(Iteration 2601 / 4900) loss: 1.477067
(Iteration 2701 / 4900) loss: 1.533054
(Iteration 2801 / 4900) loss: 1.450823
(Iteration 2901 / 4900) loss: 1.566572
(Epoch 6 / 10) train acc: 0.512000; val_acc: 0.475000
(Iteration 3001 / 4900) loss: 1.348538
(Iteration 3101 / 4900) loss: 1.610322
(Iteration 3201 / 4900) loss: 1.466658
(Iteration 3301 / 4900) loss: 1.613278
(Iteration 3401 / 4900) loss: 1.694517
(Epoch 7 / 10) train acc: 0.550000; val_acc: 0.505000
(Iteration 3501 / 4900) loss: 1.564107
(Iteration 3601 / 4900) loss: 1.625307
(Iteration 3701 / 4900) loss: 1.420901
```

```
(Iteration 3801 / 4900) loss: 1.401897
(Iteration 3901 / 4900) loss: 1.217493
(Epoch 8 / 10) train acc: 0.546000; val_acc: 0.482000
(Iteration 4001 / 4900) loss: 1.594369
(Iteration 4101 / 4900) loss: 1.511918
(Iteration 4201 / 4900) loss: 1.438712
(Iteration 4301 / 4900) loss: 1.537476
(Iteration 4401 / 4900) loss: 1.405801
(Epoch 9 / 10) train acc: 0.546000; val_acc: 0.484000
(Iteration 4501 / 4900) loss: 1.185268
(Iteration 4601 / 4900) loss: 1.683230
(Iteration 4701 / 4900) loss: 1.585832
(Iteration 4801 / 4900) loss: 1.621432
(Epoch 10 / 10) train acc: 0.544000; val_acc: 0.481000
(Iteration 1 / 4900) loss: 2.299127
(Epoch 0 / 10) train acc: 0.133000; val_acc: 0.104000
(Iteration 101 / 4900) loss: 1.902231
(Iteration 201 / 4900) loss: 1.766778
(Iteration 301 / 4900) loss: 1.604147
(Iteration 401 / 4900) loss: 1.614204
(Epoch 1 / 10) train acc: 0.422000; val_acc: 0.446000
(Iteration 501 / 4900) loss: 1.504474
(Iteration 601 / 4900) loss: 1.648727
(Iteration 701 / 4900) loss: 1.446590
(Iteration 801 / 4900) loss: 1.593642
(Iteration 901 / 4900) loss: 1.596574
(Epoch 2 / 10) train acc: 0.471000; val_acc: 0.455000
(Iteration 1001 / 4900) loss: 1.532691
(Iteration 1101 / 4900) loss: 1.397616
(Iteration 1201 / 4900) loss: 1.410562
(Iteration 1301 / 4900) loss: 1.591488
(Iteration 1401 / 4900) loss: 1.344852
(Epoch 3 / 10) train acc: 0.509000; val_acc: 0.458000
(Iteration 1501 / 4900) loss: 1.389681
(Iteration 1601 / 4900) loss: 1.404832
(Iteration 1701 / 4900) loss: 1.456351
(Iteration 1801 / 4900) loss: 1.319741
(Iteration 1901 / 4900) loss: 1.305875
(Epoch 4 / 10) train acc: 0.508000; val_acc: 0.494000
(Iteration 2001 / 4900) loss: 1.221805
(Iteration 2101 / 4900) loss: 1.571302
(Iteration 2201 / 4900) loss: 1.310668
(Iteration 2301 / 4900) loss: 1.394919
(Iteration 2401 / 4900) loss: 1.281760
(Epoch 5 / 10) train acc: 0.537000; val_acc: 0.501000
(Iteration 2501 / 4900) loss: 1.322952
(Iteration 2601 / 4900) loss: 1.147023
(Iteration 2701 / 4900) loss: 1.179337
```

```
(Iteration 2801 / 4900) loss: 1.387855
(Iteration 2901 / 4900) loss: 1.233376
(Epoch 6 / 10) train acc: 0.545000; val_acc: 0.514000
(Iteration 3001 / 4900) loss: 1.156703
(Iteration 3101 / 4900) loss: 1.118257
(Iteration 3201 / 4900) loss: 1.145619
(Iteration 3301 / 4900) loss: 1.191590
(Iteration 3401 / 4900) loss: 1.181890
(Epoch 7 / 10) train acc: 0.575000; val_acc: 0.515000
(Iteration 3501 / 4900) loss: 1.231354
(Iteration 3601 / 4900) loss: 1.125690
(Iteration 3701 / 4900) loss: 1.161948
(Iteration 3801 / 4900) loss: 1.344553
(Iteration 3901 / 4900) loss: 1.307941
(Epoch 8 / 10) train acc: 0.607000; val_acc: 0.517000
(Iteration 4001 / 4900) loss: 1.081756
(Iteration 4101 / 4900) loss: 1.208399
(Iteration 4201 / 4900) loss: 1.282724
(Iteration 4301 / 4900) loss: 1.321047
(Iteration 4401 / 4900) loss: 1.004661
(Epoch 9 / 10) train acc: 0.574000; val_acc: 0.510000
(Iteration 4501 / 4900) loss: 1.233069
(Iteration 4601 / 4900) loss: 1.428188
(Iteration 4701 / 4900) loss: 1.173545
(Iteration 4801 / 4900) loss: 1.031971
(Epoch 10 / 10) train acc: 0.582000; val_acc: 0.510000
(Iteration 1 / 4900) loss: 2.381009
(Epoch 0 / 10) train acc: 0.117000; val_acc: 0.118000
(Iteration 101 / 4900) loss: 1.960730
(Iteration 201 / 4900) loss: 1.809302
(Iteration 301 / 4900) loss: 1.754653
(Iteration 401 / 4900) loss: 1.931911
(Epoch 1 / 10) train acc: 0.443000; val_acc: 0.438000
(Iteration 501 / 4900) loss: 1.637047
(Iteration 601 / 4900) loss: 1.633475
(Iteration 701 / 4900) loss: 1.729548
(Iteration 801 / 4900) loss: 1.587773
(Iteration 901 / 4900) loss: 1.511011
(Epoch 2 / 10) train acc: 0.497000; val_acc: 0.439000
(Iteration 1001 / 4900) loss: 1.452511
(Iteration 1101 / 4900) loss: 1.650966
(Iteration 1201 / 4900) loss: 1.575161
(Iteration 1301 / 4900) loss: 1.732520
(Iteration 1401 / 4900) loss: 1.590495
(Epoch 3 / 10) train acc: 0.506000; val_acc: 0.503000
(Iteration 1501 / 4900) loss: 1.471875
(Iteration 1601 / 4900) loss: 1.451665
(Iteration 1701 / 4900) loss: 1.583858
```

```
(Iteration 1801 / 4900) loss: 1.436226
(Iteration 1901 / 4900) loss: 1.706887
(Epoch 4 / 10) train acc: 0.500000; val_acc: 0.485000
(Iteration 2001 / 4900) loss: 1.372047
(Iteration 2101 / 4900) loss: 1.479143
(Iteration 2201 / 4900) loss: 1.703727
(Iteration 2301 / 4900) loss: 1.255319
(Iteration 2401 / 4900) loss: 1.524721
(Epoch 5 / 10) train acc: 0.508000; val_acc: 0.474000
(Iteration 2501 / 4900) loss: 1.470775
(Iteration 2601 / 4900) loss: 1.601060
(Iteration 2701 / 4900) loss: 1.545293
(Iteration 2801 / 4900) loss: 1.449020
(Iteration 2901 / 4900) loss: 1.523087
(Epoch 6 / 10) train acc: 0.533000; val_acc: 0.499000
(Iteration 3001 / 4900) loss: 1.492550
(Iteration 3101 / 4900) loss: 1.419208
(Iteration 3201 / 4900) loss: 1.451660
(Iteration 3301 / 4900) loss: 1.318635
(Iteration 3401 / 4900) loss: 1.471655
(Epoch 7 / 10) train acc: 0.580000; val_acc: 0.521000
(Iteration 3501 / 4900) loss: 1.659391
(Iteration 3601 / 4900) loss: 1.210414
(Iteration 3701 / 4900) loss: 1.479249
(Iteration 3801 / 4900) loss: 1.402751
(Iteration 3901 / 4900) loss: 1.531499
(Epoch 8 / 10) train acc: 0.566000; val_acc: 0.512000
(Iteration 4001 / 4900) loss: 1.291477
(Iteration 4101 / 4900) loss: 1.350196
(Iteration 4201 / 4900) loss: 1.312674
(Iteration 4301 / 4900) loss: 1.430068
(Iteration 4401 / 4900) loss: 1.445067
(Epoch 9 / 10) train acc: 0.562000; val_acc: 0.518000
(Iteration 4501 / 4900) loss: 1.473811
(Iteration 4601 / 4900) loss: 1.380379
(Iteration 4701 / 4900) loss: 1.308887
(Iteration 4801 / 4900) loss: 1.286157
(Epoch 10 / 10) train acc: 0.565000; val_acc: 0.504000
(Iteration 1 / 4900) loss: 2.297180
(Epoch 0 / 10) train acc: 0.093000; val_acc: 0.113000
(Iteration 101 / 4900) loss: 2.204038
(Iteration 201 / 4900) loss: 2.152023
(Iteration 301 / 4900) loss: 2.021991
(Iteration 401 / 4900) loss: 1.962583
(Epoch 1 / 10) train acc: 0.328000; val_acc: 0.325000
(Iteration 501 / 4900) loss: 1.950134
(Iteration 601 / 4900) loss: 1.954188
(Iteration 701 / 4900) loss: 1.734749
```

