knn

September 6, 2022

[26]:
```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive')

# TODO: 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 /content/drive/My\ Drive/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
/content/drive/My Drive/cs231n/assignments/assignment1/cs231n/datasets
/content/drive/My Drive/cs231n/assignments/assignment1
```

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.

[27]:
```
# 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
```

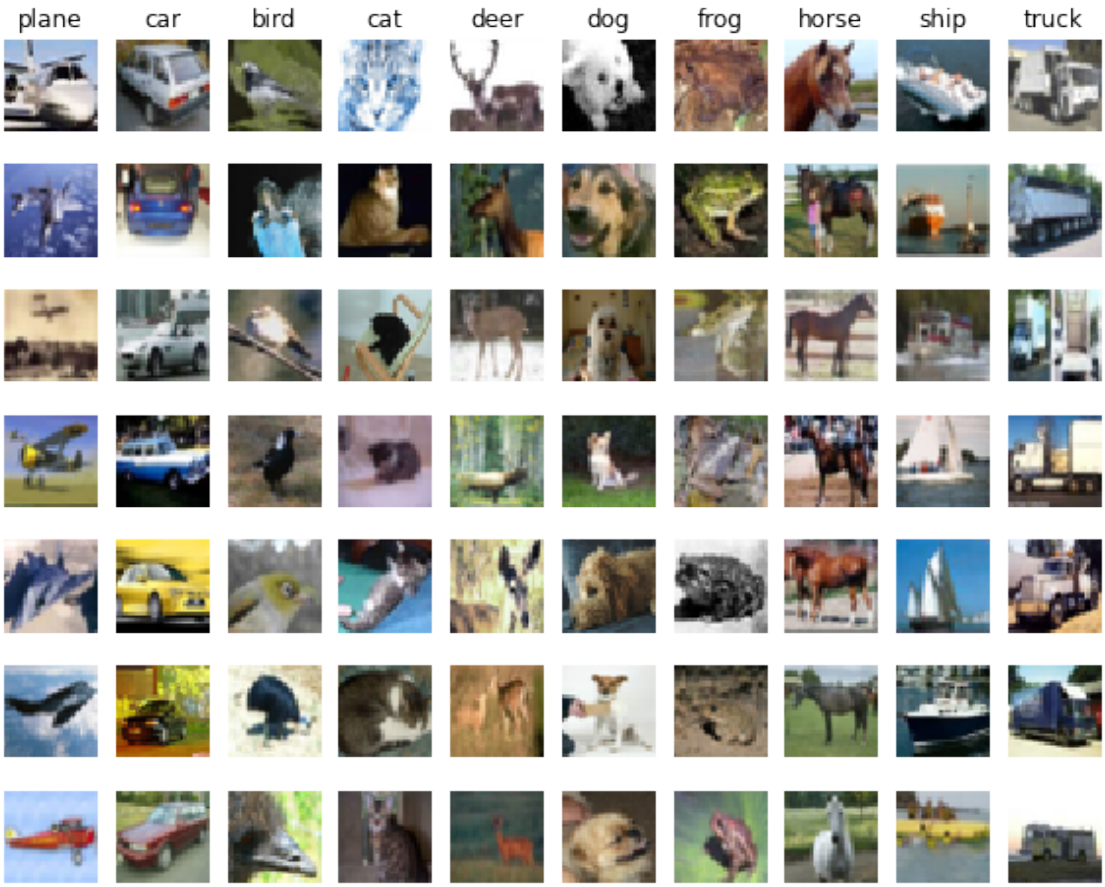
```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

[28]:
```
# 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)
```

```
Clear previously loaded data.
```

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

[29]:
```
# 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()
```



[30]:
```
# 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]

print(X_train.shape, X_test.shape)

# Reshape the image data into rows, reshape 前后没区别
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, 32, 32, 3) (500, 32, 32, 3)
(5000, 3072) (500, 3072)
```

[31]:
```
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) # lazy learning, 仅识别的时候进行分析，训练没有作用
```

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.

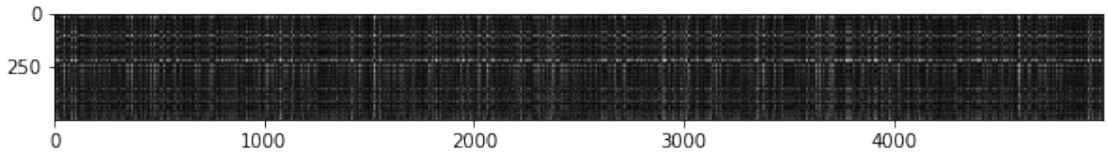
Note: For the three distance computations that we require you to implement in this notebook, you may not use the np.linalg.norm() function that numpy provides.

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.

[32]:
```
# 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)
```

[33]:
```
# 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 visibly 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：

- 这张图片与训练集中大部分图片的欧几里得距离较远，说明类似这张图片的图片在训练集中很少出现
- 这张图片与测试集中大部分图片的欧几里得距离较远，说明类似这张图片的图片在测试集中很少出现

[34]:
```
# 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`:

[24]:
```
y_test_pred = classifier.predict_labels(dists, k=5)
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
```

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

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

Inline Question 2

We can also 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

测试集中所有图片所有像素的平均值

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

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

所有照片 [i,j] 处像素的平均值

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

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：

没有变化的：1,2,3,5

YourExplanation：

1. 所有照片的所有像素减去一个相同的值
2. 所有照片的同一位置像素减去一个相同值
3. ($\tilde{p}_{ij}^{(k)} = \frac{p_{ij}^{(k)} - \mu}{\sigma}$)

所有照片的所有位置像素减去相同值后缩放相同值 4. ($\tilde{p}_{ij}^{(k)} = \frac{p_{ij}^{(k)} - \mu_{ij}}{\sigma_{ij}}$)

所有照片的同一位置像素减去相同值后缩放相同值

5. 将图片旋转

knn 使用两个图片对应像素值的差值作为分类的依据，1 和 2 同一位置减去的同一个值可以抵消，不会影响结果；

3 在 1 和 2 的基础上对所有像素等倍缩放，是矩阵的线性变换，对所有图片和像素的影响程度相同，不会影响结果。

5 对图片旋转，相当于对矩阵进行转置，不会影响两张图片像素的对应关系，也不会影响结果。

[40]:
```
# 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
```

[103]:
```
# 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
```

[104]:
```
# 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 33.958122 seconds
One loop version took 40.322520 seconds
No loop version took 0.578593 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.

[152]:
```
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)
# print(X_train_folds)
y_train_folds = np.array_split(y_train, num_folds)
```

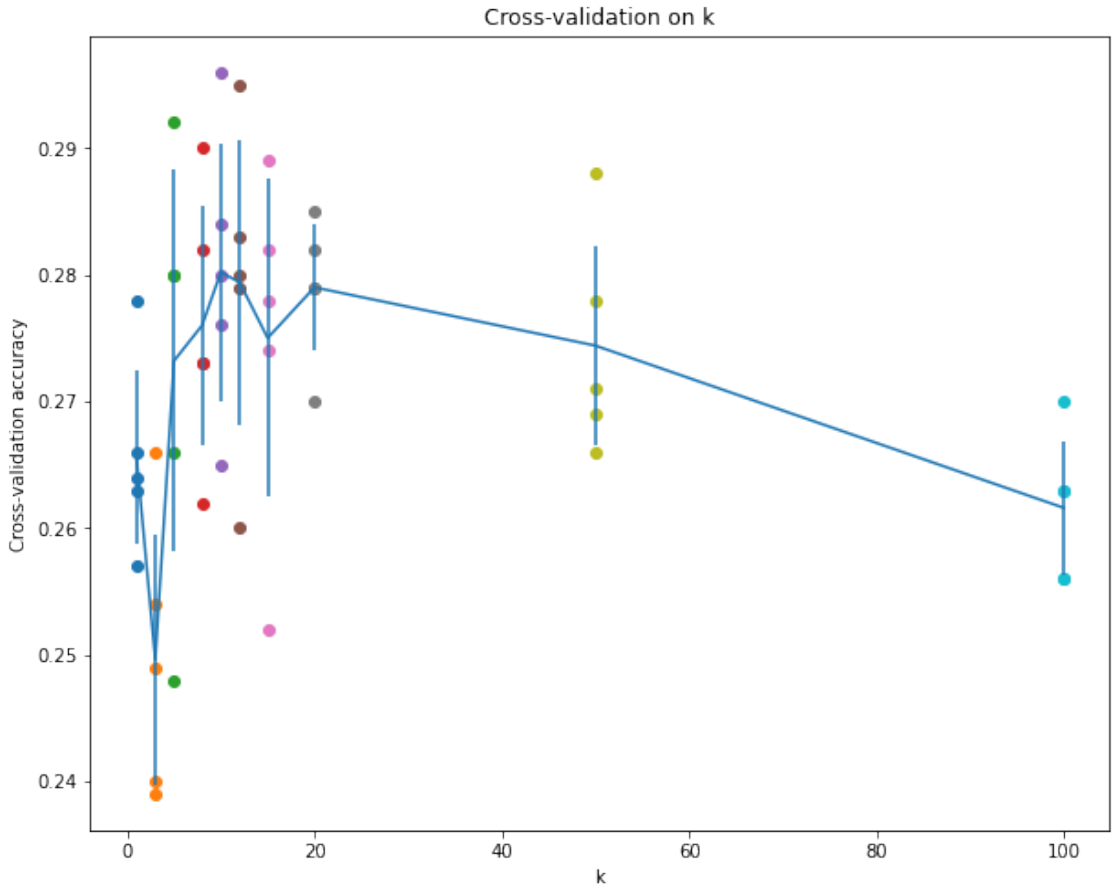
```
# print(y_train_folds)

# *****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:
  k_to_accuracies[k]=[]
  for i in range(num_folds):
    # Subsample the data for more efficient code execution in this exercise
    # print(f'new_X_train[{new_X_train}],new_y_train[{new_y_train}]')
    idxs = [j for j in range(num_folds) if j != i]
    new_X_train=np.concatenate([X_train_folds[j] for j in idxs],axis=0)
    new_y_train=np.concatenate([y_train_folds[j] for j in idxs],axis=0)
    # print(new_X_train)

    new_X_test = X_train_folds[i]
    new_y_test = y_train_folds[i]

    new_num_test = len(new_y_test)

    # Reshape the image data into rows
    new_X_train = np.reshape(new_X_train, (new_X_train.shape[0], -1))
    new_X_test = np.reshape(new_X_test, (new_X_test.shape[0], -1))
    # print(new_X_train.shape, new_X_test.shape)

    classifier.train(new_X_train, new_y_train) # lazy learning, 仅识别的时候进行
分析，训练没有作用

    new_dists = classifier.compute_distances_no_loops(new_X_test)
    # print(new_dists.shape)
```

```
    y_test_pred = classifier.predict_labels(new_dists, k)

    num_correct = np.sum(y_test_pred == new_y_test)
    accuracy = float(num_correct) / new_num_test
    # print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test,␣
 ↪accuracy))
    k_to_accuracies[k].append(accuracy)

# print(k_to_accuracies)
# *****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 = 0.239000
k = 3, accuracy = 0.249000
k = 3, accuracy = 0.240000
k = 3, accuracy = 0.266000
k = 3, accuracy = 0.254000
k = 5, accuracy = 0.248000
k = 5, accuracy = 0.266000
k = 5, accuracy = 0.280000
k = 5, accuracy = 0.292000
k = 5, accuracy = 0.280000
k = 8, accuracy = 0.262000
k = 8, accuracy = 0.282000
k = 8, accuracy = 0.273000
k = 8, accuracy = 0.290000
k = 8, accuracy = 0.273000
k = 10, accuracy = 0.265000
k = 10, accuracy = 0.296000
k = 10, accuracy = 0.276000
k = 10, accuracy = 0.284000
k = 10, accuracy = 0.280000
k = 12, accuracy = 0.260000
k = 12, accuracy = 0.295000
k = 12, accuracy = 0.279000
k = 12, accuracy = 0.283000
k = 12, accuracy = 0.280000
k = 15, accuracy = 0.252000
```

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

[153]:
```
# 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()
```



[156]:
```
# Based on the cross-validation results above, choose the best value for k,
# retrain the classifier using all the training data, and test it on the test
# data. You should be able to get above 28% accuracy on the test data.
best_k = 10

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

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

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

Inline Question 3

Which of the following statements about k-Nearest Neighbor (k-NN) are true in a classification setting, and for all k? Select all that apply. 1. The 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：

1.False 2.True 3.False 4.True

YourExplanation：

1. knn 没有类似 svm 的超平面，对 test point 进行分类的时候，用于训练的点的边界通常是曲线，即 knn classifier 是非线性的。
2. training error 指使用训练集作为数据集进行预测时，同一个点可能出现的差错。由于 k=1 时仅使用最近的一个点进行预测，而对测试集为训练集的情况来说，离一个点自己最近的就是自己，故 1-nn 的 traing error 永远不大于 5-nn。
3. test error 指使用训练集之外的数据作为数据集进行预测时出现的差错。假设有一个一维情况，一共有 2 类，训练集为 $X_{train} = \{1, 1, 1, 2, 2\}$,$y_{train} = \{0, 0, 0, 1, 1\}$, 对于数据 $x = 3, y = 0$,1nn 认为他属于类 1，而 5nn 认为他属于类 0。
4. knn 每次预测的时候都需要计算所给数据与训练集数据的所有距离，训练集越大，计算距离的计算量就越大，需要时间就越长。

svm

September 6, 2022

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

# TODO: 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 /content/drive/My\ Drive/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
/content/drive/My Drive/cs231n/assignments/assignment1/cs231n/datasets
/content/drive/My Drive/cs231n/assignments/assignment1
```

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

[ ]:
```
# 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
```

1.1 CIFAR-10 Data Loading and Preprocessing

[ ]:
```
# 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)
```

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

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

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

[ ]:
```
# 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)
```

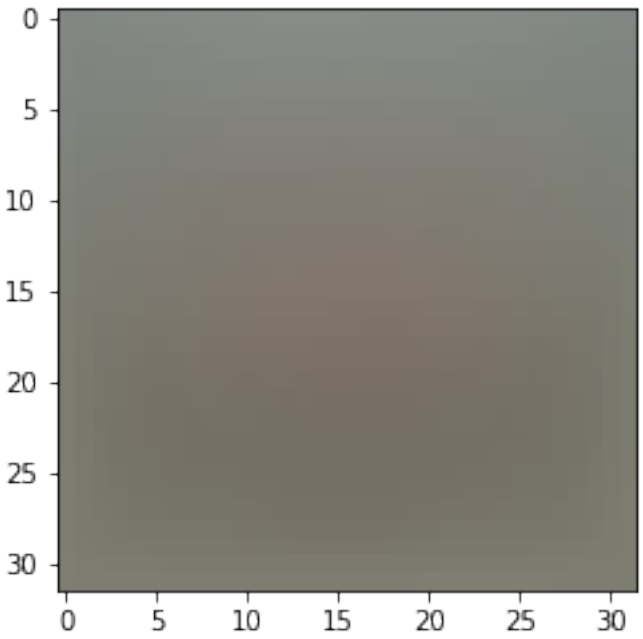
```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

[ ]:
```
# 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)
```

[ ]:
```
# 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.
# 新增一列全 1, 以消去参数 b
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.

[ ]:
```
# 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: 9.401189
```

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, 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:

[ ]:
```
# 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: -10.408229 analytic: -10.408229, relative error: 4.402479e-11
numerical: 27.476993 analytic: 27.476993, relative error: 1.855623e-11
numerical: 2.280600 analytic: 2.280600, relative error: 1.469642e-10
numerical: -19.094903 analytic: -19.094903, relative error: 5.184346e-12
numerical: 15.157032 analytic: 15.157032, relative error: 1.666856e-11
numerical: -28.282057 analytic: -28.282057, relative error: 1.052069e-11
numerical: -7.334477 analytic: -7.334477, relative error: 6.552705e-11
numerical: -10.706777 analytic: -10.706777, relative error: 3.246013e-11
numerical: -3.125464 analytic: -3.125464, relative error: 1.633782e-10
numerical: -1.752699 analytic: -1.752699, relative error: 1.570124e-10
numerical: -12.406846 analytic: -12.406846, relative error: 3.446879e-12
numerical: -28.903089 analytic: -28.903089, relative error: 7.962022e-13
numerical: 1.964044 analytic: 1.964044, relative error: 1.052466e-10
numerical: -53.543882 analytic: -53.543882, relative error: 5.434979e-12
numerical: 17.290358 analytic: 17.290358, relative error: 7.998783e-12
numerical: 3.992767 analytic: 3.992767, relative error: 3.401323e-11
numerical: 39.058781 analytic: 39.058781, relative error: 1.458996e-11
numerical: 30.773074 analytic: 30.773074, relative error: 4.611024e-12
numerical: 7.486540 analytic: 7.486540, relative error: 1.810384e-11
numerical: 1.236683 analytic: 1.236683, relative error: 3.594599e-10
```

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*

YourAnswer :

SVM 损失函数为 $\max(S_j - S_{y_i} + 1, 0)$, 在 $S_j - S_{y_i} + 1$ 处 aSVM 损失函数为 $\max(S_j - S_{y_i} + 1, 0)$, 在 $S_j - S_{y_i} + 1$ 处不可微, 所以 $S_j - S_{y_i}$ 在 0 的两侧,*Loss* 函数值会发生跳变, 导致分析方法和数值方法结果不吻合。

如果 margin 变小, 这种情况出现的频率会降低。

[ ]:
```
# 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: 9.401189e+00 computed in 0.128910s
Vectorized loss: 9.401189e+00 computed in 0.013510s
difference: -0.000000
```

[ ]:
```
# 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.
# print(f'grad_naive:\n{grad_naive}')
# print(f'grad_vectorized:\n{grad_vectorized}')
```

```
difference = np.linalg.norm(grad_naive - grad_vectorized, ord='fro')
print('difference: %f' % difference)
```

```
Naive loss and gradient: computed in 0.121577s
Vectorized loss and gradient: computed in 0.018211s
difference: 0.000000
```

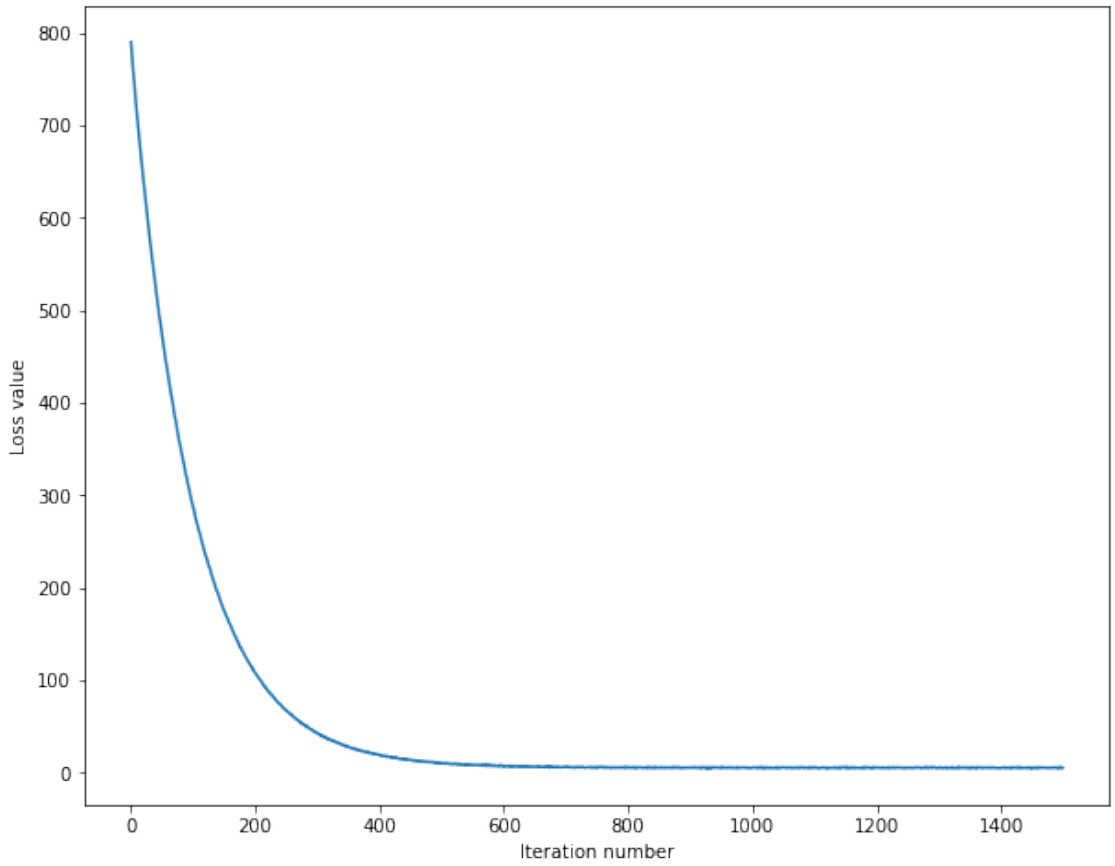
1.2.1 Stochastic Gradient Descent

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`.

[ ]:
```
# 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 789.920601
iteration 100 / 1500: loss 286.499248
iteration 200 / 1500: loss 107.763667
iteration 300 / 1500: loss 42.455947
iteration 400 / 1500: loss 19.419767
iteration 500 / 1500: loss 10.389614
iteration 600 / 1500: loss 7.432982
iteration 700 / 1500: loss 6.049148
iteration 800 / 1500: loss 5.373677
iteration 900 / 1500: loss 5.360467
iteration 1000 / 1500: loss 5.176213
iteration 1100 / 1500: loss 5.113732
iteration 1200 / 1500: loss 5.311092
iteration 1300 / 1500: loss 5.389933
iteration 1400 / 1500: loss 4.903446
That took 12.533612s
```

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



[ ]:
```
# 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: 0.364959
validation accuracy: 0.387000
```

[ ]:
```
# 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 (> 0.385) 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 = [1e-7, 5e-8]
learning_rates = np.arange(1e-8,1e-6,1e-7)
# regularization_strengths = [2.5e4, 5e4]
regularization_strengths = np.arange(1e3,100e3,10e3)

# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

for lr in learning_rates:
  for reg in regularization_strengths:
    new_svm = LinearSVM()
    new_svm.train(X_train, y_train, learning_rate=lr, reg=reg,
                      num_iters=1500, verbose=False)

    new_y_train_pred = new_svm.predict(X_train)
    new_y_val_pred = new_svm.predict(X_val)

    new_train_accuracy = np.mean(y_train == new_y_train_pred)
    new_val_accuracy = np.mean(y_val == new_y_val_pred)

    print('lr %e reg %e train accuracy: %f val accuracy: %f' % (
                lr, reg, new_train_accuracy, new_val_accuracy),end='\n\n')

    results.update({(lr, reg):(new_train_accuracy, new_val_accuracy)})
```

```
    if new_val_accuracy > best_val:
      print('new best val')
      best_val = new_val_accuracy
      best_svm = new_svm

# *****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)
```

```
lr 1.000000e-08 reg 1.000000e+03 train accuracy: 0.230041 val accuracy: 0.234000

new best val
lr 1.000000e-08 reg 1.100000e+04 train accuracy: 0.241531 val accuracy: 0.268000

new best val
lr 1.000000e-08 reg 2.100000e+04 train accuracy: 0.252918 val accuracy: 0.249000

lr 1.000000e-08 reg 3.100000e+04 train accuracy: 0.268347 val accuracy: 0.266000

lr 1.000000e-08 reg 4.100000e+04 train accuracy: 0.284367 val accuracy: 0.296000

new best val
lr 1.000000e-08 reg 5.100000e+04 train accuracy: 0.299857 val accuracy: 0.308000

new best val
lr 1.000000e-08 reg 6.100000e+04 train accuracy: 0.321286 val accuracy: 0.324000

new best val
lr 1.000000e-08 reg 7.100000e+04 train accuracy: 0.329408 val accuracy: 0.324000

lr 1.000000e-08 reg 8.100000e+04 train accuracy: 0.335612 val accuracy: 0.360000

new best val
lr 1.000000e-08 reg 9.100000e+04 train accuracy: 0.340816 val accuracy: 0.357000

lr 1.100000e-07 reg 1.000000e+03 train accuracy: 0.323408 val accuracy: 0.329000

lr 1.100000e-07 reg 1.100000e+04 train accuracy: 0.383551 val accuracy: 0.396000

new best val
```

```
lr 1.100000e-07 reg 2.100000e+04 train accuracy: 0.369857 val accuracy: 0.375000

lr 1.100000e-07 reg 3.100000e+04 train accuracy: 0.356306 val accuracy: 0.358000

lr 1.100000e-07 reg 4.100000e+04 train accuracy: 0.360061 val accuracy: 0.359000

lr 1.100000e-07 reg 5.100000e+04 train accuracy: 0.347918 val accuracy: 0.353000

lr 1.100000e-07 reg 6.100000e+04 train accuracy: 0.353694 val accuracy: 0.357000

lr 1.100000e-07 reg 7.100000e+04 train accuracy: 0.344041 val accuracy: 0.359000

lr 1.100000e-07 reg 8.100000e+04 train accuracy: 0.346776 val accuracy: 0.358000

lr 1.100000e-07 reg 9.100000e+04 train accuracy: 0.344837 val accuracy: 0.354000

lr 2.100000e-07 reg 1.000000e+03 train accuracy: 0.357327 val accuracy: 0.356000

lr 2.100000e-07 reg 1.100000e+04 train accuracy: 0.382592 val accuracy: 0.413000

new best val
lr 2.100000e-07 reg 2.100000e+04 train accuracy: 0.362959 val accuracy: 0.377000

lr 2.100000e-07 reg 3.100000e+04 train accuracy: 0.345571 val accuracy: 0.374000

lr 2.100000e-07 reg 4.100000e+04 train accuracy: 0.344082 val accuracy: 0.376000

lr 2.100000e-07 reg 5.100000e+04 train accuracy: 0.345735 val accuracy: 0.361000

lr 2.100000e-07 reg 6.100000e+04 train accuracy: 0.343592 val accuracy: 0.357000

lr 2.100000e-07 reg 7.100000e+04 train accuracy: 0.338020 val accuracy: 0.350000

lr 2.100000e-07 reg 8.100000e+04 train accuracy: 0.333449 val accuracy: 0.323000

lr 2.100000e-07 reg 9.100000e+04 train accuracy: 0.324102 val accuracy: 0.336000

lr 3.100000e-07 reg 1.000000e+03 train accuracy: 0.375122 val accuracy: 0.379000

lr 3.100000e-07 reg 1.100000e+04 train accuracy: 0.364102 val accuracy: 0.380000

lr 3.100000e-07 reg 2.100000e+04 train accuracy: 0.357714 val accuracy: 0.366000

lr 3.100000e-07 reg 3.100000e+04 train accuracy: 0.347449 val accuracy: 0.353000

lr 3.100000e-07 reg 4.100000e+04 train accuracy: 0.340061 val accuracy: 0.359000

lr 3.100000e-07 reg 5.100000e+04 train accuracy: 0.345612 val accuracy: 0.356000
```

```
lr 3.100000e-07 reg 6.100000e+04 train accuracy: 0.321388 val accuracy: 0.336000
lr 3.100000e-07 reg 7.100000e+04 train accuracy: 0.337163 val accuracy: 0.336000
lr 3.100000e-07 reg 8.100000e+04 train accuracy: 0.326510 val accuracy: 0.346000
lr 3.100000e-07 reg 9.100000e+04 train accuracy: 0.321796 val accuracy: 0.331000
lr 4.100000e-07 reg 1.000000e+03 train accuracy: 0.392714 val accuracy: 0.374000
lr 4.100000e-07 reg 1.100000e+04 train accuracy: 0.361082 val accuracy: 0.375000
lr 4.100000e-07 reg 2.100000e+04 train accuracy: 0.343245 val accuracy: 0.325000
lr 4.100000e-07 reg 3.100000e+04 train accuracy: 0.344224 val accuracy: 0.353000
lr 4.100000e-07 reg 4.100000e+04 train accuracy: 0.316122 val accuracy: 0.332000
lr 4.100000e-07 reg 5.100000e+04 train accuracy: 0.337633 val accuracy: 0.349000
lr 4.100000e-07 reg 6.100000e+04 train accuracy: 0.319102 val accuracy: 0.330000
lr 4.100000e-07 reg 7.100000e+04 train accuracy: 0.324469 val accuracy: 0.338000
lr 4.100000e-07 reg 8.100000e+04 train accuracy: 0.285122 val accuracy: 0.291000
lr 4.100000e-07 reg 9.100000e+04 train accuracy: 0.298735 val accuracy: 0.306000
lr 5.100000e-07 reg 1.000000e+03 train accuracy: 0.366837 val accuracy: 0.377000
lr 5.100000e-07 reg 1.100000e+04 train accuracy: 0.352959 val accuracy: 0.376000
lr 5.100000e-07 reg 2.100000e+04 train accuracy: 0.340796 val accuracy: 0.355000
lr 5.100000e-07 reg 3.100000e+04 train accuracy: 0.322959 val accuracy: 0.342000
lr 5.100000e-07 reg 4.100000e+04 train accuracy: 0.297714 val accuracy: 0.326000
lr 5.100000e-07 reg 5.100000e+04 train accuracy: 0.323816 val accuracy: 0.335000
lr 5.100000e-07 reg 6.100000e+04 train accuracy: 0.328551 val accuracy: 0.341000
lr 5.100000e-07 reg 7.100000e+04 train accuracy: 0.299592 val accuracy: 0.306000
lr 5.100000e-07 reg 8.100000e+04 train accuracy: 0.307408 val accuracy: 0.311000
lr 5.100000e-07 reg 9.100000e+04 train accuracy: 0.300612 val accuracy: 0.288000
```

```
lr 6.100000e-07 reg 1.000000e+03 train accuracy: 0.359469 val accuracy: 0.351000
lr 6.100000e-07 reg 1.100000e+04 train accuracy: 0.349816 val accuracy: 0.362000
lr 6.100000e-07 reg 2.100000e+04 train accuracy: 0.314449 val accuracy: 0.323000
lr 6.100000e-07 reg 3.100000e+04 train accuracy: 0.312020 val accuracy: 0.323000
lr 6.100000e-07 reg 4.100000e+04 train accuracy: 0.307755 val accuracy: 0.319000
lr 6.100000e-07 reg 5.100000e+04 train accuracy: 0.289755 val accuracy: 0.289000
lr 6.100000e-07 reg 6.100000e+04 train accuracy: 0.311000 val accuracy: 0.316000
lr 6.100000e-07 reg 7.100000e+04 train accuracy: 0.308898 val accuracy: 0.334000
lr 6.100000e-07 reg 8.100000e+04 train accuracy: 0.276122 val accuracy: 0.281000
lr 6.100000e-07 reg 9.100000e+04 train accuracy: 0.276122 val accuracy: 0.283000
lr 7.100000e-07 reg 1.000000e+03 train accuracy: 0.369224 val accuracy: 0.348000
lr 7.100000e-07 reg 1.100000e+04 train accuracy: 0.320755 val accuracy: 0.314000
lr 7.100000e-07 reg 2.100000e+04 train accuracy: 0.320857 val accuracy: 0.312000
lr 7.100000e-07 reg 3.100000e+04 train accuracy: 0.301510 val accuracy: 0.308000
lr 7.100000e-07 reg 4.100000e+04 train accuracy: 0.316000 val accuracy: 0.323000
lr 7.100000e-07 reg 5.100000e+04 train accuracy: 0.300388 val accuracy: 0.313000
lr 7.100000e-07 reg 6.100000e+04 train accuracy: 0.301041 val accuracy: 0.305000
lr 7.100000e-07 reg 7.100000e+04 train accuracy: 0.289102 val accuracy: 0.303000
lr 7.100000e-07 reg 8.100000e+04 train accuracy: 0.241184 val accuracy: 0.239000
lr 7.100000e-07 reg 9.100000e+04 train accuracy: 0.268184 val accuracy: 0.281000
lr 8.100000e-07 reg 1.000000e+03 train accuracy: 0.383204 val accuracy: 0.372000
lr 8.100000e-07 reg 1.100000e+04 train accuracy: 0.335102 val accuracy: 0.350000
lr 8.100000e-07 reg 2.100000e+04 train accuracy: 0.316776 val accuracy: 0.328000
lr 8.100000e-07 reg 3.100000e+04 train accuracy: 0.315061 val accuracy: 0.319000
```