```
(Iteration 801 / 4900) loss: 1.990557
(Iteration 901 / 4900) loss: 1.810274
(Epoch 2 / 10) train acc: 0.397000; val_acc: 0.376000
(Iteration 1001 / 4900) loss: 1.822548
(Iteration 1101 / 4900) loss: 1.719478
(Iteration 1201 / 4900) loss: 1.812156
(Iteration 1301 / 4900) loss: 1.870070
(Iteration 1401 / 4900) loss: 1.671924
(Epoch 3 / 10) train acc: 0.415000; val_acc: 0.410000
(Iteration 1501 / 4900) loss: 1.520255
(Iteration 1601 / 4900) loss: 1.484642
(Iteration 1701 / 4900) loss: 1.605882
(Iteration 1801 / 4900) loss: 1.657070
(Iteration 1901 / 4900) loss: 1.665793
(Epoch 4 / 10) train acc: 0.432000; val_acc: 0.429000
(Iteration 2001 / 4900) loss: 1.547300
(Iteration 2101 / 4900) loss: 1.542364
(Iteration 2201 / 4900) loss: 1.574748
(Iteration 2301 / 4900) loss: 1.587733
(Iteration 2401 / 4900) loss: 1.446562
(Epoch 5 / 10) train acc: 0.459000; val_acc: 0.450000
(Iteration 2501 / 4900) loss: 1.491172
(Iteration 2601 / 4900) loss: 1.687303
(Iteration 2701 / 4900) loss: 1.470343
(Iteration 2801 / 4900) loss: 1.514824
(Iteration 2901 / 4900) loss: 1.470769
(Epoch 6 / 10) train acc: 0.432000; val_acc: 0.453000
(Iteration 3001 / 4900) loss: 1.554536
(Iteration 3101 / 4900) loss: 1.370204
(Iteration 3201 / 4900) loss: 1.355612
(Iteration 3301 / 4900) loss: 1.516678
(Iteration 3401 / 4900) loss: 1.456140
(Epoch 7 / 10) train acc: 0.463000; val_acc: 0.460000
(Iteration 3501 / 4900) loss: 1.747901
(Iteration 3601 / 4900) loss: 1.388408
(Iteration 3701 / 4900) loss: 1.404288
(Iteration 3801 / 4900) loss: 1.399779
(Iteration 3901 / 4900) loss: 1.511168
(Epoch 8 / 10) train acc: 0.506000; val_acc: 0.470000
(Iteration 4001 / 4900) loss: 1.502276
(Iteration 4101 / 4900) loss: 1.439277
(Iteration 4201 / 4900) loss: 1.613766
(Iteration 4301 / 4900) loss: 1.305960
(Iteration 4401 / 4900) loss: 1.410102
(Epoch 9 / 10) train acc: 0.484000; val_acc: 0.468000
(Iteration 4501 / 4900) loss: 1.472160
(Iteration 4601 / 4900) loss: 1.363685
(Iteration 4701 / 4900) loss: 1.448979
```

```
(Iteration 4801 / 4900) loss: 1.500675
(Epoch 10 / 10) train acc: 0.498000; val_acc: 0.475000
(Iteration 1 / 4900) loss: 2.381522
(Epoch 0 / 10) train acc: 0.104000; val_acc: 0.086000
(Iteration 101 / 4900) loss: 2.307122
(Iteration 201 / 4900) loss: 2.234808
(Iteration 301 / 4900) loss: 2.128253
(Iteration 401 / 4900) loss: 2.050948
(Epoch 1 / 10) train acc: 0.309000; val_acc: 0.314000
(Iteration 501 / 4900) loss: 1.980760
(Iteration 601 / 4900) loss: 2.013353
(Iteration 701 / 4900) loss: 1.879666
(Iteration 801 / 4900) loss: 1.861027
(Iteration 901 / 4900) loss: 1.982732
(Epoch 2 / 10) train acc: 0.393000; val_acc: 0.373000
(Iteration 1001 / 4900) loss: 1.726480
(Iteration 1101 / 4900) loss: 1.883766
(Iteration 1201 / 4900) loss: 1.766789
(Iteration 1301 / 4900) loss: 1.968353
(Iteration 1401 / 4900) loss: 1.722259
(Epoch 3 / 10) train acc: 0.409000; val_acc: 0.398000
(Iteration 1501 / 4900) loss: 1.730825
(Iteration 1601 / 4900) loss: 1.761788
(Iteration 1701 / 4900) loss: 1.741395
(Iteration 1801 / 4900) loss: 1.774275
(Iteration 1901 / 4900) loss: 1.629283
(Epoch 4 / 10) train acc: 0.436000; val_acc: 0.429000
(Iteration 2001 / 4900) loss: 1.603344
(Iteration 2101 / 4900) loss: 1.742617
(Iteration 2201 / 4900) loss: 1.746428
(Iteration 2301 / 4900) loss: 1.597399
(Iteration 2401 / 4900) loss: 1.637922
(Epoch 5 / 10) train acc: 0.453000; val_acc: 0.446000
(Iteration 2501 / 4900) loss: 1.660396
(Iteration 2601 / 4900) loss: 1.614308
(Iteration 2701 / 4900) loss: 1.645025
(Iteration 2801 / 4900) loss: 1.659117
(Iteration 2901 / 4900) loss: 1.642967
(Epoch 6 / 10) train acc: 0.476000; val_acc: 0.456000
(Iteration 3001 / 4900) loss: 1.544232
(Iteration 3101 / 4900) loss: 1.603190
(Iteration 3201 / 4900) loss: 1.547277
(Iteration 3301 / 4900) loss: 1.575440
(Iteration 3401 / 4900) loss: 1.679713
(Epoch 7 / 10) train acc: 0.470000; val_acc: 0.467000
(Iteration 3501 / 4900) loss: 1.690147
(Iteration 3601 / 4900) loss: 1.504785
(Iteration 3701 / 4900) loss: 1.709383
```