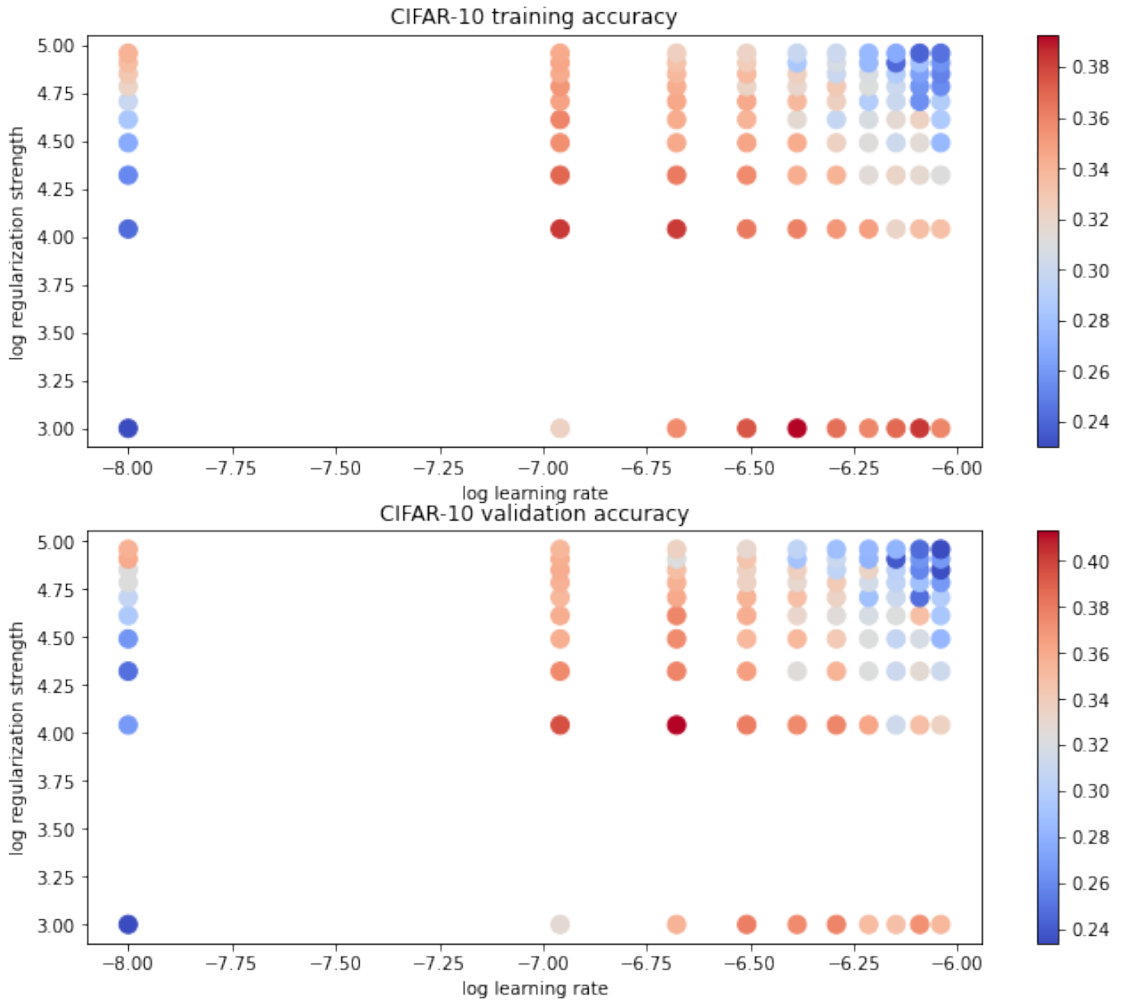
```
lr 8.100000e-07 reg 4.100000e+04 train accuracy: 0.322714 val accuracy: 0.349000

lr 8.100000e-07 reg 5.100000e+04 train accuracy: 0.255776 val accuracy: 0.248000

lr 8.100000e-07 reg 6.100000e+04 train accuracy: 0.258204 val accuracy: 0.289000

lr 8.100000e-07 reg 7.100000e+04 train accuracy: 0.263143 val accuracy: 0.258000

lr 8.100000e-07 reg 8.100000e+04 train accuracy: 0.280449 val accuracy: 0.265000

lr 8.100000e-07 reg 9.100000e+04 train accuracy: 0.239041 val accuracy: 0.247000

lr 9.100000e-07 reg 1.000000e+03 train accuracy: 0.359367 val accuracy: 0.353000

lr 9.100000e-07 reg 1.100000e+04 train accuracy: 0.333694 val accuracy: 0.336000

lr 9.100000e-07 reg 2.100000e+04 train accuracy: 0.311735 val accuracy: 0.313000

lr 9.100000e-07 reg 3.100000e+04 train accuracy: 0.276347 val accuracy: 0.283000

lr 9.100000e-07 reg 4.100000e+04 train accuracy: 0.286653 val accuracy: 0.293000

lr 9.100000e-07 reg 5.100000e+04 train accuracy: 0.288571 val accuracy: 0.299000

lr 9.100000e-07 reg 6.100000e+04 train accuracy: 0.254204 val accuracy: 0.267000

lr 9.100000e-07 reg 7.100000e+04 train accuracy: 0.250469 val accuracy: 0.235000

lr 9.100000e-07 reg 8.100000e+04 train accuracy: 0.257653 val accuracy: 0.267000

lr 9.100000e-07 reg 9.100000e+04 train accuracy: 0.244224 val accuracy: 0.234000

lr 1.000000e-08 reg 1.000000e+03 train accuracy: 0.230041 val accuracy: 0.234000
lr 1.000000e-08 reg 1.100000e+04 train accuracy: 0.241531 val accuracy: 0.268000
lr 1.000000e-08 reg 2.100000e+04 train accuracy: 0.252918 val accuracy: 0.249000
lr 1.000000e-08 reg 3.100000e+04 train accuracy: 0.268347 val accuracy: 0.266000
lr 1.000000e-08 reg 4.100000e+04 train accuracy: 0.284367 val accuracy: 0.296000
lr 1.000000e-08 reg 5.100000e+04 train accuracy: 0.299857 val accuracy: 0.308000
lr 1.000000e-08 reg 6.100000e+04 train accuracy: 0.321286 val accuracy: 0.324000
lr 1.000000e-08 reg 7.100000e+04 train accuracy: 0.329408 val accuracy: 0.324000
lr 1.000000e-08 reg 8.100000e+04 train accuracy: 0.335612 val accuracy: 0.360000
lr 1.000000e-08 reg 9.100000e+04 train accuracy: 0.340816 val accuracy: 0.357000
lr 1.100000e-07 reg 1.000000e+03 train accuracy: 0.323408 val accuracy: 0.329000
lr 1.100000e-07 reg 1.100000e+04 train accuracy: 0.383551 val accuracy: 0.396000
lr 1.100000e-07 reg 2.100000e+04 train accuracy: 0.369857 val accuracy: 0.375000
lr 1.100000e-07 reg 3.100000e+04 train accuracy: 0.356306 val accuracy: 0.358000
lr 1.100000e-07 reg 4.100000e+04 train accuracy: 0.360061 val accuracy: 0.359000
```

```
lr 1.100000e-07 reg 5.100000e+04 train accuracy: 0.347918 val accuracy: 0.353000
lr 1.100000e-07 reg 6.100000e+04 train accuracy: 0.353694 val accuracy: 0.357000
lr 1.100000e-07 reg 7.100000e+04 train accuracy: 0.344041 val accuracy: 0.359000
lr 1.100000e-07 reg 8.100000e+04 train accuracy: 0.346776 val accuracy: 0.358000
lr 1.100000e-07 reg 9.100000e+04 train accuracy: 0.344837 val accuracy: 0.354000
lr 2.100000e-07 reg 1.000000e+03 train accuracy: 0.357327 val accuracy: 0.356000
lr 2.100000e-07 reg 1.100000e+04 train accuracy: 0.382592 val accuracy: 0.413000
lr 2.100000e-07 reg 2.100000e+04 train accuracy: 0.362959 val accuracy: 0.377000
lr 2.100000e-07 reg 3.100000e+04 train accuracy: 0.345571 val accuracy: 0.374000
lr 2.100000e-07 reg 4.100000e+04 train accuracy: 0.344082 val accuracy: 0.376000
lr 2.100000e-07 reg 5.100000e+04 train accuracy: 0.345735 val accuracy: 0.361000
lr 2.100000e-07 reg 6.100000e+04 train accuracy: 0.343592 val accuracy: 0.357000
lr 2.100000e-07 reg 7.100000e+04 train accuracy: 0.338020 val accuracy: 0.350000
lr 2.100000e-07 reg 8.100000e+04 train accuracy: 0.333449 val accuracy: 0.323000
lr 2.100000e-07 reg 9.100000e+04 train accuracy: 0.324102 val accuracy: 0.336000
lr 3.100000e-07 reg 1.000000e+03 train accuracy: 0.375122 val accuracy: 0.379000
lr 3.100000e-07 reg 1.100000e+04 train accuracy: 0.364102 val accuracy: 0.380000
lr 3.100000e-07 reg 2.100000e+04 train accuracy: 0.357714 val accuracy: 0.366000
lr 3.100000e-07 reg 3.100000e+04 train accuracy: 0.347449 val accuracy: 0.353000
lr 3.100000e-07 reg 4.100000e+04 train accuracy: 0.340061 val accuracy: 0.359000
lr 3.100000e-07 reg 5.100000e+04 train accuracy: 0.345612 val accuracy: 0.356000
lr 3.100000e-07 reg 6.100000e+04 train accuracy: 0.321388 val accuracy: 0.336000
lr 3.100000e-07 reg 7.100000e+04 train accuracy: 0.337163 val accuracy: 0.336000
lr 3.100000e-07 reg 8.100000e+04 train accuracy: 0.326510 val accuracy: 0.346000
lr 3.100000e-07 reg 9.100000e+04 train accuracy: 0.321796 val accuracy: 0.331000
lr 4.100000e-07 reg 1.000000e+03 train accuracy: 0.392714 val accuracy: 0.374000
lr 4.100000e-07 reg 1.100000e+04 train accuracy: 0.361082 val accuracy: 0.375000
lr 4.100000e-07 reg 2.100000e+04 train accuracy: 0.343245 val accuracy: 0.325000
lr 4.100000e-07 reg 3.100000e+04 train accuracy: 0.344224 val accuracy: 0.353000
lr 4.100000e-07 reg 4.100000e+04 train accuracy: 0.316122 val accuracy: 0.332000
lr 4.100000e-07 reg 5.100000e+04 train accuracy: 0.337633 val accuracy: 0.349000
lr 4.100000e-07 reg 6.100000e+04 train accuracy: 0.319102 val accuracy: 0.330000
lr 4.100000e-07 reg 7.100000e+04 train accuracy: 0.324469 val accuracy: 0.338000
lr 4.100000e-07 reg 8.100000e+04 train accuracy: 0.285122 val accuracy: 0.291000
lr 4.100000e-07 reg 9.100000e+04 train accuracy: 0.298735 val accuracy: 0.306000
lr 5.100000e-07 reg 1.000000e+03 train accuracy: 0.366837 val accuracy: 0.377000
lr 5.100000e-07 reg 1.100000e+04 train accuracy: 0.352959 val accuracy: 0.376000
lr 5.100000e-07 reg 2.100000e+04 train accuracy: 0.340796 val accuracy: 0.355000
lr 5.100000e-07 reg 3.100000e+04 train accuracy: 0.322959 val accuracy: 0.342000
lr 5.100000e-07 reg 4.100000e+04 train accuracy: 0.297714 val accuracy: 0.326000
lr 5.100000e-07 reg 5.100000e+04 train accuracy: 0.323816 val accuracy: 0.335000
lr 5.100000e-07 reg 6.100000e+04 train accuracy: 0.328551 val accuracy: 0.341000
lr 5.100000e-07 reg 7.100000e+04 train accuracy: 0.299592 val accuracy: 0.306000
lr 5.100000e-07 reg 8.100000e+04 train accuracy: 0.307408 val accuracy: 0.311000
lr 5.100000e-07 reg 9.100000e+04 train accuracy: 0.300612 val accuracy: 0.288000
lr 6.100000e-07 reg 1.000000e+03 train accuracy: 0.359469 val accuracy: 0.351000
lr 6.100000e-07 reg 1.100000e+04 train accuracy: 0.349816 val accuracy: 0.362000
lr 6.100000e-07 reg 2.100000e+04 train accuracy: 0.314449 val accuracy: 0.323000
```

```
lr 6.100000e-07 reg 3.100000e+04 train accuracy: 0.312020 val accuracy: 0.323000
lr 6.100000e-07 reg 4.100000e+04 train accuracy: 0.307755 val accuracy: 0.319000
lr 6.100000e-07 reg 5.100000e+04 train accuracy: 0.289755 val accuracy: 0.289000
lr 6.100000e-07 reg 6.100000e+04 train accuracy: 0.311000 val accuracy: 0.316000
lr 6.100000e-07 reg 7.100000e+04 train accuracy: 0.308898 val accuracy: 0.334000
lr 6.100000e-07 reg 8.100000e+04 train accuracy: 0.276122 val accuracy: 0.281000
lr 6.100000e-07 reg 9.100000e+04 train accuracy: 0.276122 val accuracy: 0.283000
lr 7.100000e-07 reg 1.000000e+03 train accuracy: 0.369224 val accuracy: 0.348000
lr 7.100000e-07 reg 1.100000e+04 train accuracy: 0.320755 val accuracy: 0.314000
lr 7.100000e-07 reg 2.100000e+04 train accuracy: 0.320857 val accuracy: 0.312000
lr 7.100000e-07 reg 3.100000e+04 train accuracy: 0.301510 val accuracy: 0.308000
lr 7.100000e-07 reg 4.100000e+04 train accuracy: 0.316000 val accuracy: 0.323000
lr 7.100000e-07 reg 5.100000e+04 train accuracy: 0.300388 val accuracy: 0.313000
lr 7.100000e-07 reg 6.100000e+04 train accuracy: 0.301041 val accuracy: 0.305000
lr 7.100000e-07 reg 7.100000e+04 train accuracy: 0.289102 val accuracy: 0.303000
lr 7.100000e-07 reg 8.100000e+04 train accuracy: 0.241184 val accuracy: 0.239000
lr 7.100000e-07 reg 9.100000e+04 train accuracy: 0.268184 val accuracy: 0.281000
lr 8.100000e-07 reg 1.000000e+03 train accuracy: 0.383204 val accuracy: 0.372000
lr 8.100000e-07 reg 1.100000e+04 train accuracy: 0.335102 val accuracy: 0.350000
lr 8.100000e-07 reg 2.100000e+04 train accuracy: 0.316776 val accuracy: 0.328000
lr 8.100000e-07 reg 3.100000e+04 train accuracy: 0.315061 val accuracy: 0.319000
lr 8.100000e-07 reg 4.100000e+04 train accuracy: 0.322714 val accuracy: 0.349000
lr 8.100000e-07 reg 5.100000e+04 train accuracy: 0.255776 val accuracy: 0.248000
lr 8.100000e-07 reg 6.100000e+04 train accuracy: 0.258204 val accuracy: 0.289000
lr 8.100000e-07 reg 7.100000e+04 train accuracy: 0.263143 val accuracy: 0.258000
lr 8.100000e-07 reg 8.100000e+04 train accuracy: 0.280449 val accuracy: 0.265000
lr 8.100000e-07 reg 9.100000e+04 train accuracy: 0.239041 val accuracy: 0.247000
lr 9.100000e-07 reg 1.000000e+03 train accuracy: 0.359367 val accuracy: 0.353000
lr 9.100000e-07 reg 1.100000e+04 train accuracy: 0.333694 val accuracy: 0.336000
lr 9.100000e-07 reg 2.100000e+04 train accuracy: 0.311735 val accuracy: 0.313000
lr 9.100000e-07 reg 3.100000e+04 train accuracy: 0.276347 val accuracy: 0.283000
lr 9.100000e-07 reg 4.100000e+04 train accuracy: 0.286653 val accuracy: 0.293000
lr 9.100000e-07 reg 5.100000e+04 train accuracy: 0.288571 val accuracy: 0.299000
lr 9.100000e-07 reg 6.100000e+04 train accuracy: 0.254204 val accuracy: 0.267000
lr 9.100000e-07 reg 7.100000e+04 train accuracy: 0.250469 val accuracy: 0.235000
lr 9.100000e-07 reg 8.100000e+04 train accuracy: 0.257653 val accuracy: 0.267000
lr 9.100000e-07 reg 9.100000e+04 train accuracy: 0.244224 val accuracy: 0.234000
best validation accuracy achieved during cross-validation: 0.413000
```

[ ]:
```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()
```

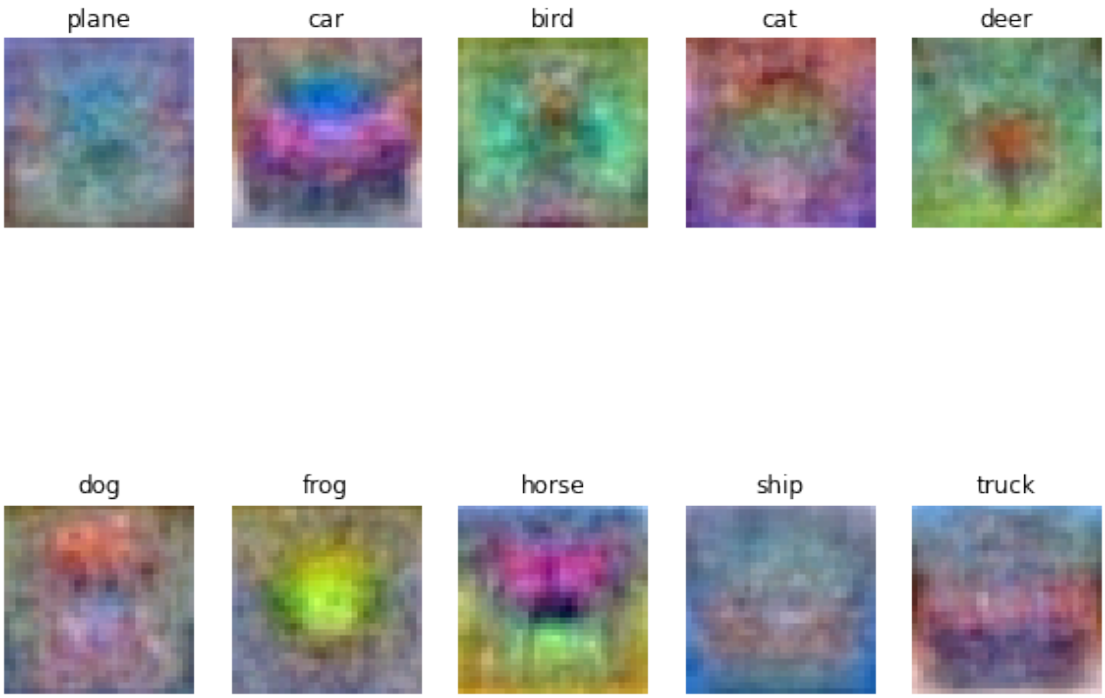


[ ]:
```
# Evaluate the best svm on test set
y_test_pred = best_svm.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
```

```
linear SVM on raw pixels final test set accuracy: 0.376000
```

[ ]:
```
# 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 the way they do.

Your Answer :

看起来像每个类别自己的大体轮廓特征。因为 loss 和 gradient 的配合驱使 weights 向更能描述自己类别特征的方向发展。

softmax

September 6, 2022

[2]:
```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive')

# TODO: 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 /content/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

[3]:
```
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
```

[4]:
```
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`.

[49]:
```
# 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: 2.352783
sanity check: 2.302585
```

Inline Question 1

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

Your Answer：

由

$$L_i = -\log \frac{\exp f_{y_i}}{\sum_j \exp f_j}$$

而初始时各 class 的概率基本相同, 故 $\exp f_j$ 对任意 j 都基本相同, 即

$$\frac{\exp f_{y_i}}{\sum_j \exp f_j} \approx \frac{1}{10}$$

有

$$L_i = -\log \frac{\exp f_{y_i}}{\sum_j \exp f_j} \approx -\log(0.1)$$

[50]:
```
# 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: 0.004721 analytic: 0.004721, relative error: 8.061647e-06
numerical: 0.843025 analytic: 0.843025, relative error: 4.338165e-08
numerical: 0.394924 analytic: 0.394924, relative error: 1.464735e-08
numerical: -4.641628 analytic: -4.641628, relative error: 4.740453e-09
numerical: 1.267248 analytic: 1.267248, relative error: 1.315392e-08
numerical: -0.239782 analytic: -0.239782, relative error: 1.094073e-07
numerical: 1.492718 analytic: 1.492718, relative error: 1.252059e-08
numerical: 1.457192 analytic: 1.457192, relative error: 3.074820e-08
numerical: 0.276637 analytic: 0.276637, relative error: 2.431547e-07
numerical: 1.462175 analytic: 1.462175, relative error: 7.040979e-09
numerical: 0.882241 analytic: 0.882241, relative error: 2.733032e-08
numerical: -1.082140 analytic: -1.082140, relative error: 1.038308e-08
numerical: 2.518560 analytic: 2.518560, relative error: 7.925521e-09
numerical: -0.378379 analytic: -0.378379, relative error: 4.323677e-08
numerical: 1.383907 analytic: 1.383907, relative error: 7.724030e-09
numerical: 0.992532 analytic: 0.992532, relative error: 4.003208e-08
numerical: -0.769438 analytic: -0.769438, relative error: 8.683382e-08
numerical: 1.038054 analytic: 1.038054, relative error: 7.125941e-08
numerical: 3.611365 analytic: 3.611365, relative error: 1.529367e-08
numerical: -0.320074 analytic: -0.320074, relative error: 4.609827e-08
```

[71]:
```
# 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: 2.352783e+00 computed in 0.144330s
margins.shape=(500, 10)
vectorized loss: 2.352783e+00 computed in 0.017722s
Loss difference: 0.000000
Gradient difference: 0.000000
```

[76]:
```
# 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 = [1e-7, 5e-7]
# regularization_strengths = [2.5e4, 5e4]

# my custom hyperparameters
learning_rates = np.arange(1e-8, 1e-6, 1e-7)
regularization_strengths = np.arange(1e3, 1e5, 1e4)

# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

for lr in learning_rates:
  for reg in regularization_strengths:
    new_softmax = Softmax()
    new_softmax.train(X_train, y_train, learning_rate=lr, reg=reg,
                      num_iters=1500, verbose=False)

    new_y_train_pred = new_softmax.predict(X_train)
    new_y_val_pred = new_softmax.predict(X_val)

    new_train_accuracy = np.mean(y_train == new_y_train_pred)
```

```
    new_val_accuracy = np.mean(y_val == new_y_val_pred)

    print('lr %e reg %e train accuracy: %f val accuracy: %f' % (
                lr, reg, new_train_accuracy, new_val_accuracy))

    results.update({(lr, reg):(new_train_accuracy, new_val_accuracy)})
    if new_val_accuracy > best_val:
      print('new best val')
      best_val = new_val_accuracy
      best_softmax = new_softmax

# *****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)
```

```
lr 1.000000e-08 reg 1.000000e+03 train accuracy: 0.152653 val accuracy: 0.141000
new best val
lr 1.000000e-08 reg 1.100000e+04 train accuracy: 0.175184 val accuracy: 0.178000
new best val
lr 1.000000e-08 reg 2.100000e+04 train accuracy: 0.166592 val accuracy: 0.178000
lr 1.000000e-08 reg 3.100000e+04 train accuracy: 0.175163 val accuracy: 0.167000
lr 1.000000e-08 reg 4.100000e+04 train accuracy: 0.203367 val accuracy: 0.202000
new best val
lr 1.000000e-08 reg 5.100000e+04 train accuracy: 0.208102 val accuracy: 0.228000
new best val
lr 1.000000e-08 reg 6.100000e+04 train accuracy: 0.200388 val accuracy: 0.199000
lr 1.000000e-08 reg 7.100000e+04 train accuracy: 0.234673 val accuracy: 0.233000
new best val
lr 1.000000e-08 reg 8.100000e+04 train accuracy: 0.245816 val accuracy: 0.250000
new best val
lr 1.000000e-08 reg 9.100000e+04 train accuracy: 0.261776 val accuracy: 0.267000
new best val
lr 1.100000e-07 reg 1.000000e+03 train accuracy: 0.266306 val accuracy: 0.275000
new best val
lr 1.100000e-07 reg 1.100000e+04 train accuracy: 0.352857 val accuracy: 0.373000
new best val
lr 1.100000e-07 reg 2.100000e+04 train accuracy: 0.334041 val accuracy: 0.346000
lr 1.100000e-07 reg 3.100000e+04 train accuracy: 0.321980 val accuracy: 0.333000
lr 1.100000e-07 reg 4.100000e+04 train accuracy: 0.318796 val accuracy: 0.327000
lr 1.100000e-07 reg 5.100000e+04 train accuracy: 0.311469 val accuracy: 0.332000
```

```
lr 1.100000e-07 reg 6.100000e+04 train accuracy: 0.307939 val accuracy: 0.314000
lr 1.100000e-07 reg 7.100000e+04 train accuracy: 0.295102 val accuracy: 0.307000
lr 1.100000e-07 reg 8.100000e+04 train accuracy: 0.290102 val accuracy: 0.302000
lr 1.100000e-07 reg 9.100000e+04 train accuracy: 0.289082 val accuracy: 0.305000
lr 2.100000e-07 reg 1.000000e+03 train accuracy: 0.320918 val accuracy: 0.316000
lr 2.100000e-07 reg 1.100000e+04 train accuracy: 0.357735 val accuracy: 0.369000
lr 2.100000e-07 reg 2.100000e+04 train accuracy: 0.334061 val accuracy: 0.346000
lr 2.100000e-07 reg 3.100000e+04 train accuracy: 0.315367 val accuracy: 0.329000
lr 2.100000e-07 reg 4.100000e+04 train accuracy: 0.310633 val accuracy: 0.322000
lr 2.100000e-07 reg 5.100000e+04 train accuracy: 0.307020 val accuracy: 0.319000
lr 2.100000e-07 reg 6.100000e+04 train accuracy: 0.305959 val accuracy: 0.310000
lr 2.100000e-07 reg 7.100000e+04 train accuracy: 0.289939 val accuracy: 0.309000
lr 2.100000e-07 reg 8.100000e+04 train accuracy: 0.293959 val accuracy: 0.312000
lr 2.100000e-07 reg 9.100000e+04 train accuracy: 0.291245 val accuracy: 0.297000
lr 3.100000e-07 reg 1.000000e+03 train accuracy: 0.352612 val accuracy: 0.368000
lr 3.100000e-07 reg 1.100000e+04 train accuracy: 0.351878 val accuracy: 0.368000
lr 3.100000e-07 reg 2.100000e+04 train accuracy: 0.336000 val accuracy: 0.344000
lr 3.100000e-07 reg 3.100000e+04 train accuracy: 0.327286 val accuracy: 0.348000
lr 3.100000e-07 reg 4.100000e+04 train accuracy: 0.319061 val accuracy: 0.330000
lr 3.100000e-07 reg 5.100000e+04 train accuracy: 0.302796 val accuracy: 0.310000
lr 3.100000e-07 reg 6.100000e+04 train accuracy: 0.304143 val accuracy: 0.318000
lr 3.100000e-07 reg 7.100000e+04 train accuracy: 0.294041 val accuracy: 0.310000
lr 3.100000e-07 reg 8.100000e+04 train accuracy: 0.273469 val accuracy: 0.287000
lr 3.100000e-07 reg 9.100000e+04 train accuracy: 0.276694 val accuracy: 0.284000
lr 4.100000e-07 reg 1.000000e+03 train accuracy: 0.376204 val accuracy: 0.378000
new best val
lr 4.100000e-07 reg 1.100000e+04 train accuracy: 0.348633 val accuracy: 0.355000
lr 4.100000e-07 reg 2.100000e+04 train accuracy: 0.334408 val accuracy: 0.354000
lr 4.100000e-07 reg 3.100000e+04 train accuracy: 0.305796 val accuracy: 0.329000
lr 4.100000e-07 reg 4.100000e+04 train accuracy: 0.317653 val accuracy: 0.333000
lr 4.100000e-07 reg 5.100000e+04 train accuracy: 0.315163 val accuracy: 0.322000
lr 4.100000e-07 reg 6.100000e+04 train accuracy: 0.283878 val accuracy: 0.295000
lr 4.100000e-07 reg 7.100000e+04 train accuracy: 0.288102 val accuracy: 0.298000
lr 4.100000e-07 reg 8.100000e+04 train accuracy: 0.286286 val accuracy: 0.299000
lr 4.100000e-07 reg 9.100000e+04 train accuracy: 0.291612 val accuracy: 0.307000
lr 5.100000e-07 reg 1.000000e+03 train accuracy: 0.388837 val accuracy: 0.396000
new best val
lr 5.100000e-07 reg 1.100000e+04 train accuracy: 0.348776 val accuracy: 0.359000
lr 5.100000e-07 reg 2.100000e+04 train accuracy: 0.330959 val accuracy: 0.353000
lr 5.100000e-07 reg 3.100000e+04 train accuracy: 0.314857 val accuracy: 0.338000
lr 5.100000e-07 reg 4.100000e+04 train accuracy: 0.313592 val accuracy: 0.333000
lr 5.100000e-07 reg 5.100000e+04 train accuracy: 0.299143 val accuracy: 0.316000
lr 5.100000e-07 reg 6.100000e+04 train accuracy: 0.300082 val accuracy: 0.318000
lr 5.100000e-07 reg 7.100000e+04 train accuracy: 0.291531 val accuracy: 0.295000
lr 5.100000e-07 reg 8.100000e+04 train accuracy: 0.279306 val accuracy: 0.278000
lr 5.100000e-07 reg 9.100000e+04 train accuracy: 0.283755 val accuracy: 0.293000
lr 6.100000e-07 reg 1.000000e+03 train accuracy: 0.394122 val accuracy: 0.388000
lr 6.100000e-07 reg 1.100000e+04 train accuracy: 0.346939 val accuracy: 0.360000
```

```
lr 6.100000e-07 reg 2.100000e+04 train accuracy: 0.333184 val accuracy: 0.343000
lr 6.100000e-07 reg 3.100000e+04 train accuracy: 0.315469 val accuracy: 0.331000
lr 6.100000e-07 reg 4.100000e+04 train accuracy: 0.297735 val accuracy: 0.317000
lr 6.100000e-07 reg 5.100000e+04 train accuracy: 0.307694 val accuracy: 0.325000
lr 6.100000e-07 reg 6.100000e+04 train accuracy: 0.299327 val accuracy: 0.301000
lr 6.100000e-07 reg 7.100000e+04 train accuracy: 0.292980 val accuracy: 0.312000
lr 6.100000e-07 reg 8.100000e+04 train accuracy: 0.285245 val accuracy: 0.300000
lr 6.100000e-07 reg 9.100000e+04 train accuracy: 0.275776 val accuracy: 0.291000
lr 7.100000e-07 reg 1.000000e+03 train accuracy: 0.396612 val accuracy: 0.400000
new best val
lr 7.100000e-07 reg 1.100000e+04 train accuracy: 0.339980 val accuracy: 0.354000
lr 7.100000e-07 reg 2.100000e+04 train accuracy: 0.324061 val accuracy: 0.342000
lr 7.100000e-07 reg 3.100000e+04 train accuracy: 0.322735 val accuracy: 0.331000
lr 7.100000e-07 reg 4.100000e+04 train accuracy: 0.313755 val accuracy: 0.317000
lr 7.100000e-07 reg 5.100000e+04 train accuracy: 0.285653 val accuracy: 0.291000
lr 7.100000e-07 reg 6.100000e+04 train accuracy: 0.298918 val accuracy: 0.320000
lr 7.100000e-07 reg 7.100000e+04 train accuracy: 0.292082 val accuracy: 0.295000
lr 7.100000e-07 reg 8.100000e+04 train accuracy: 0.279898 val accuracy: 0.288000
lr 7.100000e-07 reg 9.100000e+04 train accuracy: 0.293020 val accuracy: 0.295000
lr 8.100000e-07 reg 1.000000e+03 train accuracy: 0.395163 val accuracy: 0.410000
new best val
lr 8.100000e-07 reg 1.100000e+04 train accuracy: 0.346571 val accuracy: 0.359000
lr 8.100000e-07 reg 2.100000e+04 train accuracy: 0.334122 val accuracy: 0.343000
lr 8.100000e-07 reg 3.100000e+04 train accuracy: 0.319551 val accuracy: 0.334000
lr 8.100000e-07 reg 4.100000e+04 train accuracy: 0.299673 val accuracy: 0.318000
lr 8.100000e-07 reg 5.100000e+04 train accuracy: 0.287878 val accuracy: 0.292000
lr 8.100000e-07 reg 6.100000e+04 train accuracy: 0.286224 val accuracy: 0.302000
lr 8.100000e-07 reg 7.100000e+04 train accuracy: 0.271143 val accuracy: 0.279000
lr 8.100000e-07 reg 8.100000e+04 train accuracy: 0.288082 val accuracy: 0.301000
lr 8.100000e-07 reg 9.100000e+04 train accuracy: 0.268102 val accuracy: 0.266000
lr 9.100000e-07 reg 1.000000e+03 train accuracy: 0.402388 val accuracy: 0.398000
lr 9.100000e-07 reg 1.100000e+04 train accuracy: 0.352041 val accuracy: 0.376000
lr 9.100000e-07 reg 2.100000e+04 train accuracy: 0.333367 val accuracy: 0.343000
lr 9.100000e-07 reg 3.100000e+04 train accuracy: 0.314000 val accuracy: 0.339000
lr 9.100000e-07 reg 4.100000e+04 train accuracy: 0.304510 val accuracy: 0.319000
lr 9.100000e-07 reg 5.100000e+04 train accuracy: 0.281694 val accuracy: 0.293000
lr 9.100000e-07 reg 6.100000e+04 train accuracy: 0.305490 val accuracy: 0.316000
lr 9.100000e-07 reg 7.100000e+04 train accuracy: 0.286143 val accuracy: 0.305000
lr 9.100000e-07 reg 8.100000e+04 train accuracy: 0.269633 val accuracy: 0.288000
lr 9.100000e-07 reg 9.100000e+04 train accuracy: 0.265082 val accuracy: 0.269000
lr 1.000000e-08 reg 1.000000e+03 train accuracy: 0.152653 val accuracy: 0.141000
lr 1.000000e-08 reg 1.100000e+04 train accuracy: 0.175184 val accuracy: 0.178000
lr 1.000000e-08 reg 2.100000e+04 train accuracy: 0.166592 val accuracy: 0.178000
lr 1.000000e-08 reg 3.100000e+04 train accuracy: 0.175163 val accuracy: 0.167000
lr 1.000000e-08 reg 4.100000e+04 train accuracy: 0.203367 val accuracy: 0.202000
lr 1.000000e-08 reg 5.100000e+04 train accuracy: 0.208102 val accuracy: 0.228000
lr 1.000000e-08 reg 6.100000e+04 train accuracy: 0.200388 val accuracy: 0.199000
lr 1.000000e-08 reg 7.100000e+04 train accuracy: 0.234673 val accuracy: 0.233000
```