```
(Iteration 3801 / 4900) loss: 1.698078
(Iteration 3901 / 4900) loss: 1.722319
(Epoch 8 / 10) train acc: 0.455000; val_acc: 0.461000
(Iteration 4001 / 4900) loss: 1.665688
(Iteration 4101 / 4900) loss: 1.582188
(Iteration 4201 / 4900) loss: 1.535284
(Iteration 4301 / 4900) loss: 1.445238
(Iteration 4401 / 4900) loss: 1.598317
(Epoch 9 / 10) train acc: 0.484000; val_acc: 0.468000
(Iteration 4501 / 4900) loss: 1.699275
(Iteration 4601 / 4900) loss: 1.638127
(Iteration 4701 / 4900) loss: 1.620317
(Iteration 4801 / 4900) loss: 1.432336
(Epoch 10 / 10) train acc: 0.502000; val_acc: 0.467000
hid 100 lr 1.000000e-04 epo 10 reg 0.000000e+00 dec 1.000000-> train accuracy:
0.498000 val accuracy: 0.475000
hid 100 lr 1.000000e-04 epo 10 reg 5.000000e-01 dec 1.000000-> train accuracy:
0.502000 val accuracy: 0.467000
hid 100 lr 5.000000e-04 epo 10 reg 0.000000e+00 dec 1.000000-> train accuracy:
0.582000 val accuracy: 0.510000
hid 100 lr 5.000000e-04 epo 10 reg 5.000000e-01 dec 1.000000-> train accuracy:
0.565000 val accuracy: 0.504000
hid 100 lr 1.000000e-03 epo 10 reg 0.000000e+00 dec 1.000000-> train accuracy:
0.571000 val accuracy: 0.481000
hid 100 lr 1.000000e-03 epo 10 reg 5.000000e-01 dec 1.000000-> train accuracy:
0.544000 val accuracy: 0.481000
best validation accuracy achieved during cross-validation: 0.510000
```

12 Test your model!

Run your best model on the validation and test sets. You should achieve above 48% accuracy on the validation set and the test set.

[ ]:
```
y_val_pred = np.argmax(best_model.loss(data['X_val']), axis=1)
print('Validation set accuracy: ', (y_val_pred == data['y_val']).mean())
```

```
Validation set accuracy:  0.557
```

[ ]:
```
y_test_pred = np.argmax(best_model.loss(data['X_test']), axis=1)
print('Test set accuracy: ', (y_test_pred == data['y_test']).mean())
```

```
Test set accuracy:  0.524
```

12.1 Inline Question 2:

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.

YourAnswer : 1,3
YourExplanation :
The gap between testing accuracy and training accuracy means our model makes overfitting. To reduce overfitting, we have to train more datas and perform regularization. But if we already done regularization, we can increase its strength. It helps our classifier more simple and flexible.

features

August 30, 2021

[1]:
```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

# Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment1/'
FOLDERNAME = 'cs231n/assignments/assignment1/'
assert FOLDERNAME is not None, "[!] Enter the foldername."

# Now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# This downloads the CIFAR-10 dataset to your Drive
# if it doesn't already exist.
%cd drive/My\ Drive/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/cs231n/assignments/assignment1/cs231n/datasets
/content/drive/My Drive/cs231n/assignments/assignment1
```

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.

[2]:
```
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt


%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading extenrnal modules
# see http://stackoverflow.com/questions/1907993/
 ↪autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

1.1 Load data

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

[3]:
```
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'

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

    X_train, y_train, X_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

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 own interest.

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.

[4]:
```
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
Done extracting features for 49000 / 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.