```
lr 1.000000e-08 reg 8.100000e+04 train accuracy: 0.245816 val accuracy: 0.250000
lr 1.000000e-08 reg 9.100000e+04 train accuracy: 0.261776 val accuracy: 0.267000
lr 1.100000e-07 reg 1.000000e+03 train accuracy: 0.266306 val accuracy: 0.275000
lr 1.100000e-07 reg 1.100000e+04 train accuracy: 0.352857 val accuracy: 0.373000
lr 1.100000e-07 reg 2.100000e+04 train accuracy: 0.334041 val accuracy: 0.346000
lr 1.100000e-07 reg 3.100000e+04 train accuracy: 0.321980 val accuracy: 0.333000
lr 1.100000e-07 reg 4.100000e+04 train accuracy: 0.318796 val accuracy: 0.327000
lr 1.100000e-07 reg 5.100000e+04 train accuracy: 0.311469 val accuracy: 0.332000
lr 1.100000e-07 reg 6.100000e+04 train accuracy: 0.307939 val accuracy: 0.314000
lr 1.100000e-07 reg 7.100000e+04 train accuracy: 0.295102 val accuracy: 0.307000
lr 1.100000e-07 reg 8.100000e+04 train accuracy: 0.290102 val accuracy: 0.302000
lr 1.100000e-07 reg 9.100000e+04 train accuracy: 0.289082 val accuracy: 0.305000
lr 2.100000e-07 reg 1.000000e+03 train accuracy: 0.320918 val accuracy: 0.316000
lr 2.100000e-07 reg 1.100000e+04 train accuracy: 0.357735 val accuracy: 0.369000
lr 2.100000e-07 reg 2.100000e+04 train accuracy: 0.334061 val accuracy: 0.346000
lr 2.100000e-07 reg 3.100000e+04 train accuracy: 0.315367 val accuracy: 0.329000
lr 2.100000e-07 reg 4.100000e+04 train accuracy: 0.310633 val accuracy: 0.322000
lr 2.100000e-07 reg 5.100000e+04 train accuracy: 0.307020 val accuracy: 0.319000
lr 2.100000e-07 reg 6.100000e+04 train accuracy: 0.305959 val accuracy: 0.310000
lr 2.100000e-07 reg 7.100000e+04 train accuracy: 0.289939 val accuracy: 0.309000
lr 2.100000e-07 reg 8.100000e+04 train accuracy: 0.293959 val accuracy: 0.312000
lr 2.100000e-07 reg 9.100000e+04 train accuracy: 0.291245 val accuracy: 0.297000
lr 3.100000e-07 reg 1.000000e+03 train accuracy: 0.352612 val accuracy: 0.368000
lr 3.100000e-07 reg 1.100000e+04 train accuracy: 0.351878 val accuracy: 0.368000
lr 3.100000e-07 reg 2.100000e+04 train accuracy: 0.336000 val accuracy: 0.344000
lr 3.100000e-07 reg 3.100000e+04 train accuracy: 0.327286 val accuracy: 0.348000
lr 3.100000e-07 reg 4.100000e+04 train accuracy: 0.319061 val accuracy: 0.330000
lr 3.100000e-07 reg 5.100000e+04 train accuracy: 0.302796 val accuracy: 0.310000
lr 3.100000e-07 reg 6.100000e+04 train accuracy: 0.304143 val accuracy: 0.318000
lr 3.100000e-07 reg 7.100000e+04 train accuracy: 0.294041 val accuracy: 0.310000
lr 3.100000e-07 reg 8.100000e+04 train accuracy: 0.273469 val accuracy: 0.287000
lr 3.100000e-07 reg 9.100000e+04 train accuracy: 0.276694 val accuracy: 0.284000
lr 4.100000e-07 reg 1.000000e+03 train accuracy: 0.376204 val accuracy: 0.378000
lr 4.100000e-07 reg 1.100000e+04 train accuracy: 0.348633 val accuracy: 0.355000
lr 4.100000e-07 reg 2.100000e+04 train accuracy: 0.334408 val accuracy: 0.354000
lr 4.100000e-07 reg 3.100000e+04 train accuracy: 0.305796 val accuracy: 0.329000
lr 4.100000e-07 reg 4.100000e+04 train accuracy: 0.317653 val accuracy: 0.333000
lr 4.100000e-07 reg 5.100000e+04 train accuracy: 0.315163 val accuracy: 0.322000
lr 4.100000e-07 reg 6.100000e+04 train accuracy: 0.283878 val accuracy: 0.295000
lr 4.100000e-07 reg 7.100000e+04 train accuracy: 0.288102 val accuracy: 0.298000
lr 4.100000e-07 reg 8.100000e+04 train accuracy: 0.286286 val accuracy: 0.299000
lr 4.100000e-07 reg 9.100000e+04 train accuracy: 0.291612 val accuracy: 0.307000
lr 5.100000e-07 reg 1.000000e+03 train accuracy: 0.388837 val accuracy: 0.396000
lr 5.100000e-07 reg 1.100000e+04 train accuracy: 0.348776 val accuracy: 0.359000
lr 5.100000e-07 reg 2.100000e+04 train accuracy: 0.330959 val accuracy: 0.353000
lr 5.100000e-07 reg 3.100000e+04 train accuracy: 0.314857 val accuracy: 0.338000
lr 5.100000e-07 reg 4.100000e+04 train accuracy: 0.313592 val accuracy: 0.333000
lr 5.100000e-07 reg 5.100000e+04 train accuracy: 0.299143 val accuracy: 0.316000
```

```
lr 5.100000e-07 reg 6.100000e+04 train accuracy: 0.300082 val accuracy: 0.318000
lr 5.100000e-07 reg 7.100000e+04 train accuracy: 0.291531 val accuracy: 0.295000
lr 5.100000e-07 reg 8.100000e+04 train accuracy: 0.279306 val accuracy: 0.278000
lr 5.100000e-07 reg 9.100000e+04 train accuracy: 0.283755 val accuracy: 0.293000
lr 6.100000e-07 reg 1.000000e+03 train accuracy: 0.394122 val accuracy: 0.388000
lr 6.100000e-07 reg 1.100000e+04 train accuracy: 0.346939 val accuracy: 0.360000
lr 6.100000e-07 reg 2.100000e+04 train accuracy: 0.333184 val accuracy: 0.343000
lr 6.100000e-07 reg 3.100000e+04 train accuracy: 0.315469 val accuracy: 0.331000
lr 6.100000e-07 reg 4.100000e+04 train accuracy: 0.297735 val accuracy: 0.317000
lr 6.100000e-07 reg 5.100000e+04 train accuracy: 0.307694 val accuracy: 0.325000
lr 6.100000e-07 reg 6.100000e+04 train accuracy: 0.299327 val accuracy: 0.301000
lr 6.100000e-07 reg 7.100000e+04 train accuracy: 0.292980 val accuracy: 0.312000
lr 6.100000e-07 reg 8.100000e+04 train accuracy: 0.285245 val accuracy: 0.300000
lr 6.100000e-07 reg 9.100000e+04 train accuracy: 0.275776 val accuracy: 0.291000
lr 7.100000e-07 reg 1.000000e+03 train accuracy: 0.396612 val accuracy: 0.400000
lr 7.100000e-07 reg 1.100000e+04 train accuracy: 0.339980 val accuracy: 0.354000
lr 7.100000e-07 reg 2.100000e+04 train accuracy: 0.324061 val accuracy: 0.342000
lr 7.100000e-07 reg 3.100000e+04 train accuracy: 0.322735 val accuracy: 0.331000
lr 7.100000e-07 reg 4.100000e+04 train accuracy: 0.313755 val accuracy: 0.317000
lr 7.100000e-07 reg 5.100000e+04 train accuracy: 0.285653 val accuracy: 0.291000
lr 7.100000e-07 reg 6.100000e+04 train accuracy: 0.298918 val accuracy: 0.320000
lr 7.100000e-07 reg 7.100000e+04 train accuracy: 0.292082 val accuracy: 0.295000
lr 7.100000e-07 reg 8.100000e+04 train accuracy: 0.279898 val accuracy: 0.288000
lr 7.100000e-07 reg 9.100000e+04 train accuracy: 0.293020 val accuracy: 0.295000
lr 8.100000e-07 reg 1.000000e+03 train accuracy: 0.395163 val accuracy: 0.410000
lr 8.100000e-07 reg 1.100000e+04 train accuracy: 0.346571 val accuracy: 0.359000
lr 8.100000e-07 reg 2.100000e+04 train accuracy: 0.334122 val accuracy: 0.343000
lr 8.100000e-07 reg 3.100000e+04 train accuracy: 0.319551 val accuracy: 0.334000
lr 8.100000e-07 reg 4.100000e+04 train accuracy: 0.299673 val accuracy: 0.318000
lr 8.100000e-07 reg 5.100000e+04 train accuracy: 0.287878 val accuracy: 0.292000
lr 8.100000e-07 reg 6.100000e+04 train accuracy: 0.286224 val accuracy: 0.302000
lr 8.100000e-07 reg 7.100000e+04 train accuracy: 0.271143 val accuracy: 0.279000
lr 8.100000e-07 reg 8.100000e+04 train accuracy: 0.288082 val accuracy: 0.301000
lr 8.100000e-07 reg 9.100000e+04 train accuracy: 0.268102 val accuracy: 0.266000
lr 9.100000e-07 reg 1.000000e+03 train accuracy: 0.402388 val accuracy: 0.398000
lr 9.100000e-07 reg 1.100000e+04 train accuracy: 0.352041 val accuracy: 0.376000
lr 9.100000e-07 reg 2.100000e+04 train accuracy: 0.333367 val accuracy: 0.343000
lr 9.100000e-07 reg 3.100000e+04 train accuracy: 0.314000 val accuracy: 0.339000
lr 9.100000e-07 reg 4.100000e+04 train accuracy: 0.304510 val accuracy: 0.319000
lr 9.100000e-07 reg 5.100000e+04 train accuracy: 0.281694 val accuracy: 0.293000
lr 9.100000e-07 reg 6.100000e+04 train accuracy: 0.305490 val accuracy: 0.316000
lr 9.100000e-07 reg 7.100000e+04 train accuracy: 0.286143 val accuracy: 0.305000
lr 9.100000e-07 reg 8.100000e+04 train accuracy: 0.269633 val accuracy: 0.288000
lr 9.100000e-07 reg 9.100000e+04 train accuracy: 0.265082 val accuracy: 0.269000
best validation accuracy achieved during cross-validation: 0.410000
```

[77]:
```
# 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: 0.376000
```

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：

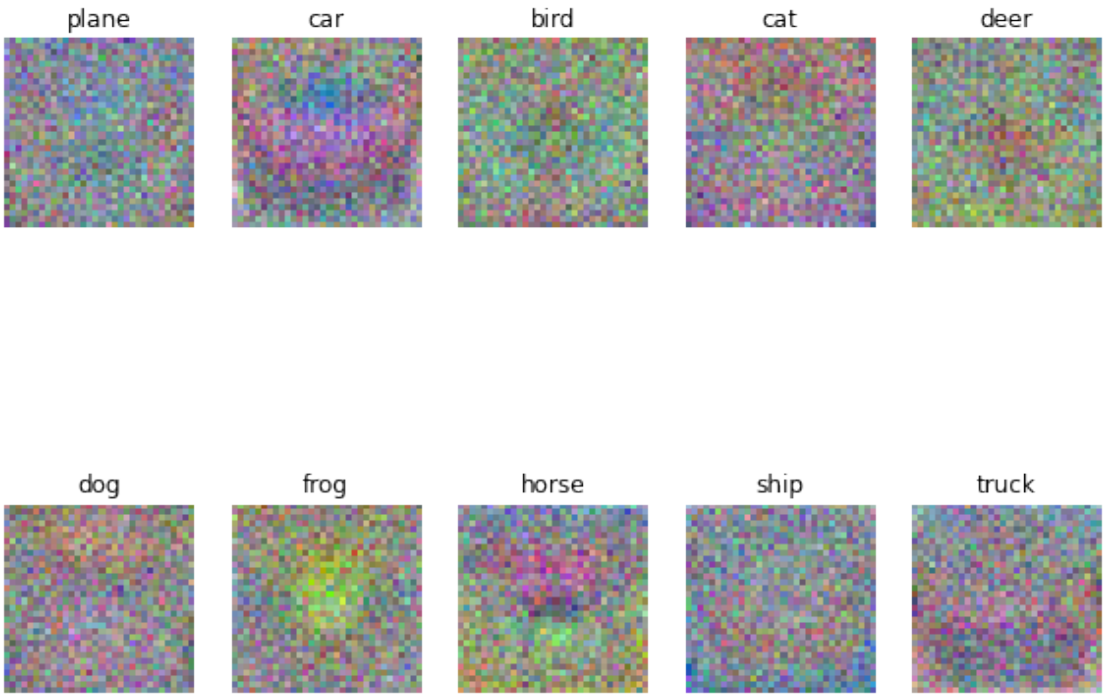
True

YourExplanation：

对于 SVM, 要让 loss 函数 $\sum_i \sum_{j \neq y_i} \max(0, S_j - S_{y_i} + 1)$ 不变, 只需保证新加入的数据点的 $S_{y_i} - S_j = 1$ 恒成立即可。

然而对于 Softmax, 由于采用了概率方式表示 loss 函数, 增加数据点必然会改变 loss 函数的分母, 导致 loss 函数的值一定变化。

[78]:
```
# 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

September 6, 2022

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

# TODO: 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 /content/drive/My\ Drive/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

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

[ ]:
```
# 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
```

[ ]:
```
# 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:

[ ]:
```
# 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.

[ ]:
```
# 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:

[ ]:
```
# 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:

[ ]:
```
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. 在 $x \to \infty$ 的时候, gradient 会趋近于 0.
2. 因为 Relu 形式为 $\max(0, x)$, 当 $x < 0$ 的时候就会出现 gradient 为 0 的情况.
3. $x < 0$ 的时候会趋近于 0.

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:

[ ]:
```
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:

[ ]:
```
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 svm_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:  9.341692129091487e-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.

[ ]:
```
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: 3.97e-08
W2 relative error: 8.97e-10
```

```
b1 relative error: 2.26e-08
b2 relative error: 9.39e-10
Running numeric gradient check with reg =  0.7
W1 relative error: 3.12e-07
W2 relative error: 7.98e-08
b1 relative error: 2.80e-08
b2 relative error: 1.97e-09
```

9 Solver

Open the file `cs231n/solver.py` and read through it to familiarize yourself with the API. You also need to implement 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.

[ ]:
```
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
model = TwoLayerNet(input_size, hidden_size, num_classes)
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': 1e-4},
                lr_decay=0.95, num_epochs=5,
                batch_size=200, print_every=100)
solver.train()

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
################################################################################
#                              END OF YOUR CODE                                #
################################################################################
```

```
(Iteration 1 / 1225) loss: 2.303854
(Epoch 0 / 5) train acc: 0.118000; val_acc: 0.128000
(Iteration 101 / 1225) loss: 2.270676
(Iteration 201 / 1225) loss: 2.155035
(Epoch 1 / 5) train acc: 0.218000; val_acc: 0.253000
(Iteration 301 / 1225) loss: 2.049379
(Iteration 401 / 1225) loss: 2.028418
(Epoch 2 / 5) train acc: 0.286000; val_acc: 0.279000
(Iteration 501 / 1225) loss: 1.921624
(Iteration 601 / 1225) loss: 1.939255
```

```
(Iteration 701 / 1225) loss: 1.832826
(Epoch 3 / 5) train acc: 0.354000; val_acc: 0.315000
(Iteration 801 / 1225) loss: 1.887060
(Iteration 901 / 1225) loss: 1.768584
(Epoch 4 / 5) train acc: 0.355000; val_acc: 0.346000
(Iteration 1001 / 1225) loss: 1.804191
(Iteration 1101 / 1225) loss: 1.725366
(Iteration 1201 / 1225) loss: 1.833903
(Epoch 5 / 5) train acc: 0.364000; val_acc: 0.371000
```

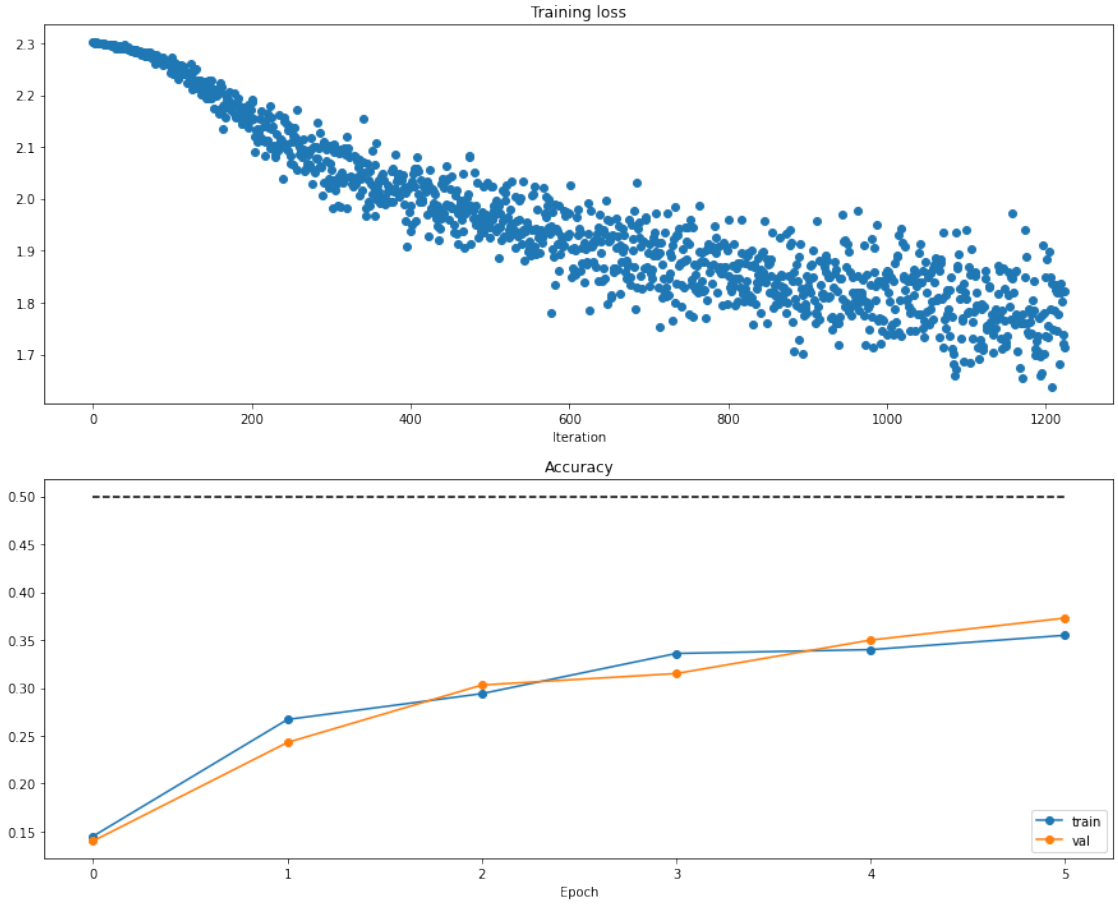
10 Debug the training

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.

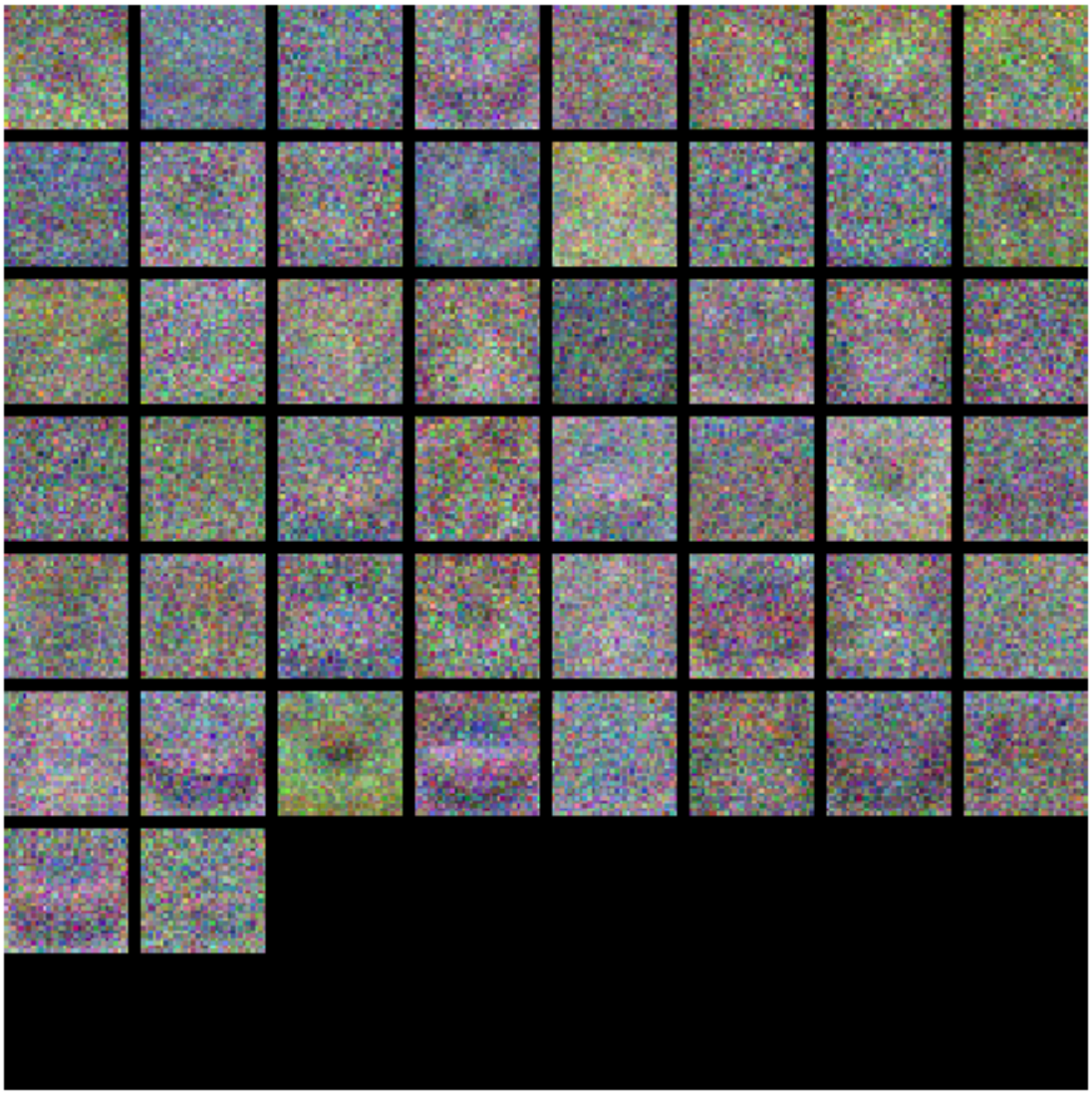
One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

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.

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



[ ]:
```python
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(3, 32, 32, -1).transpose(3, 1, 2, 0)
    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.).

[ ]:
```
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)*****

best_acc = -1
model = None
solver = None

input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10

num_epoches = 25

model = TwoLayerNet(input_size, hidden_size, num_classes)
```

```
solver = Solver(model, data,
                update_rule='sgd',
                optim_config={'learning_rate': 5e-4},
                lr_decay=0.95, num_epochs=30,
                batch_size=200, print_every=100)
solver.train()

# y_val_pred = np.argmax(model.loss(data['X_val']), axis=1)
# accuracy = (y_val_pred == data['y_val']).mean()
accuracy = solver.best_val_acc
if accuracy > best_acc:
  best_model = model
  best_acc = accuracy
  print('new best accuracy: ', accuracy)

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
################################################################################
#                                END OF YOUR CODE                              #
################################################################################
```

```
(Iteration 1 / 7350) loss: 2.300190
(Epoch 0 / 30) train acc: 0.145000; val_acc: 0.165000
(Iteration 101 / 7350) loss: 2.015234
(Iteration 201 / 7350) loss: 1.790591
(Epoch 1 / 30) train acc: 0.390000; val_acc: 0.380000
(Iteration 301 / 7350) loss: 1.758738
(Iteration 401 / 7350) loss: 1.666508
(Epoch 2 / 30) train acc: 0.431000; val_acc: 0.438000
(Iteration 501 / 7350) loss: 1.620063
(Iteration 601 / 7350) loss: 1.648616
(Iteration 701 / 7350) loss: 1.682793
(Epoch 3 / 30) train acc: 0.480000; val_acc: 0.454000
(Iteration 801 / 7350) loss: 1.548954
(Iteration 901 / 7350) loss: 1.393497
(Epoch 4 / 30) train acc: 0.424000; val_acc: 0.473000
(Iteration 1001 / 7350) loss: 1.566226
(Iteration 1101 / 7350) loss: 1.517484
(Iteration 1201 / 7350) loss: 1.500529
(Epoch 5 / 30) train acc: 0.466000; val_acc: 0.469000
(Iteration 1301 / 7350) loss: 1.296426
(Iteration 1401 / 7350) loss: 1.359932
(Epoch 6 / 30) train acc: 0.491000; val_acc: 0.476000
(Iteration 1501 / 7350) loss: 1.484785
(Iteration 1601 / 7350) loss: 1.516682
(Iteration 1701 / 7350) loss: 1.460739
(Epoch 7 / 30) train acc: 0.481000; val_acc: 0.489000
```

```
(Iteration 1801 / 7350) loss: 1.543901
(Iteration 1901 / 7350) loss: 1.464250
(Epoch 8 / 30) train acc: 0.497000; val_acc: 0.485000
(Iteration 2001 / 7350) loss: 1.312434
(Iteration 2101 / 7350) loss: 1.412041
(Iteration 2201 / 7350) loss: 1.529823
(Epoch 9 / 30) train acc: 0.532000; val_acc: 0.494000
(Iteration 2301 / 7350) loss: 1.226684
(Iteration 2401 / 7350) loss: 1.323381
(Epoch 10 / 30) train acc: 0.535000; val_acc: 0.497000
(Iteration 2501 / 7350) loss: 1.352839
(Iteration 2601 / 7350) loss: 1.337114
(Epoch 11 / 30) train acc: 0.531000; val_acc: 0.501000
(Iteration 2701 / 7350) loss: 1.267915
(Iteration 2801 / 7350) loss: 1.367660
(Iteration 2901 / 7350) loss: 1.356661
(Epoch 12 / 30) train acc: 0.549000; val_acc: 0.521000
(Iteration 3001 / 7350) loss: 1.306190
(Iteration 3101 / 7350) loss: 1.378280
(Epoch 13 / 30) train acc: 0.511000; val_acc: 0.518000
(Iteration 3201 / 7350) loss: 1.388568
(Iteration 3301 / 7350) loss: 1.323884
(Iteration 3401 / 7350) loss: 1.354564
(Epoch 14 / 30) train acc: 0.547000; val_acc: 0.522000
(Iteration 3501 / 7350) loss: 1.233557
(Iteration 3601 / 7350) loss: 1.211154
(Epoch 15 / 30) train acc: 0.575000; val_acc: 0.515000
(Iteration 3701 / 7350) loss: 1.273599
(Iteration 3801 / 7350) loss: 1.163792
(Iteration 3901 / 7350) loss: 1.298349
(Epoch 16 / 30) train acc: 0.543000; val_acc: 0.513000
(Iteration 4001 / 7350) loss: 1.163151
(Iteration 4101 / 7350) loss: 1.333338
(Epoch 17 / 30) train acc: 0.523000; val_acc: 0.522000
(Iteration 4201 / 7350) loss: 1.220220
(Iteration 4301 / 7350) loss: 1.396700
(Iteration 4401 / 7350) loss: 1.153267
(Epoch 18 / 30) train acc: 0.555000; val_acc: 0.515000
(Iteration 4501 / 7350) loss: 1.140811
(Iteration 4601 / 7350) loss: 1.295988
(Epoch 19 / 30) train acc: 0.533000; val_acc: 0.511000
(Iteration 4701 / 7350) loss: 1.244502
(Iteration 4801 / 7350) loss: 1.220347
(Epoch 20 / 30) train acc: 0.559000; val_acc: 0.523000
(Iteration 4901 / 7350) loss: 1.241858
(Iteration 5001 / 7350) loss: 1.200796
(Iteration 5101 / 7350) loss: 1.394251
(Epoch 21 / 30) train acc: 0.577000; val_acc: 0.525000
```

```
(Iteration 5201 / 7350) loss: 1.206179
(Iteration 5301 / 7350) loss: 1.098918
(Epoch 22 / 30) train acc: 0.565000; val_acc: 0.520000
(Iteration 5401 / 7350) loss: 1.171365
(Iteration 5501 / 7350) loss: 1.228942
(Iteration 5601 / 7350) loss: 1.163735
(Epoch 23 / 30) train acc: 0.555000; val_acc: 0.528000
(Iteration 5701 / 7350) loss: 1.308055
(Iteration 5801 / 7350) loss: 1.228080
(Epoch 24 / 30) train acc: 0.589000; val_acc: 0.519000
(Iteration 5901 / 7350) loss: 1.227241
(Iteration 6001 / 7350) loss: 1.333534
(Iteration 6101 / 7350) loss: 1.144016
(Epoch 25 / 30) train acc: 0.556000; val_acc: 0.527000
(Iteration 6201 / 7350) loss: 1.231183
(Iteration 6301 / 7350) loss: 1.239231
(Epoch 26 / 30) train acc: 0.573000; val_acc: 0.521000
(Iteration 6401 / 7350) loss: 1.195566
(Iteration 6501 / 7350) loss: 1.243926
(Iteration 6601 / 7350) loss: 1.287579
(Epoch 27 / 30) train acc: 0.570000; val_acc: 0.528000
(Iteration 6701 / 7350) loss: 1.144590
(Iteration 6801 / 7350) loss: 1.183924
(Epoch 28 / 30) train acc: 0.564000; val_acc: 0.525000
(Iteration 6901 / 7350) loss: 1.211566
(Iteration 7001 / 7350) loss: 1.274509
(Iteration 7101 / 7350) loss: 1.162291
(Epoch 29 / 30) train acc: 0.578000; val_acc: 0.520000
(Iteration 7201 / 7350) loss: 1.156013
(Iteration 7301 / 7350) loss: 1.281307
(Epoch 30 / 30) train acc: 0.561000; val_acc: 0.519000
new best accuracy:  0.528
```

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:  0.528
```

[ ]:
```
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:  0.519
```

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：

对训练集拟合效果好而对测试集拟合效果差得多, 说明对训练集存在过拟合现象, 可以通过增大训练集或增大 regularization 程度来避免过拟合现象的发生.

[ ]:

features

September 6, 2022

[25]:
```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive')

# TODO: 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 /content/drive/My\ Drive/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

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

[26]:
```
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
```

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

1.1 Load data

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

[27]:
```
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.

[28]:
```
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.