[5]:
```
# Use the validation set to tune the learning rate and regularization strength

from cs231n.classifiers.linear_classifier import LinearSVM

learning_rates = [8.5e-9, 9e-9, 9.5e-9, 1e-8]
regularization_strengths = [3.5e5, 4e5, 4.5e5, 5e5, 5.5e5]

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.       ␣
 ↪#
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
for l in learning_rates:
    for r in regularization_strengths:
        svm = LinearSVM()
        svm.train(X_train_feats, y_train, l, r,
                      num_iters=1500, verbose=True)
        train_ac = np.mean(y_train == svm.predict(X_train_feats))
        val_ac = np.mean(y_val == svm.predict(X_val_feats))
        if val_ac > best_val:
          best_val, best_svm = val_ac, svm
        results[(l,r)] = (train_ac ,val_ac)
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

# 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: %f' % best_val)
```

```
iteration 0 / 1500: loss 542.738659
iteration 100 / 1500: loss 170.800441
iteration 200 / 1500: loss 58.048285
iteration 300 / 1500: loss 23.865857
iteration 400 / 1500: loss 13.507107
iteration 500 / 1500: loss 10.365869
iteration 600 / 1500: loss 9.413846
iteration 700 / 1500: loss 9.125645
iteration 800 / 1500: loss 9.037969
iteration 900 / 1500: loss 9.011479
iteration 1000 / 1500: loss 9.003448
iteration 1100 / 1500: loss 9.001002
iteration 1200 / 1500: loss 9.000270
iteration 1300 / 1500: loss 9.000060
iteration 1400 / 1500: loss 8.999969
iteration 0 / 1500: loss 635.722110
iteration 100 / 1500: loss 169.109113
iteration 200 / 1500: loss 49.906112
iteration 300 / 1500: loss 19.449341
iteration 400 / 1500: loss 11.668146
iteration 500 / 1500: loss 9.682428
iteration 600 / 1500: loss 9.174142
iteration 700 / 1500: loss 9.044477
iteration 800 / 1500: loss 9.011369
iteration 900 / 1500: loss 9.002814
iteration 1000 / 1500: loss 9.000697
iteration 1100 / 1500: loss 9.000148
iteration 1200 / 1500: loss 9.000004
iteration 1300 / 1500: loss 8.999970
iteration 1400 / 1500: loss 8.999961
iteration 0 / 1500: loss 714.439303
iteration 100 / 1500: loss 160.859928
iteration 200 / 1500: loss 41.690168
iteration 300 / 1500: loss 16.036677
iteration 400 / 1500: loss 10.514832
iteration 500 / 1500: loss 9.325955
iteration 600 / 1500: loss 9.070256
iteration 700 / 1500: loss 9.015095
iteration 800 / 1500: loss 9.003208
iteration 900 / 1500: loss 9.000657
iteration 1000 / 1500: loss 9.000112
iteration 1100 / 1500: loss 9.000003
iteration 1200 / 1500: loss 8.999965
```

```
iteration 1300 / 1500: loss 8.999960
iteration 1400 / 1500: loss 8.999962
iteration 0 / 1500: loss 746.315810
iteration 100 / 1500: loss 142.722830
iteration 200 / 1500: loss 33.250953
iteration 300 / 1500: loss 13.398176
iteration 400 / 1500: loss 9.797893
iteration 500 / 1500: loss 9.144551
iteration 600 / 1500: loss 9.026259
iteration 700 / 1500: loss 9.004718
iteration 800 / 1500: loss 9.000832
iteration 900 / 1500: loss 9.000119
iteration 1000 / 1500: loss 8.999987
iteration 1100 / 1500: loss 8.999974
iteration 1200 / 1500: loss 8.999969
iteration 1300 / 1500: loss 8.999959
iteration 1400 / 1500: loss 8.999967
iteration 0 / 1500: loss 837.929848
iteration 100 / 1500: loss 135.641371
iteration 200 / 1500: loss 28.348578
iteration 300 / 1500: loss 11.955808
iteration 400 / 1500: loss 9.451340
iteration 500 / 1500: loss 9.068986
iteration 600 / 1500: loss 9.010500
iteration 700 / 1500: loss 9.001582
iteration 800 / 1500: loss 9.000222
iteration 900 / 1500: loss 9.000003
iteration 1000 / 1500: loss 8.999971
iteration 1100 / 1500: loss 8.999969
iteration 1200 / 1500: loss 8.999971
iteration 1300 / 1500: loss 8.999971
iteration 1400 / 1500: loss 8.999967
iteration 0 / 1500: loss 557.401099
iteration 100 / 1500: loss 163.940167
iteration 200 / 1500: loss 52.774281
iteration 300 / 1500: loss 21.366963
iteration 400 / 1500: loss 12.493510
iteration 500 / 1500: loss 9.987369
iteration 600 / 1500: loss 9.278581
iteration 700 / 1500: loss 9.078651
iteration 800 / 1500: loss 9.022237
iteration 900 / 1500: loss 9.006232
iteration 1000 / 1500: loss 9.001704
iteration 1100 / 1500: loss 9.000456
iteration 1200 / 1500: loss 9.000094
iteration 1300 / 1500: loss 8.999999
iteration 1400 / 1500: loss 8.999958
iteration 0 / 1500: loss 601.810844
```

```
iteration 100 / 1500: loss 148.724033
iteration 200 / 1500: loss 41.931186
iteration 300 / 1500: loss 16.760915
iteration 400 / 1500: loss 10.829604
iteration 500 / 1500: loss 9.430997
iteration 600 / 1500: loss 9.101475
iteration 700 / 1500: loss 9.023930
iteration 800 / 1500: loss 9.005629
iteration 900 / 1500: loss 9.001287
iteration 1000 / 1500: loss 9.000260
iteration 1100 / 1500: loss 9.000023
iteration 1200 / 1500: loss 8.999980
iteration 1300 / 1500: loss 8.999962
iteration 1400 / 1500: loss 8.999961
iteration 0 / 1500: loss 738.929361
iteration 100 / 1500: loss 152.505632
iteration 200 / 1500: loss 37.211758
iteration 300 / 1500: loss 14.546623
iteration 400 / 1500: loss 10.090469
iteration 500 / 1500: loss 9.214342
iteration 600 / 1500: loss 9.042088
iteration 700 / 1500: loss 9.008237
iteration 800 / 1500: loss 9.001578
iteration 900 / 1500: loss 9.000286
iteration 1000 / 1500: loss 9.000015
iteration 1100 / 1500: loss 8.999969
iteration 1200 / 1500: loss 8.999962
iteration 1300 / 1500: loss 8.999963
iteration 1400 / 1500: loss 8.999950
iteration 0 / 1500: loss 843.858317
iteration 100 / 1500: loss 145.879930
iteration 200 / 1500: loss 31.442588
iteration 300 / 1500: loss 12.679069
iteration 400 / 1500: loss 9.603505
iteration 500 / 1500: loss 9.098930
iteration 600 / 1500: loss 9.016201
iteration 700 / 1500: loss 9.002622
iteration 800 / 1500: loss 9.000401
iteration 900 / 1500: loss 9.000035
iteration 1000 / 1500: loss 8.999969
iteration 1100 / 1500: loss 8.999982
iteration 1200 / 1500: loss 8.999965
iteration 1300 / 1500: loss 8.999967
iteration 1400 / 1500: loss 8.999964
iteration 0 / 1500: loss 841.408103
iteration 100 / 1500: loss 122.802392
iteration 200 / 1500: loss 24.557622
iteration 300 / 1500: loss 11.127320
```

```
iteration 400 / 1500: loss 9.290831
iteration 500 / 1500: loss 9.039753
iteration 600 / 1500: loss 9.005451
iteration 700 / 1500: loss 9.000705
iteration 800 / 1500: loss 9.000079
iteration 900 / 1500: loss 8.999980
iteration 1000 / 1500: loss 8.999968
iteration 1100 / 1500: loss 8.999968
iteration 1200 / 1500: loss 8.999967
iteration 1300 / 1500: loss 8.999963
iteration 1400 / 1500: loss 8.999963
iteration 0 / 1500: loss 564.508056
iteration 100 / 1500: loss 155.262693
iteration 200 / 1500: loss 47.511821
iteration 300 / 1500: loss 19.140199
iteration 400 / 1500: loss 11.669964
iteration 500 / 1500: loss 9.702763
iteration 600 / 1500: loss 9.184794
iteration 700 / 1500: loss 9.048718
iteration 800 / 1500: loss 9.012803
iteration 900 / 1500: loss 9.003302
iteration 1000 / 1500: loss 9.000856
iteration 1100 / 1500: loss 9.000182
iteration 1200 / 1500: loss 9.000026
iteration 1300 / 1500: loss 8.999966
iteration 1400 / 1500: loss 8.999955
iteration 0 / 1500: loss 635.619315
iteration 100 / 1500: loss 145.259507
iteration 200 / 1500: loss 38.626679
iteration 300 / 1500: loss 15.443923
iteration 400 / 1500: loss 10.401232
iteration 500 / 1500: loss 9.304626
iteration 600 / 1500: loss 9.066200
iteration 700 / 1500: loss 9.014322
iteration 800 / 1500: loss 9.003098
iteration 900 / 1500: loss 9.000638
iteration 1000 / 1500: loss 9.000098
iteration 1100 / 1500: loss 8.999995
iteration 1200 / 1500: loss 8.999957
iteration 1300 / 1500: loss 8.999957
iteration 1400 / 1500: loss 8.999951
iteration 0 / 1500: loss 671.264594
iteration 100 / 1500: loss 127.906623
iteration 200 / 1500: loss 30.347116
iteration 300 / 1500: loss 12.832244
iteration 400 / 1500: loss 9.688061
iteration 500 / 1500: loss 9.123602
iteration 600 / 1500: loss 9.022183
```

```
iteration 700 / 1500: loss 9.003992
iteration 800 / 1500: loss 9.000688
iteration 900 / 1500: loss 9.000074
iteration 1000 / 1500: loss 8.999990
iteration 1100 / 1500: loss 8.999968
iteration 1200 / 1500: loss 8.999967
iteration 1300 / 1500: loss 8.999957
iteration 1400 / 1500: loss 8.999961
iteration 0 / 1500: loss 817.804213
iteration 100 / 1500: loss 128.877363
iteration 200 / 1500: loss 26.767567
iteration 300 / 1500: loss 11.633959
iteration 400 / 1500: loss 9.390301
iteration 500 / 1500: loss 9.057832
iteration 600 / 1500: loss 9.008545
iteration 700 / 1500: loss 9.001228
iteration 800 / 1500: loss 9.000136
iteration 900 / 1500: loss 8.999993
iteration 1000 / 1500: loss 8.999975
iteration 1100 / 1500: loss 8.999963
iteration 1200 / 1500: loss 8.999962
iteration 1300 / 1500: loss 8.999961
iteration 1400 / 1500: loss 8.999969
iteration 0 / 1500: loss 933.827608
iteration 100 / 1500: loss 122.139293
iteration 200 / 1500: loss 22.840670
iteration 300 / 1500: loss 10.693646
iteration 400 / 1500: loss 9.206947
iteration 500 / 1500: loss 9.025337
iteration 600 / 1500: loss 9.003049
iteration 700 / 1500: loss 9.000349
iteration 800 / 1500: loss 9.000015
iteration 900 / 1500: loss 8.999972
iteration 1000 / 1500: loss 8.999973
iteration 1100 / 1500: loss 8.999971
iteration 1200 / 1500: loss 8.999968
iteration 1300 / 1500: loss 8.999962
iteration 1400 / 1500: loss 8.999972
iteration 0 / 1500: loss 552.182348
iteration 100 / 1500: loss 142.289807
iteration 200 / 1500: loss 41.704332
iteration 300 / 1500: loss 17.026998
iteration 400 / 1500: loss 10.969814
iteration 500 / 1500: loss 9.483001
iteration 600 / 1500: loss 9.118740
iteration 700 / 1500: loss 9.029148
iteration 800 / 1500: loss 9.007051
iteration 900 / 1500: loss 9.001684
```

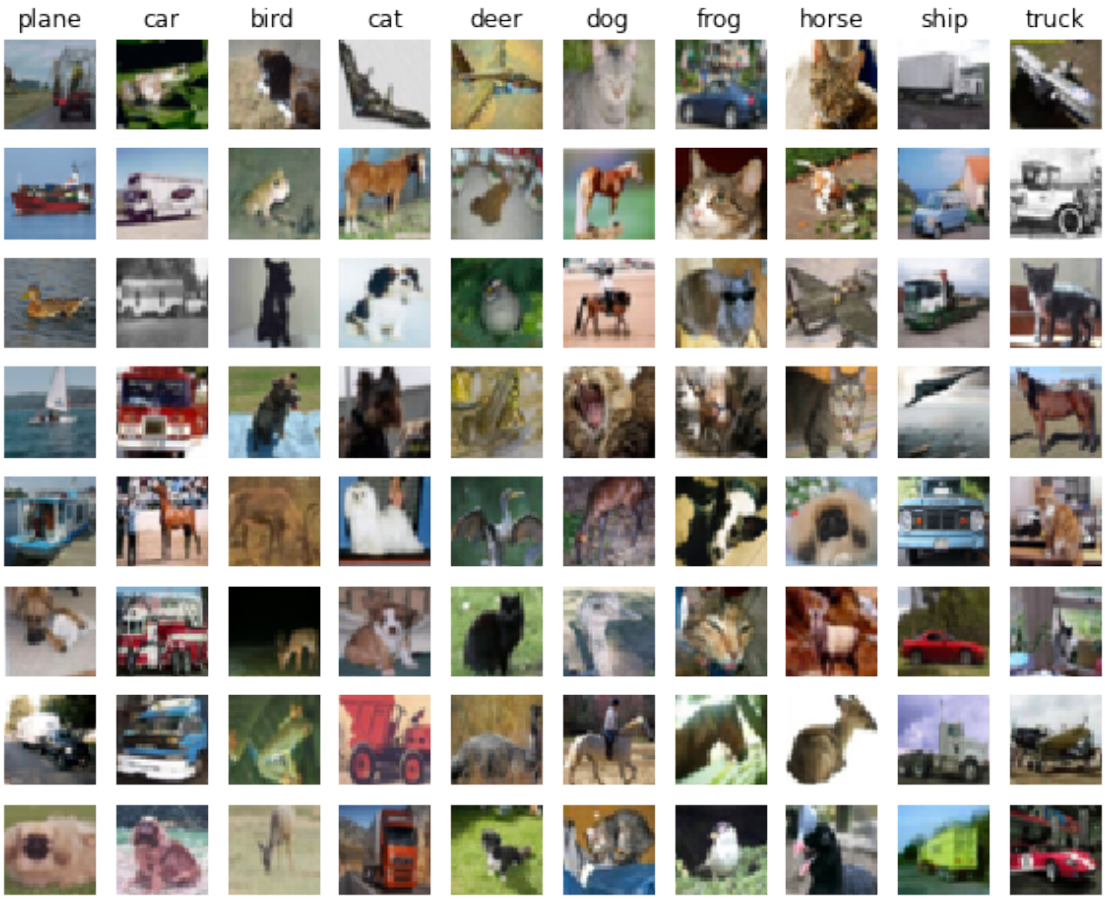
```
iteration 1000 / 1500: loss 9.000382
iteration 1100 / 1500: loss 9.000055
iteration 1200 / 1500: loss 8.999964
iteration 1300 / 1500: loss 8.999952
iteration 1400 / 1500: loss 8.999956
iteration 0 / 1500: loss 603.833061
iteration 100 / 1500: loss 128.324098
iteration 200 / 1500: loss 32.938282
iteration 300 / 1500: loss 13.801876
iteration 400 / 1500: loss 9.962948
iteration 500 / 1500: loss 9.193282
iteration 600 / 1500: loss 9.038705
iteration 700 / 1500: loss 9.007714
iteration 800 / 1500: loss 9.001516
iteration 900 / 1500: loss 9.000266
iteration 1000 / 1500: loss 9.000024
iteration 1100 / 1500: loss 8.999971
iteration 1200 / 1500: loss 8.999966
iteration 1300 / 1500: loss 8.999960
iteration 1400 / 1500: loss 8.999960
iteration 0 / 1500: loss 682.462915
iteration 100 / 1500: loss 119.422453
iteration 200 / 1500: loss 27.103620
iteration 300 / 1500: loss 11.968019
iteration 400 / 1500: loss 9.486636
iteration 500 / 1500: loss 9.079689
iteration 600 / 1500: loss 9.013057
iteration 700 / 1500: loss 9.002089
iteration 800 / 1500: loss 9.000321
iteration 900 / 1500: loss 9.000031
iteration 1000 / 1500: loss 8.999971
iteration 1100 / 1500: loss 8.999961
iteration 1200 / 1500: loss 8.999963
iteration 1300 / 1500: loss 8.999956
iteration 1400 / 1500: loss 8.999946
iteration 0 / 1500: loss 849.094113
iteration 100 / 1500: loss 121.560210
iteration 200 / 1500: loss 24.082080
iteration 300 / 1500: loss 11.020582
iteration 400 / 1500: loss 9.270780
iteration 500 / 1500: loss 9.036204
iteration 600 / 1500: loss 9.004838
iteration 700 / 1500: loss 9.000623
iteration 800 / 1500: loss 9.000047
iteration 900 / 1500: loss 8.999982
iteration 1000 / 1500: loss 8.999969
iteration 1100 / 1500: loss 8.999971
iteration 1200 / 1500: loss 8.999960
```

```
iteration 1300 / 1500: loss 8.999967
iteration 1400 / 1500: loss 8.999960
iteration 0 / 1500: loss 790.924986
iteration 100 / 1500: loss 94.589658
iteration 200 / 1500: loss 18.368760
iteration 300 / 1500: loss 10.025325
iteration 400 / 1500: loss 9.112101
iteration 500 / 1500: loss 9.012295
iteration 600 / 1500: loss 9.001299
iteration 700 / 1500: loss 9.000119
iteration 800 / 1500: loss 8.999980
iteration 900 / 1500: loss 8.999963
iteration 1000 / 1500: loss 8.999963
iteration 1100 / 1500: loss 8.999966
iteration 1200 / 1500: loss 8.999965
iteration 1300 / 1500: loss 8.999971
iteration 1400 / 1500: loss 8.999972
lr 8.500000e-09 reg 3.500000e+05 train accuracy: 0.405694 val accuracy: 0.410000
lr 8.500000e-09 reg 4.000000e+05 train accuracy: 0.413245 val accuracy: 0.423000
lr 8.500000e-09 reg 4.500000e+05 train accuracy: 0.414857 val accuracy: 0.410000
lr 8.500000e-09 reg 5.000000e+05 train accuracy: 0.414918 val accuracy: 0.421000
lr 8.500000e-09 reg 5.500000e+05 train accuracy: 0.411469 val accuracy: 0.407000
lr 9.000000e-09 reg 3.500000e+05 train accuracy: 0.414490 val accuracy: 0.422000
lr 9.000000e-09 reg 4.000000e+05 train accuracy: 0.417000 val accuracy: 0.410000
lr 9.000000e-09 reg 4.500000e+05 train accuracy: 0.410735 val accuracy: 0.408000
lr 9.000000e-09 reg 5.000000e+05 train accuracy: 0.414265 val accuracy: 0.415000
lr 9.000000e-09 reg 5.500000e+05 train accuracy: 0.412449 val accuracy: 0.416000
lr 9.500000e-09 reg 3.500000e+05 train accuracy: 0.417163 val accuracy: 0.411000
lr 9.500000e-09 reg 4.000000e+05 train accuracy: 0.415327 val accuracy: 0.416000
lr 9.500000e-09 reg 4.500000e+05 train accuracy: 0.415000 val accuracy: 0.416000
lr 9.500000e-09 reg 5.000000e+05 train accuracy: 0.415041 val accuracy: 0.413000
lr 9.500000e-09 reg 5.500000e+05 train accuracy: 0.416510 val accuracy: 0.417000
lr 1.000000e-08 reg 3.500000e+05 train accuracy: 0.415102 val accuracy: 0.418000
lr 1.000000e-08 reg 4.000000e+05 train accuracy: 0.418082 val accuracy: 0.421000
lr 1.000000e-08 reg 4.500000e+05 train accuracy: 0.416184 val accuracy: 0.419000
lr 1.000000e-08 reg 5.000000e+05 train accuracy: 0.414612 val accuracy: 0.417000
lr 1.000000e-08 reg 5.500000e+05 train accuracy: 0.415449 val accuracy: 0.411000
best validation accuracy achieved: 0.423000
```

[6]:
```
# Evaluate your trained SVM on the test set: you should be able to get at least␣
 ↪0.40
y_test_pred = best_svm.predict(X_test_feats)
test_accuracy = np.mean(y_test == y_test_pred)
print(test_accuracy)
```

```
0.412
```

[7]:
```
# An important way to gain intuition about how an algorithm works is to
# visualize the mistakes that it makes. In this visualization, we show examples
# of images that are misclassified by our current system. The first column
# shows images that our system labeled as "plane" but whose true label is
# something other than "plane".

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



1.3.1 Inline question 1:

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

Your Answer :

I think no. As the most highest accuracy is lower than 0.5, so we can predict there are many misclassifications.

In more details for the plane class, the images with bright background could get highsocres. In truck cases, there are less animal images but there are still some animals.

Generally, our classifier mostly could decide whether the image is veichle or animal. But it couldn't classify well into detail classes. And we could see the images with similar background tone were classified in ths same class.

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.

[8]:
```
# 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)
```

[22]:
```
from cs231n.classifiers.fc_net import TwoLayerNet
from cs231n.solver import Solver