[29]:
```
# Use the validation set to tune the learning rate and regularization strength

from cs231n.classifiers.linear_classifier import LinearSVM

learning_rates = np.arange(1e-8, 50e-8, 5e-8)
regularization_strengths = np.arange(1e5, 10e5, 2e5)

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 lr in learning_rates:
  for reg in regularization_strengths:
    svm = LinearSVM()
    svm.train(X_train_feats, y_train, learning_rate=lr, reg=reg,␣
 ↪num_iters=1500, verbose=False)

    y_train_pred = svm.predict(X_train_feats)
    y_val_pred = svm.predict(X_val_feats)

    train_accuracy = np.mean(y_train == y_train_pred)
    val_accuracy = np.mean(y_val == y_val_pred)

    results.update({(lr, reg):(train_accuracy, val_accuracy)})
    print('lr %e reg %e train accuracy: %f val accuracy: %f' % (
                lr, reg, train_accuracy, val_accuracy))
```

```
    if val_accuracy > best_val:
      print(f'new best val accuracy: {val_accuracy}')
      best_val = val_accuracy
      best_svm = svm

# *****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)
```

```
lr 1.000000e-08 reg 1.000000e+05 train accuracy: 0.136122 val accuracy: 0.146000
new best val accuracy: 0.146
lr 1.000000e-08 reg 3.000000e+05 train accuracy: 0.411980 val accuracy: 0.409000
new best val accuracy: 0.409
lr 1.000000e-08 reg 5.000000e+05 train accuracy: 0.416163 val accuracy: 0.418000
new best val accuracy: 0.418
lr 1.000000e-08 reg 7.000000e+05 train accuracy: 0.411612 val accuracy: 0.405000
lr 1.000000e-08 reg 9.000000e+05 train accuracy: 0.419143 val accuracy: 0.428000
new best val accuracy: 0.428
lr 6.000000e-08 reg 1.000000e+05 train accuracy: 0.414000 val accuracy: 0.420000
lr 6.000000e-08 reg 3.000000e+05 train accuracy: 0.412122 val accuracy: 0.416000
lr 6.000000e-08 reg 5.000000e+05 train accuracy: 0.412367 val accuracy: 0.426000
lr 6.000000e-08 reg 7.000000e+05 train accuracy: 0.398796 val accuracy: 0.387000
lr 6.000000e-08 reg 9.000000e+05 train accuracy: 0.401735 val accuracy: 0.397000
lr 1.100000e-07 reg 1.000000e+05 train accuracy: 0.411633 val accuracy: 0.418000
lr 1.100000e-07 reg 3.000000e+05 train accuracy: 0.409551 val accuracy: 0.418000
lr 1.100000e-07 reg 5.000000e+05 train accuracy: 0.403347 val accuracy: 0.391000
lr 1.100000e-07 reg 7.000000e+05 train accuracy: 0.399041 val accuracy: 0.381000
lr 1.100000e-07 reg 9.000000e+05 train accuracy: 0.396939 val accuracy: 0.382000
lr 1.600000e-07 reg 1.000000e+05 train accuracy: 0.418510 val accuracy: 0.420000
lr 1.600000e-07 reg 3.000000e+05 train accuracy: 0.408714 val accuracy: 0.406000
lr 1.600000e-07 reg 5.000000e+05 train accuracy: 0.403531 val accuracy: 0.398000
lr 1.600000e-07 reg 7.000000e+05 train accuracy: 0.393816 val accuracy: 0.406000
lr 1.600000e-07 reg 9.000000e+05 train accuracy: 0.397367 val accuracy: 0.408000
lr 2.100000e-07 reg 1.000000e+05 train accuracy: 0.421122 val accuracy: 0.419000
lr 2.100000e-07 reg 3.000000e+05 train accuracy: 0.411898 val accuracy: 0.412000
lr 2.100000e-07 reg 5.000000e+05 train accuracy: 0.386592 val accuracy: 0.368000
lr 2.100000e-07 reg 7.000000e+05 train accuracy: 0.391490 val accuracy: 0.391000
lr 2.100000e-07 reg 9.000000e+05 train accuracy: 0.366143 val accuracy: 0.381000
lr 2.600000e-07 reg 1.000000e+05 train accuracy: 0.408184 val accuracy: 0.411000
lr 2.600000e-07 reg 3.000000e+05 train accuracy: 0.391102 val accuracy: 0.371000
```

```
lr 2.600000e-07 reg 5.000000e+05 train accuracy: 0.390612 val accuracy: 0.381000
lr 2.600000e-07 reg 7.000000e+05 train accuracy: 0.376122 val accuracy: 0.380000
lr 2.600000e-07 reg 9.000000e+05 train accuracy: 0.374796 val accuracy: 0.394000
lr 3.100000e-07 reg 1.000000e+05 train accuracy: 0.408735 val accuracy: 0.410000
lr 3.100000e-07 reg 3.000000e+05 train accuracy: 0.403347 val accuracy: 0.401000
lr 3.100000e-07 reg 5.000000e+05 train accuracy: 0.384388 val accuracy: 0.386000
lr 3.100000e-07 reg 7.000000e+05 train accuracy: 0.357939 val accuracy: 0.364000
lr 3.100000e-07 reg 9.000000e+05 train accuracy: 0.357653 val accuracy: 0.364000
lr 3.600000e-07 reg 1.000000e+05 train accuracy: 0.411714 val accuracy: 0.405000
lr 3.600000e-07 reg 3.000000e+05 train accuracy: 0.389449 val accuracy: 0.383000
lr 3.600000e-07 reg 5.000000e+05 train accuracy: 0.386939 val accuracy: 0.366000
lr 3.600000e-07 reg 7.000000e+05 train accuracy: 0.387408 val accuracy: 0.379000
lr 3.600000e-07 reg 9.000000e+05 train accuracy: 0.351633 val accuracy: 0.368000
lr 4.100000e-07 reg 1.000000e+05 train accuracy: 0.403367 val accuracy: 0.413000
lr 4.100000e-07 reg 3.000000e+05 train accuracy: 0.387184 val accuracy: 0.386000
lr 4.100000e-07 reg 5.000000e+05 train accuracy: 0.377776 val accuracy: 0.385000
lr 4.100000e-07 reg 7.000000e+05 train accuracy: 0.370980 val accuracy: 0.352000
lr 4.100000e-07 reg 9.000000e+05 train accuracy: 0.361633 val accuracy: 0.352000
lr 4.600000e-07 reg 1.000000e+05 train accuracy: 0.405082 val accuracy: 0.399000
lr 4.600000e-07 reg 3.000000e+05 train accuracy: 0.388429 val accuracy: 0.387000
lr 4.600000e-07 reg 5.000000e+05 train accuracy: 0.378347 val accuracy: 0.389000
lr 4.600000e-07 reg 7.000000e+05 train accuracy: 0.371612 val accuracy: 0.386000
lr 4.600000e-07 reg 9.000000e+05 train accuracy: 0.344306 val accuracy: 0.360000
lr 1.000000e-08 reg 1.000000e+05 train accuracy: 0.136122 val accuracy: 0.146000
lr 1.000000e-08 reg 3.000000e+05 train accuracy: 0.411980 val accuracy: 0.409000
lr 1.000000e-08 reg 5.000000e+05 train accuracy: 0.416163 val accuracy: 0.418000
lr 1.000000e-08 reg 7.000000e+05 train accuracy: 0.411612 val accuracy: 0.405000
lr 1.000000e-08 reg 9.000000e+05 train accuracy: 0.419143 val accuracy: 0.428000
lr 6.000000e-08 reg 1.000000e+05 train accuracy: 0.414000 val accuracy: 0.420000
lr 6.000000e-08 reg 3.000000e+05 train accuracy: 0.412122 val accuracy: 0.416000
lr 6.000000e-08 reg 5.000000e+05 train accuracy: 0.412367 val accuracy: 0.426000
lr 6.000000e-08 reg 7.000000e+05 train accuracy: 0.398796 val accuracy: 0.387000
lr 6.000000e-08 reg 9.000000e+05 train accuracy: 0.401735 val accuracy: 0.397000
lr 1.100000e-07 reg 1.000000e+05 train accuracy: 0.411633 val accuracy: 0.418000
lr 1.100000e-07 reg 3.000000e+05 train accuracy: 0.409551 val accuracy: 0.418000
lr 1.100000e-07 reg 5.000000e+05 train accuracy: 0.403347 val accuracy: 0.391000
lr 1.100000e-07 reg 7.000000e+05 train accuracy: 0.399041 val accuracy: 0.381000
lr 1.100000e-07 reg 9.000000e+05 train accuracy: 0.396939 val accuracy: 0.382000
lr 1.600000e-07 reg 1.000000e+05 train accuracy: 0.418510 val accuracy: 0.420000
lr 1.600000e-07 reg 3.000000e+05 train accuracy: 0.408714 val accuracy: 0.406000
lr 1.600000e-07 reg 5.000000e+05 train accuracy: 0.403531 val accuracy: 0.398000
lr 1.600000e-07 reg 7.000000e+05 train accuracy: 0.393816 val accuracy: 0.406000
lr 1.600000e-07 reg 9.000000e+05 train accuracy: 0.397367 val accuracy: 0.408000
lr 2.100000e-07 reg 1.000000e+05 train accuracy: 0.421122 val accuracy: 0.419000
lr 2.100000e-07 reg 3.000000e+05 train accuracy: 0.411898 val accuracy: 0.412000
lr 2.100000e-07 reg 5.000000e+05 train accuracy: 0.386592 val accuracy: 0.368000
lr 2.100000e-07 reg 7.000000e+05 train accuracy: 0.391490 val accuracy: 0.391000
lr 2.100000e-07 reg 9.000000e+05 train accuracy: 0.366143 val accuracy: 0.381000
```

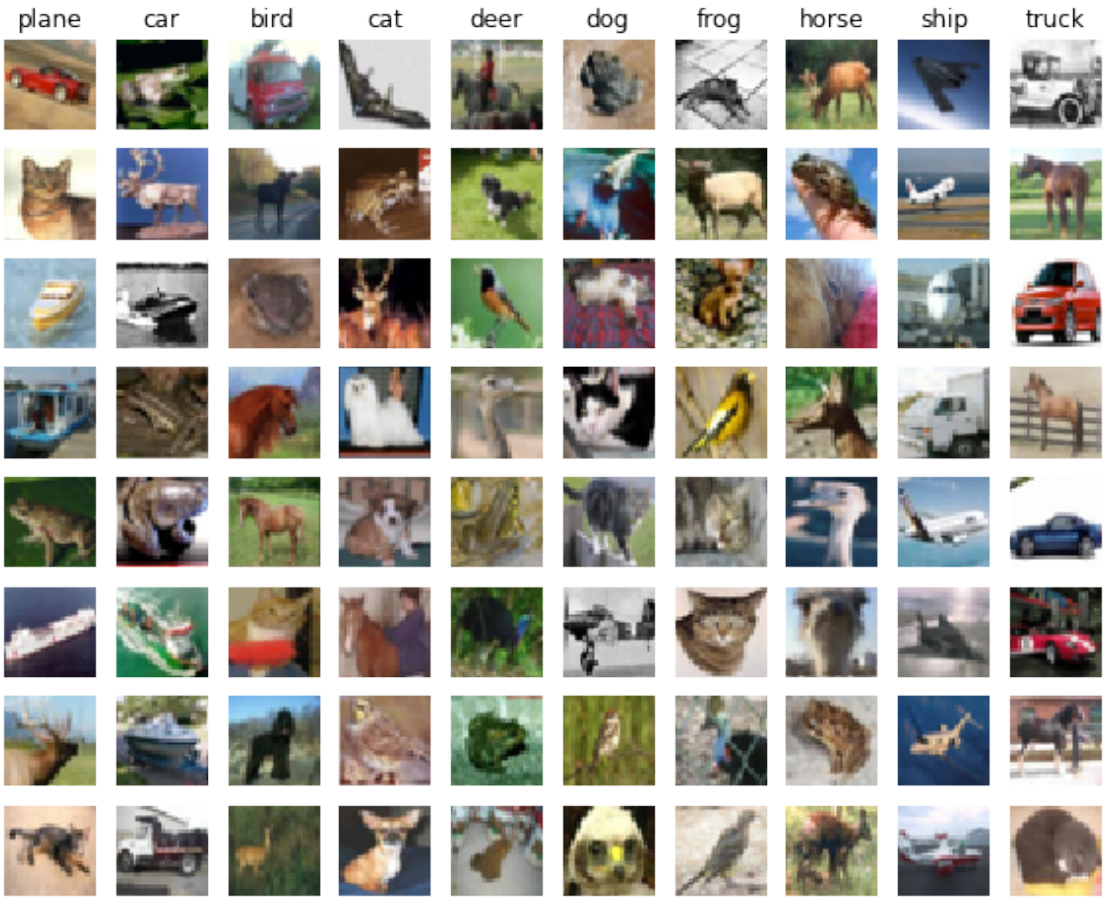
```
lr 2.600000e-07 reg 1.000000e+05 train accuracy: 0.408184 val accuracy: 0.411000
lr 2.600000e-07 reg 3.000000e+05 train accuracy: 0.391102 val accuracy: 0.371000
lr 2.600000e-07 reg 5.000000e+05 train accuracy: 0.390612 val accuracy: 0.381000
lr 2.600000e-07 reg 7.000000e+05 train accuracy: 0.376122 val accuracy: 0.380000
lr 2.600000e-07 reg 9.000000e+05 train accuracy: 0.374796 val accuracy: 0.394000
lr 3.100000e-07 reg 1.000000e+05 train accuracy: 0.408735 val accuracy: 0.410000
lr 3.100000e-07 reg 3.000000e+05 train accuracy: 0.403347 val accuracy: 0.401000
lr 3.100000e-07 reg 5.000000e+05 train accuracy: 0.384388 val accuracy: 0.386000
lr 3.100000e-07 reg 7.000000e+05 train accuracy: 0.357939 val accuracy: 0.364000
lr 3.100000e-07 reg 9.000000e+05 train accuracy: 0.357653 val accuracy: 0.364000
lr 3.600000e-07 reg 1.000000e+05 train accuracy: 0.411714 val accuracy: 0.405000
lr 3.600000e-07 reg 3.000000e+05 train accuracy: 0.389449 val accuracy: 0.383000
lr 3.600000e-07 reg 5.000000e+05 train accuracy: 0.386939 val accuracy: 0.366000
lr 3.600000e-07 reg 7.000000e+05 train accuracy: 0.387408 val accuracy: 0.379000
lr 3.600000e-07 reg 9.000000e+05 train accuracy: 0.351633 val accuracy: 0.368000
lr 4.100000e-07 reg 1.000000e+05 train accuracy: 0.403367 val accuracy: 0.413000
lr 4.100000e-07 reg 3.000000e+05 train accuracy: 0.387184 val accuracy: 0.386000
lr 4.100000e-07 reg 5.000000e+05 train accuracy: 0.377776 val accuracy: 0.385000
lr 4.100000e-07 reg 7.000000e+05 train accuracy: 0.370980 val accuracy: 0.352000
lr 4.100000e-07 reg 9.000000e+05 train accuracy: 0.361633 val accuracy: 0.352000
lr 4.600000e-07 reg 1.000000e+05 train accuracy: 0.405082 val accuracy: 0.399000
lr 4.600000e-07 reg 3.000000e+05 train accuracy: 0.388429 val accuracy: 0.387000
lr 4.600000e-07 reg 5.000000e+05 train accuracy: 0.378347 val accuracy: 0.389000
lr 4.600000e-07 reg 7.000000e+05 train accuracy: 0.371612 val accuracy: 0.386000
lr 4.600000e-07 reg 9.000000e+05 train accuracy: 0.344306 val accuracy: 0.360000
best validation accuracy achieved: 0.428000
```

[30]:
```
# 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)
```

```
0.423
```

[31]:
```
# 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?

YourAnswer :

在纹理, 形状和颜色方面都和被错误分到的类别的平均特征比较相像.

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.

[32]:
```
# 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)
```

[55]:
```
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

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,
}

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)*****

best_acc = -1
```

```
# learning_rates = np.arange(1e-8, 50e-8, 5e-8)5e-4,
learning_rates = [ 8e-2, 1e-1]

# default learning_rate: 5e-4
# regularization_strengths = np.arange(1e5, 10e5, 2e5)
regularization_strengths = [0, 1e-5, 1e-3]

for lr in learning_rates:
  for reg in regularization_strengths:
    print("\nlr=[%e], reg=[%e]" % (lr, reg))

    model = TwoLayerNet(input_dim, hidden_dim, num_classes, weight_scale=1e-3,␣
 ↪reg=reg)

    solver = Solver(model, data,
                    update_rule='sgd',
                    optim_config={'learning_rate': lr},
                    lr_decay=0.95, num_epochs=20,
                    batch_size=200, print_every=100, verbose=1)
    solver.train()

    accuracy = solver.best_val_acc

    if accuracy > best_acc:
      best_net = model
      best_acc = accuracy
      print('new best accuracy: ', accuracy)

print('best accuracy: ', accuracy)