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.
 ↪#
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
data = {
      'X_train': X_train_feats,
      'y_train': y_train,
      'X_val': X_val_feats,
      'y_val': y_val,
      'X_test' : X_test_feats,
      'y_test' : y_test
    }
best_acc = 0
results = {}
learning_rates = [5e-2,7.5e-2]
regularizations = [7.5e-4]
hidden_dims = [1000]
lr_decays = [0.95,1]

for h in hidden_dims:
  for l in learning_rates:
      for r in regularizations:
        for d in lr_decays:
          net = TwoLayerNet(input_dim, hidden_dim, num_classes, reg=r)
          solver = Solver(net, data,
                      update_rule='sgd',
                      optim_config={
                        'learning_rate': l,
                      },
                      lr_decay=d,
                      num_epochs=15, batch_size=100,
                      print_every = 100
                      )
          solver.train()
          val_acc = solver.val_acc_history[-1]
          train_acc = solver.train_acc_history[-1]
          if val_acc > best_acc:
              best_acc  = val_acc
              best_net = net
          results[(h,l,r,d)] = (train_acc,val_acc)

for hid, lr, reg, dec in sorted(results):
    train_acc, val_acc = results[(hid,lr,reg,dec)]
    print('hid %d lr %e reg %e dec %f -> train accuracy: %f val accuracy: %f' %
 ↪(
```

```
                hid, lr,reg, dec, train_acc, val_acc))
print('best validation accuracy achieved during cross-validation: %f' %␣
 ↪best_acc)
# We could get the highest accuracy when hid 1000 lr 7.5e-02 reg 7.5e-04 dec 0.
 ↪95
# train accuracy: 0.685000 val accuracy: 0.606000
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

```
(Iteration 1 / 7350) loss: 2.302637
(Epoch 0 / 15) train acc: 0.091000; val_acc: 0.113000
(Iteration 101 / 7350) loss: 2.298792
(Iteration 201 / 7350) loss: 2.225908
(Iteration 301 / 7350) loss: 1.977015
(Iteration 401 / 7350) loss: 1.886540
(Epoch 1 / 15) train acc: 0.413000; val_acc: 0.385000
(Iteration 501 / 7350) loss: 1.619968
(Iteration 601 / 7350) loss: 1.559841
(Iteration 701 / 7350) loss: 1.549589
(Iteration 801 / 7350) loss: 1.435941
(Iteration 901 / 7350) loss: 1.469190
(Epoch 2 / 15) train acc: 0.466000; val_acc: 0.492000
(Iteration 1001 / 7350) loss: 1.498903
(Iteration 1101 / 7350) loss: 1.365768
(Iteration 1201 / 7350) loss: 1.490767
(Iteration 1301 / 7350) loss: 1.329678
(Iteration 1401 / 7350) loss: 1.343050
(Epoch 3 / 15) train acc: 0.524000; val_acc: 0.516000
(Iteration 1501 / 7350) loss: 1.488644
(Iteration 1601 / 7350) loss: 1.306923
(Iteration 1701 / 7350) loss: 1.382592
(Iteration 1801 / 7350) loss: 1.180373
(Iteration 1901 / 7350) loss: 1.341741
(Epoch 4 / 15) train acc: 0.513000; val_acc: 0.521000
(Iteration 2001 / 7350) loss: 1.043674
(Iteration 2101 / 7350) loss: 1.342527
(Iteration 2201 / 7350) loss: 1.313248
(Iteration 2301 / 7350) loss: 1.260829
(Iteration 2401 / 7350) loss: 1.253048
(Epoch 5 / 15) train acc: 0.538000; val_acc: 0.528000
(Iteration 2501 / 7350) loss: 1.235293
(Iteration 2601 / 7350) loss: 1.522805
(Iteration 2701 / 7350) loss: 1.363956
(Iteration 2801 / 7350) loss: 1.289523
(Iteration 2901 / 7350) loss: 1.220184
(Epoch 6 / 15) train acc: 0.545000; val_acc: 0.529000
(Iteration 3001 / 7350) loss: 1.241970
(Iteration 3101 / 7350) loss: 1.383422
```

```
(Iteration 3201 / 7350) loss: 1.275302
(Iteration 3301 / 7350) loss: 1.361766
(Iteration 3401 / 7350) loss: 1.258789
(Epoch 7 / 15) train acc: 0.557000; val_acc: 0.545000
(Iteration 3501 / 7350) loss: 1.455399
(Iteration 3601 / 7350) loss: 1.069238
(Iteration 3701 / 7350) loss: 1.310937
(Iteration 3801 / 7350) loss: 1.089669
(Iteration 3901 / 7350) loss: 1.259042
(Epoch 8 / 15) train acc: 0.582000; val_acc: 0.548000
(Iteration 4001 / 7350) loss: 1.197906
(Iteration 4101 / 7350) loss: 1.195529
(Iteration 4201 / 7350) loss: 1.271367
(Iteration 4301 / 7350) loss: 1.182892
(Iteration 4401 / 7350) loss: 1.192620
(Epoch 9 / 15) train acc: 0.597000; val_acc: 0.551000
(Iteration 4501 / 7350) loss: 1.282212
(Iteration 4601 / 7350) loss: 1.074964
(Iteration 4701 / 7350) loss: 1.243546
(Iteration 4801 / 7350) loss: 1.183684
(Epoch 10 / 15) train acc: 0.603000; val_acc: 0.555000
(Iteration 4901 / 7350) loss: 1.208911
(Iteration 5001 / 7350) loss: 1.249232
(Iteration 5101 / 7350) loss: 1.119314
(Iteration 5201 / 7350) loss: 1.068691
(Iteration 5301 / 7350) loss: 1.318671
(Epoch 11 / 15) train acc: 0.605000; val_acc: 0.558000
(Iteration 5401 / 7350) loss: 1.310015
(Iteration 5501 / 7350) loss: 1.319869
(Iteration 5601 / 7350) loss: 1.163894
(Iteration 5701 / 7350) loss: 1.160541
(Iteration 5801 / 7350) loss: 1.122808
(Epoch 12 / 15) train acc: 0.594000; val_acc: 0.559000
(Iteration 5901 / 7350) loss: 1.241741
(Iteration 6001 / 7350) loss: 1.216470
(Iteration 6101 / 7350) loss: 1.307397
(Iteration 6201 / 7350) loss: 1.187155
(Iteration 6301 / 7350) loss: 1.101535
(Epoch 13 / 15) train acc: 0.633000; val_acc: 0.563000
(Iteration 6401 / 7350) loss: 0.999084
(Iteration 6501 / 7350) loss: 1.057238
(Iteration 6601 / 7350) loss: 1.221072
(Iteration 6701 / 7350) loss: 0.936815
(Iteration 6801 / 7350) loss: 1.183464
(Epoch 14 / 15) train acc: 0.648000; val_acc: 0.571000
(Iteration 6901 / 7350) loss: 1.101150
(Iteration 7001 / 7350) loss: 1.283738
(Iteration 7101 / 7350) loss: 1.022380
```

```
(Iteration 7201 / 7350) loss: 0.833682
(Iteration 7301 / 7350) loss: 1.177640
(Epoch 15 / 15) train acc: 0.601000; val_acc: 0.569000
(Iteration 1 / 7350) loss: 2.302612
(Epoch 0 / 15) train acc: 0.095000; val_acc: 0.100000
(Iteration 101 / 7350) loss: 2.300395
(Iteration 201 / 7350) loss: 2.237987
(Iteration 301 / 7350) loss: 1.951894
(Iteration 401 / 7350) loss: 1.810957
(Epoch 1 / 15) train acc: 0.405000; val_acc: 0.397000
(Iteration 501 / 7350) loss: 1.649346
(Iteration 601 / 7350) loss: 1.587139
(Iteration 701 / 7350) loss: 1.533080
(Iteration 801 / 7350) loss: 1.584612
(Iteration 901 / 7350) loss: 1.384245
(Epoch 2 / 15) train acc: 0.518000; val_acc: 0.493000
(Iteration 1001 / 7350) loss: 1.385464
(Iteration 1101 / 7350) loss: 1.413110
(Iteration 1201 / 7350) loss: 1.417637
(Iteration 1301 / 7350) loss: 1.227583
(Iteration 1401 / 7350) loss: 1.349809
(Epoch 3 / 15) train acc: 0.562000; val_acc: 0.507000
(Iteration 1501 / 7350) loss: 1.283625
(Iteration 1601 / 7350) loss: 1.542105
(Iteration 1701 / 7350) loss: 1.164401
(Iteration 1801 / 7350) loss: 1.085740
(Iteration 1901 / 7350) loss: 1.445053
(Epoch 4 / 15) train acc: 0.530000; val_acc: 0.530000
(Iteration 2001 / 7350) loss: 1.368702
(Iteration 2101 / 7350) loss: 1.183658
(Iteration 2201 / 7350) loss: 1.488109
(Iteration 2301 / 7350) loss: 1.436086
(Iteration 2401 / 7350) loss: 1.155782
(Epoch 5 / 15) train acc: 0.564000; val_acc: 0.524000
(Iteration 2501 / 7350) loss: 1.381722
(Iteration 2601 / 7350) loss: 1.305464
(Iteration 2701 / 7350) loss: 1.291186
(Iteration 2801 / 7350) loss: 1.244308
(Iteration 2901 / 7350) loss: 1.246414
(Epoch 6 / 15) train acc: 0.566000; val_acc: 0.536000
(Iteration 3001 / 7350) loss: 1.240748
(Iteration 3101 / 7350) loss: 1.263657
(Iteration 3201 / 7350) loss: 1.136563
(Iteration 3301 / 7350) loss: 1.254294
(Iteration 3401 / 7350) loss: 1.259034
(Epoch 7 / 15) train acc: 0.563000; val_acc: 0.558000
(Iteration 3501 / 7350) loss: 1.224417
(Iteration 3601 / 7350) loss: 0.953222
```

```
(Iteration 3701 / 7350) loss: 1.332598
(Iteration 3801 / 7350) loss: 1.447310
(Iteration 3901 / 7350) loss: 1.193288
(Epoch 8 / 15) train acc: 0.571000; val_acc: 0.559000
(Iteration 4001 / 7350) loss: 1.177468
(Iteration 4101 / 7350) loss: 1.318500
(Iteration 4201 / 7350) loss: 1.465835
(Iteration 4301 / 7350) loss: 1.208174
(Iteration 4401 / 7350) loss: 1.096750
(Epoch 9 / 15) train acc: 0.576000; val_acc: 0.555000
(Iteration 4501 / 7350) loss: 1.234626
(Iteration 4601 / 7350) loss: 1.132016
(Iteration 4701 / 7350) loss: 1.210080
(Iteration 4801 / 7350) loss: 1.341429
(Epoch 10 / 15) train acc: 0.610000; val_acc: 0.571000
(Iteration 4901 / 7350) loss: 1.198891
(Iteration 5001 / 7350) loss: 1.423332
(Iteration 5101 / 7350) loss: 1.144722
(Iteration 5201 / 7350) loss: 1.108890
(Iteration 5301 / 7350) loss: 1.076879
(Epoch 11 / 15) train acc: 0.636000; val_acc: 0.578000
(Iteration 5401 / 7350) loss: 1.288067
(Iteration 5501 / 7350) loss: 1.104806
(Iteration 5601 / 7350) loss: 1.321237
(Iteration 5701 / 7350) loss: 1.181936
(Iteration 5801 / 7350) loss: 1.017538
(Epoch 12 / 15) train acc: 0.631000; val_acc: 0.583000
(Iteration 5901 / 7350) loss: 1.144165
(Iteration 6001 / 7350) loss: 1.178819
(Iteration 6101 / 7350) loss: 1.067563
(Iteration 6201 / 7350) loss: 1.126763
(Iteration 6301 / 7350) loss: 1.083036
(Epoch 13 / 15) train acc: 0.649000; val_acc: 0.591000
(Iteration 6401 / 7350) loss: 1.224166
(Iteration 6501 / 7350) loss: 1.189684
(Iteration 6601 / 7350) loss: 1.061268
(Iteration 6701 / 7350) loss: 1.073349
(Iteration 6801 / 7350) loss: 1.124049
(Epoch 14 / 15) train acc: 0.653000; val_acc: 0.591000
(Iteration 6901 / 7350) loss: 1.053357
(Iteration 7001 / 7350) loss: 0.941507
(Iteration 7101 / 7350) loss: 0.950950
(Iteration 7201 / 7350) loss: 0.939852
(Iteration 7301 / 7350) loss: 0.911124
(Epoch 15 / 15) train acc: 0.662000; val_acc: 0.599000
(Iteration 1 / 7350) loss: 2.302638
(Epoch 0 / 15) train acc: 0.113000; val_acc: 0.112000
(Iteration 101 / 7350) loss: 2.292105
```

```
(Iteration 201 / 7350) loss: 1.926618
(Iteration 301 / 7350) loss: 1.683028
(Iteration 401 / 7350) loss: 1.463754
(Epoch 1 / 15) train acc: 0.475000; val_acc: 0.469000
(Iteration 501 / 7350) loss: 1.428909
(Iteration 601 / 7350) loss: 1.510136
(Iteration 701 / 7350) loss: 1.419903
(Iteration 801 / 7350) loss: 1.396947
(Iteration 901 / 7350) loss: 1.381335
(Epoch 2 / 15) train acc: 0.545000; val_acc: 0.507000
(Iteration 1001 / 7350) loss: 1.458191
(Iteration 1101 / 7350) loss: 1.387379
(Iteration 1201 / 7350) loss: 1.221595
(Iteration 1301 / 7350) loss: 1.256643
(Iteration 1401 / 7350) loss: 1.378597
(Epoch 3 / 15) train acc: 0.531000; val_acc: 0.525000
(Iteration 1501 / 7350) loss: 1.242743
(Iteration 1601 / 7350) loss: 1.123136
(Iteration 1701 / 7350) loss: 1.399724
(Iteration 1801 / 7350) loss: 1.223332
(Iteration 1901 / 7350) loss: 1.147880
(Epoch 4 / 15) train acc: 0.532000; val_acc: 0.531000
(Iteration 2001 / 7350) loss: 1.277556
(Iteration 2101 / 7350) loss: 1.229457
(Iteration 2201 / 7350) loss: 1.283694
(Iteration 2301 / 7350) loss: 1.035781
(Iteration 2401 / 7350) loss: 1.276396
(Epoch 5 / 15) train acc: 0.577000; val_acc: 0.547000
(Iteration 2501 / 7350) loss: 1.009870
(Iteration 2601 / 7350) loss: 1.254372
(Iteration 2701 / 7350) loss: 1.407223
(Iteration 2801 / 7350) loss: 1.116028
(Iteration 2901 / 7350) loss: 1.251909
(Epoch 6 / 15) train acc: 0.575000; val_acc: 0.544000
(Iteration 3001 / 7350) loss: 1.221936
(Iteration 3101 / 7350) loss: 1.416140
(Iteration 3201 / 7350) loss: 1.141047
(Iteration 3301 / 7350) loss: 1.274481
(Iteration 3401 / 7350) loss: 1.222229
(Epoch 7 / 15) train acc: 0.572000; val_acc: 0.570000
(Iteration 3501 / 7350) loss: 1.122615
(Iteration 3601 / 7350) loss: 1.203105
(Iteration 3701 / 7350) loss: 1.147558
(Iteration 3801 / 7350) loss: 1.113586
(Iteration 3901 / 7350) loss: 0.993502
(Epoch 8 / 15) train acc: 0.603000; val_acc: 0.583000
(Iteration 4001 / 7350) loss: 1.145129
(Iteration 4101 / 7350) loss: 1.058000
```

```
(Iteration 4201 / 7350) loss: 0.976771
(Iteration 4301 / 7350) loss: 1.101208
(Iteration 4401 / 7350) loss: 1.330676
(Epoch 9 / 15) train acc: 0.603000; val_acc: 0.581000
(Iteration 4501 / 7350) loss: 1.026608
(Iteration 4601 / 7350) loss: 1.205496
(Iteration 4701 / 7350) loss: 1.182552
(Iteration 4801 / 7350) loss: 1.064185
(Epoch 10 / 15) train acc: 0.627000; val_acc: 0.593000
(Iteration 4901 / 7350) loss: 0.936072
(Iteration 5001 / 7350) loss: 1.139640
(Iteration 5101 / 7350) loss: 1.108754
(Iteration 5201 / 7350) loss: 1.230305
(Iteration 5301 / 7350) loss: 1.027178
(Epoch 11 / 15) train acc: 0.642000; val_acc: 0.606000
(Iteration 5401 / 7350) loss: 1.041193
(Iteration 5501 / 7350) loss: 1.021880
(Iteration 5601 / 7350) loss: 1.222774
(Iteration 5701 / 7350) loss: 0.927519
(Iteration 5801 / 7350) loss: 1.009441
(Epoch 12 / 15) train acc: 0.633000; val_acc: 0.603000
(Iteration 5901 / 7350) loss: 1.093675
(Iteration 6001 / 7350) loss: 1.110292
(Iteration 6101 / 7350) loss: 1.022119
(Iteration 6201 / 7350) loss: 1.017833
(Iteration 6301 / 7350) loss: 1.057401
(Epoch 13 / 15) train acc: 0.661000; val_acc: 0.593000
(Iteration 6401 / 7350) loss: 1.273984
(Iteration 6501 / 7350) loss: 1.116996
(Iteration 6601 / 7350) loss: 1.046503
(Iteration 6701 / 7350) loss: 1.084690
(Iteration 6801 / 7350) loss: 0.944342
(Epoch 14 / 15) train acc: 0.664000; val_acc: 0.602000
(Iteration 6901 / 7350) loss: 0.835705
(Iteration 7001 / 7350) loss: 0.951573
(Iteration 7101 / 7350) loss: 0.837838
(Iteration 7201 / 7350) loss: 1.148888
(Iteration 7301 / 7350) loss: 0.940799
(Epoch 15 / 15) train acc: 0.685000; val_acc: 0.606000
(Iteration 1 / 7350) loss: 2.302619
(Epoch 0 / 15) train acc: 0.106000; val_acc: 0.112000
(Iteration 101 / 7350) loss: 2.288060
(Iteration 201 / 7350) loss: 2.047102
(Iteration 301 / 7350) loss: 1.769512
(Iteration 401 / 7350) loss: 1.463092
(Epoch 1 / 15) train acc: 0.498000; val_acc: 0.462000
(Iteration 501 / 7350) loss: 1.488804
(Iteration 601 / 7350) loss: 1.387770
```

```
(Iteration 701 / 7350) loss: 1.374312
(Iteration 801 / 7350) loss: 1.229878
(Iteration 901 / 7350) loss: 1.275084
(Epoch 2 / 15) train acc: 0.520000; val_acc: 0.510000
(Iteration 1001 / 7350) loss: 1.509392
(Iteration 1101 / 7350) loss: 1.397519
(Iteration 1201 / 7350) loss: 1.209838
(Iteration 1301 / 7350) loss: 1.443855
(Iteration 1401 / 7350) loss: 1.367120
(Epoch 3 / 15) train acc: 0.560000; val_acc: 0.518000
(Iteration 1501 / 7350) loss: 1.269744
(Iteration 1601 / 7350) loss: 1.323169
(Iteration 1701 / 7350) loss: 1.193759
(Iteration 1801 / 7350) loss: 1.413514
(Iteration 1901 / 7350) loss: 1.386219
(Epoch 4 / 15) train acc: 0.572000; val_acc: 0.529000
(Iteration 2001 / 7350) loss: 1.166136
(Iteration 2101 / 7350) loss: 1.139600
(Iteration 2201 / 7350) loss: 1.353914
(Iteration 2301 / 7350) loss: 1.257243
(Iteration 2401 / 7350) loss: 1.239450
(Epoch 5 / 15) train acc: 0.574000; val_acc: 0.551000
(Iteration 2501 / 7350) loss: 1.404887
(Iteration 2601 / 7350) loss: 1.217464
(Iteration 2701 / 7350) loss: 1.394538
(Iteration 2801 / 7350) loss: 1.198220
(Iteration 2901 / 7350) loss: 1.298651
(Epoch 6 / 15) train acc: 0.587000; val_acc: 0.545000
(Iteration 3001 / 7350) loss: 1.083325
(Iteration 3101 / 7350) loss: 1.197822
(Iteration 3201 / 7350) loss: 1.140229
(Iteration 3301 / 7350) loss: 1.113023
(Iteration 3401 / 7350) loss: 1.241779
(Epoch 7 / 15) train acc: 0.617000; val_acc: 0.560000
(Iteration 3501 / 7350) loss: 1.148170
(Iteration 3601 / 7350) loss: 1.149429
(Iteration 3701 / 7350) loss: 1.163261
(Iteration 3801 / 7350) loss: 1.058150
(Iteration 3901 / 7350) loss: 0.951148
(Epoch 8 / 15) train acc: 0.642000; val_acc: 0.584000
(Iteration 4001 / 7350) loss: 1.203359
(Iteration 4101 / 7350) loss: 1.080620
(Iteration 4201 / 7350) loss: 1.106543
(Iteration 4301 / 7350) loss: 1.174770
(Iteration 4401 / 7350) loss: 0.971756
(Epoch 9 / 15) train acc: 0.639000; val_acc: 0.584000
(Iteration 4501 / 7350) loss: 0.985334
(Iteration 4601 / 7350) loss: 1.020206
```

```
(Iteration 4701 / 7350) loss: 1.204021
(Iteration 4801 / 7350) loss: 1.180934
(Epoch 10 / 15) train acc: 0.695000; val_acc: 0.587000
(Iteration 4901 / 7350) loss: 1.078842
(Iteration 5001 / 7350) loss: 0.966815
(Iteration 5101 / 7350) loss: 0.950071
(Iteration 5201 / 7350) loss: 1.018168
(Iteration 5301 / 7350) loss: 0.990872
(Epoch 11 / 15) train acc: 0.654000; val_acc: 0.603000
(Iteration 5401 / 7350) loss: 1.023716
(Iteration 5501 / 7350) loss: 1.013031
(Iteration 5601 / 7350) loss: 1.040342
(Iteration 5701 / 7350) loss: 0.874436
(Iteration 5801 / 7350) loss: 0.979286
(Epoch 12 / 15) train acc: 0.663000; val_acc: 0.604000
(Iteration 5901 / 7350) loss: 0.864236
(Iteration 6001 / 7350) loss: 1.095502
(Iteration 6101 / 7350) loss: 0.941411
(Iteration 6201 / 7350) loss: 0.976685
(Iteration 6301 / 7350) loss: 1.112620
(Epoch 13 / 15) train acc: 0.697000; val_acc: 0.586000
(Iteration 6401 / 7350) loss: 0.811316
(Iteration 6501 / 7350) loss: 0.773595
(Iteration 6601 / 7350) loss: 1.010268
(Iteration 6701 / 7350) loss: 1.248723
(Iteration 6801 / 7350) loss: 1.033883
(Epoch 14 / 15) train acc: 0.691000; val_acc: 0.596000
(Iteration 6901 / 7350) loss: 0.907942
(Iteration 7001 / 7350) loss: 0.908795
(Iteration 7101 / 7350) loss: 1.009889
(Iteration 7201 / 7350) loss: 0.960158
(Iteration 7301 / 7350) loss: 1.157532
(Epoch 15 / 15) train acc: 0.728000; val_acc: 0.596000
hid 1000 lr 5.000000e-02 reg 7.500000e-04 dec 0.950000 -> train accuracy:
0.601000 val accuracy: 0.569000
hid 1000 lr 5.000000e-02 reg 7.500000e-04 dec 1.000000 -> train accuracy:
0.662000 val accuracy: 0.599000
hid 1000 lr 7.500000e-02 reg 7.500000e-04 dec 0.950000 -> train accuracy:
0.685000 val accuracy: 0.606000
hid 1000 lr 7.500000e-02 reg 7.500000e-04 dec 1.000000 -> train accuracy:
0.728000 val accuracy: 0.596000
best validation accuracy achieved during cross-validation: 0.606000
```

[23]:
```python
# Run your best neural net classifier on the test set. You should be able
# to get more than 55% accuracy.

y_test_pred = np.argmax(best_net.loss(data['X_test']), axis=1)
```

```
test_acc = (y_test_pred == data['y_test']).mean()
print(test_acc)
```

```
0.576
```