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

```
lr=[8.000000e-02], reg=[0.000000e+00]
(Iteration 1 / 4900) loss: 2.302622
(Epoch 0 / 20) train acc: 0.085000; val_acc: 0.078000
(Iteration 101 / 4900) loss: 2.286931
(Iteration 201 / 4900) loss: 1.928599
(Epoch 1 / 20) train acc: 0.344000; val_acc: 0.338000
(Iteration 301 / 4900) loss: 1.704965
(Iteration 401 / 4900) loss: 1.435758
(Epoch 2 / 20) train acc: 0.482000; val_acc: 0.477000
(Iteration 501 / 4900) loss: 1.414846
(Iteration 601 / 4900) loss: 1.444802
(Iteration 701 / 4900) loss: 1.394903
(Epoch 3 / 20) train acc: 0.531000; val_acc: 0.502000
```

```
(Iteration 801 / 4900) loss: 1.260161
(Iteration 901 / 4900) loss: 1.336885
(Epoch 4 / 20) train acc: 0.491000; val_acc: 0.513000
(Iteration 1001 / 4900) loss: 1.384597
(Iteration 1101 / 4900) loss: 1.306056
(Iteration 1201 / 4900) loss: 1.280956
(Epoch 5 / 20) train acc: 0.523000; val_acc: 0.531000
(Iteration 1301 / 4900) loss: 1.362702
(Iteration 1401 / 4900) loss: 1.317527
(Epoch 6 / 20) train acc: 0.553000; val_acc: 0.528000
(Iteration 1501 / 4900) loss: 1.126388
(Iteration 1601 / 4900) loss: 1.374586
(Iteration 1701 / 4900) loss: 1.081325
(Epoch 7 / 20) train acc: 0.549000; val_acc: 0.535000
(Iteration 1801 / 4900) loss: 1.219752
(Iteration 1901 / 4900) loss: 1.194975
(Epoch 8 / 20) train acc: 0.581000; val_acc: 0.537000
(Iteration 2001 / 4900) loss: 1.276603
(Iteration 2101 / 4900) loss: 1.308115
(Iteration 2201 / 4900) loss: 1.264899
(Epoch 9 / 20) train acc: 0.575000; val_acc: 0.555000
(Iteration 2301 / 4900) loss: 1.150610
(Iteration 2401 / 4900) loss: 1.335365
(Epoch 10 / 20) train acc: 0.588000; val_acc: 0.552000
(Iteration 2501 / 4900) loss: 1.199193
(Iteration 2601 / 4900) loss: 1.054419
(Epoch 11 / 20) train acc: 0.603000; val_acc: 0.557000
(Iteration 2701 / 4900) loss: 1.223268
(Iteration 2801 / 4900) loss: 1.133676
(Iteration 2901 / 4900) loss: 1.210104
(Epoch 12 / 20) train acc: 0.552000; val_acc: 0.560000
(Iteration 3001 / 4900) loss: 1.318343
(Iteration 3101 / 4900) loss: 1.139619
(Epoch 13 / 20) train acc: 0.580000; val_acc: 0.555000
(Iteration 3201 / 4900) loss: 1.145285
(Iteration 3301 / 4900) loss: 1.150572
(Iteration 3401 / 4900) loss: 1.254765
(Epoch 14 / 20) train acc: 0.618000; val_acc: 0.560000
(Iteration 3501 / 4900) loss: 1.253052
(Iteration 3601 / 4900) loss: 0.987641
(Epoch 15 / 20) train acc: 0.587000; val_acc: 0.569000
(Iteration 3701 / 4900) loss: 1.221455
(Iteration 3801 / 4900) loss: 1.110772
(Iteration 3901 / 4900) loss: 1.159542
(Epoch 16 / 20) train acc: 0.598000; val_acc: 0.572000
(Iteration 4001 / 4900) loss: 0.994873
(Iteration 4101 / 4900) loss: 1.179974
(Epoch 17 / 20) train acc: 0.609000; val_acc: 0.574000
```

```
(Iteration 4201 / 4900) loss: 1.114083
(Iteration 4301 / 4900) loss: 1.129617
(Iteration 4401 / 4900) loss: 1.141559
(Epoch 18 / 20) train acc: 0.636000; val_acc: 0.573000
(Iteration 4501 / 4900) loss: 1.099054
(Iteration 4601 / 4900) loss: 1.022275
(Epoch 19 / 20) train acc: 0.624000; val_acc: 0.575000
(Iteration 4701 / 4900) loss: 1.247808
(Iteration 4801 / 4900) loss: 1.076249
(Epoch 20 / 20) train acc: 0.662000; val_acc: 0.580000
new best accuracy:  0.58

lr=[8.000000e-02], reg=[1.000000e-05]
(Iteration 1 / 4900) loss: 2.302582
(Epoch 0 / 20) train acc: 0.102000; val_acc: 0.100000
(Iteration 101 / 4900) loss: 2.279332
(Iteration 201 / 4900) loss: 1.957010
(Epoch 1 / 20) train acc: 0.366000; val_acc: 0.362000
(Iteration 301 / 4900) loss: 1.689029
(Iteration 401 / 4900) loss: 1.377337
(Epoch 2 / 20) train acc: 0.453000; val_acc: 0.469000
(Iteration 501 / 4900) loss: 1.454473
(Iteration 601 / 4900) loss: 1.329089
(Iteration 701 / 4900) loss: 1.474159
(Epoch 3 / 20) train acc: 0.498000; val_acc: 0.506000
(Iteration 801 / 4900) loss: 1.355174
(Iteration 901 / 4900) loss: 1.298777
(Epoch 4 / 20) train acc: 0.549000; val_acc: 0.519000
(Iteration 1001 / 4900) loss: 1.292656
(Iteration 1101 / 4900) loss: 1.390064
(Iteration 1201 / 4900) loss: 1.419241
(Epoch 5 / 20) train acc: 0.538000; val_acc: 0.522000
(Iteration 1301 / 4900) loss: 1.154339
(Iteration 1401 / 4900) loss: 1.469299
(Epoch 6 / 20) train acc: 0.541000; val_acc: 0.517000
(Iteration 1501 / 4900) loss: 1.332341
(Iteration 1601 / 4900) loss: 1.278407
(Iteration 1701 / 4900) loss: 1.318819
(Epoch 7 / 20) train acc: 0.533000; val_acc: 0.541000
(Iteration 1801 / 4900) loss: 1.224090
(Iteration 1901 / 4900) loss: 1.314861
(Epoch 8 / 20) train acc: 0.563000; val_acc: 0.536000
(Iteration 2001 / 4900) loss: 1.241408
(Iteration 2101 / 4900) loss: 1.244774
(Iteration 2201 / 4900) loss: 1.243860
(Epoch 9 / 20) train acc: 0.576000; val_acc: 0.544000
(Iteration 2301 / 4900) loss: 1.130182
(Iteration 2401 / 4900) loss: 1.124001
```

```
(Epoch 10 / 20) train acc: 0.584000; val_acc: 0.547000
(Iteration 2501 / 4900) loss: 1.173580
(Iteration 2601 / 4900) loss: 1.216008
(Epoch 11 / 20) train acc: 0.599000; val_acc: 0.549000
(Iteration 2701 / 4900) loss: 1.125201
(Iteration 2801 / 4900) loss: 1.240387
(Iteration 2901 / 4900) loss: 1.202577
(Epoch 12 / 20) train acc: 0.581000; val_acc: 0.558000
(Iteration 3001 / 4900) loss: 1.218987
(Iteration 3101 / 4900) loss: 1.083482
(Epoch 13 / 20) train acc: 0.576000; val_acc: 0.556000
(Iteration 3201 / 4900) loss: 1.125721
(Iteration 3301 / 4900) loss: 1.296972
(Iteration 3401 / 4900) loss: 1.112760
(Epoch 14 / 20) train acc: 0.619000; val_acc: 0.568000
(Iteration 3501 / 4900) loss: 1.170662
(Iteration 3601 / 4900) loss: 1.148641
(Epoch 15 / 20) train acc: 0.586000; val_acc: 0.557000
(Iteration 3701 / 4900) loss: 1.224092
(Iteration 3801 / 4900) loss: 1.060980
(Iteration 3901 / 4900) loss: 1.121947
(Epoch 16 / 20) train acc: 0.609000; val_acc: 0.570000
(Iteration 4001 / 4900) loss: 1.131951
(Iteration 4101 / 4900) loss: 1.085411
(Epoch 17 / 20) train acc: 0.591000; val_acc: 0.577000
(Iteration 4201 / 4900) loss: 1.083873
(Iteration 4301 / 4900) loss: 1.111030
(Iteration 4401 / 4900) loss: 1.140475
(Epoch 18 / 20) train acc: 0.630000; val_acc: 0.571000
(Iteration 4501 / 4900) loss: 1.131660
(Iteration 4601 / 4900) loss: 1.099706
(Epoch 19 / 20) train acc: 0.620000; val_acc: 0.580000
(Iteration 4701 / 4900) loss: 1.114464
(Iteration 4801 / 4900) loss: 1.008431
(Epoch 20 / 20) train acc: 0.628000; val_acc: 0.584000
new best accuracy:  0.584

lr=[8.000000e-02], reg=[1.000000e-03]
(Iteration 1 / 4900) loss: 2.302619
(Epoch 0 / 20) train acc: 0.101000; val_acc: 0.119000
(Iteration 101 / 4900) loss: 2.282076
(Iteration 201 / 4900) loss: 1.953241
(Epoch 1 / 20) train acc: 0.344000; val_acc: 0.357000
(Iteration 301 / 4900) loss: 1.642411
(Iteration 401 / 4900) loss: 1.545704
(Epoch 2 / 20) train acc: 0.472000; val_acc: 0.467000
(Iteration 501 / 4900) loss: 1.470352
(Iteration 601 / 4900) loss: 1.466944
```

```
(Iteration 701 / 4900) loss: 1.568184
(Epoch 3 / 20) train acc: 0.499000; val_acc: 0.522000
(Iteration 801 / 4900) loss: 1.395515
(Iteration 901 / 4900) loss: 1.343099
(Epoch 4 / 20) train acc: 0.539000; val_acc: 0.502000
(Iteration 1001 / 4900) loss: 1.569294
(Iteration 1101 / 4900) loss: 1.453773
(Iteration 1201 / 4900) loss: 1.395249
(Epoch 5 / 20) train acc: 0.531000; val_acc: 0.524000
(Iteration 1301 / 4900) loss: 1.305968
(Iteration 1401 / 4900) loss: 1.293080
(Epoch 6 / 20) train acc: 0.534000; val_acc: 0.522000
(Iteration 1501 / 4900) loss: 1.296069
(Iteration 1601 / 4900) loss: 1.251443
(Iteration 1701 / 4900) loss: 1.246231
(Epoch 7 / 20) train acc: 0.553000; val_acc: 0.530000
(Iteration 1801 / 4900) loss: 1.302080
(Iteration 1901 / 4900) loss: 1.341497
(Epoch 8 / 20) train acc: 0.537000; val_acc: 0.536000
(Iteration 2001 / 4900) loss: 1.223531
(Iteration 2101 / 4900) loss: 1.277232
(Iteration 2201 / 4900) loss: 1.273507
(Epoch 9 / 20) train acc: 0.572000; val_acc: 0.534000
(Iteration 2301 / 4900) loss: 1.270039
(Iteration 2401 / 4900) loss: 1.235782
(Epoch 10 / 20) train acc: 0.565000; val_acc: 0.550000
(Iteration 2501 / 4900) loss: 1.218867
(Iteration 2601 / 4900) loss: 1.201168
(Epoch 11 / 20) train acc: 0.578000; val_acc: 0.554000
(Iteration 2701 / 4900) loss: 1.249023
(Iteration 2801 / 4900) loss: 1.215988
(Iteration 2901 / 4900) loss: 1.319889
(Epoch 12 / 20) train acc: 0.588000; val_acc: 0.550000
(Iteration 3001 / 4900) loss: 1.198954
(Iteration 3101 / 4900) loss: 1.229085
(Epoch 13 / 20) train acc: 0.591000; val_acc: 0.541000
(Iteration 3201 / 4900) loss: 1.157168
(Iteration 3301 / 4900) loss: 1.056483
(Iteration 3401 / 4900) loss: 1.249550
(Epoch 14 / 20) train acc: 0.570000; val_acc: 0.560000
(Iteration 3501 / 4900) loss: 1.115789
(Iteration 3601 / 4900) loss: 1.230970
(Epoch 15 / 20) train acc: 0.603000; val_acc: 0.560000
(Iteration 3701 / 4900) loss: 1.300283
(Iteration 3801 / 4900) loss: 1.098935
(Iteration 3901 / 4900) loss: 1.083509
(Epoch 16 / 20) train acc: 0.620000; val_acc: 0.580000
(Iteration 4001 / 4900) loss: 1.181173
```

```
(Iteration 4101 / 4900) loss: 1.214223
(Epoch 17 / 20) train acc: 0.597000; val_acc: 0.559000
(Iteration 4201 / 4900) loss: 1.168058
(Iteration 4301 / 4900) loss: 1.184302
(Iteration 4401 / 4900) loss: 1.176099
(Epoch 18 / 20) train acc: 0.580000; val_acc: 0.572000
(Iteration 4501 / 4900) loss: 1.075072
(Iteration 4601 / 4900) loss: 1.149669
(Epoch 19 / 20) train acc: 0.629000; val_acc: 0.576000
(Iteration 4701 / 4900) loss: 1.120599
(Iteration 4801 / 4900) loss: 1.146646
(Epoch 20 / 20) train acc: 0.601000; val_acc: 0.577000

lr=[1.000000e-01], reg=[0.000000e+00]
(Iteration 1 / 4900) loss: 2.302590
(Epoch 0 / 20) train acc: 0.112000; val_acc: 0.078000
(Iteration 101 / 4900) loss: 2.236383
(Iteration 201 / 4900) loss: 1.774593
(Epoch 1 / 20) train acc: 0.391000; val_acc: 0.393000
(Iteration 301 / 4900) loss: 1.541420
(Iteration 401 / 4900) loss: 1.434870
(Epoch 2 / 20) train acc: 0.493000; val_acc: 0.500000
(Iteration 501 / 4900) loss: 1.456043
(Iteration 601 / 4900) loss: 1.230485
(Iteration 701 / 4900) loss: 1.512534
(Epoch 3 / 20) train acc: 0.540000; val_acc: 0.509000
(Iteration 801 / 4900) loss: 1.310777
(Iteration 901 / 4900) loss: 1.443361
(Epoch 4 / 20) train acc: 0.529000; val_acc: 0.507000
(Iteration 1001 / 4900) loss: 1.375571
(Iteration 1101 / 4900) loss: 1.228884
(Iteration 1201 / 4900) loss: 1.375662
(Epoch 5 / 20) train acc: 0.552000; val_acc: 0.517000
(Iteration 1301 / 4900) loss: 1.224642
(Iteration 1401 / 4900) loss: 1.236784
(Epoch 6 / 20) train acc: 0.575000; val_acc: 0.534000
(Iteration 1501 / 4900) loss: 1.223496
(Iteration 1601 / 4900) loss: 1.240032
(Iteration 1701 / 4900) loss: 1.323501
(Epoch 7 / 20) train acc: 0.562000; val_acc: 0.551000
(Iteration 1801 / 4900) loss: 1.107593
(Iteration 1901 / 4900) loss: 1.076361
(Epoch 8 / 20) train acc: 0.575000; val_acc: 0.554000
(Iteration 2001 / 4900) loss: 1.094492
(Iteration 2101 / 4900) loss: 1.251643
(Iteration 2201 / 4900) loss: 1.283564
(Epoch 9 / 20) train acc: 0.572000; val_acc: 0.556000
(Iteration 2301 / 4900) loss: 1.167513
```

```
(Iteration 2401 / 4900) loss: 1.141331
(Epoch 10 / 20) train acc: 0.648000; val_acc: 0.566000
(Iteration 2501 / 4900) loss: 1.081622
(Iteration 2601 / 4900) loss: 1.204024
(Epoch 11 / 20) train acc: 0.613000; val_acc: 0.557000
(Iteration 2701 / 4900) loss: 1.186019
(Iteration 2801 / 4900) loss: 1.053492
(Iteration 2901 / 4900) loss: 1.285352
(Epoch 12 / 20) train acc: 0.605000; val_acc: 0.573000
(Iteration 3001 / 4900) loss: 1.032835
(Iteration 3101 / 4900) loss: 1.067538
(Epoch 13 / 20) train acc: 0.649000; val_acc: 0.573000
(Iteration 3201 / 4900) loss: 1.213955
(Iteration 3301 / 4900) loss: 1.132523
(Iteration 3401 / 4900) loss: 1.121052
(Epoch 14 / 20) train acc: 0.603000; val_acc: 0.588000
(Iteration 3501 / 4900) loss: 0.977680
(Iteration 3601 / 4900) loss: 0.938381
(Epoch 15 / 20) train acc: 0.642000; val_acc: 0.591000
(Iteration 3701 / 4900) loss: 1.123199
(Iteration 3801 / 4900) loss: 1.094130
(Iteration 3901 / 4900) loss: 1.044856
(Epoch 16 / 20) train acc: 0.631000; val_acc: 0.590000
(Iteration 4001 / 4900) loss: 0.968724
(Iteration 4101 / 4900) loss: 0.911977
(Epoch 17 / 20) train acc: 0.640000; val_acc: 0.596000
(Iteration 4201 / 4900) loss: 1.092546
(Iteration 4301 / 4900) loss: 1.106916
(Iteration 4401 / 4900) loss: 1.066006
(Epoch 18 / 20) train acc: 0.631000; val_acc: 0.597000
(Iteration 4501 / 4900) loss: 1.023933
(Iteration 4601 / 4900) loss: 1.116080
(Epoch 19 / 20) train acc: 0.650000; val_acc: 0.596000
(Iteration 4701 / 4900) loss: 1.002024
(Iteration 4801 / 4900) loss: 1.038736
(Epoch 20 / 20) train acc: 0.672000; val_acc: 0.599000
new best accuracy:  0.599

lr=[1.000000e-01], reg=[1.000000e-05]
(Iteration 1 / 4900) loss: 2.302619
(Epoch 0 / 20) train acc: 0.103000; val_acc: 0.113000
(Iteration 101 / 4900) loss: 2.259277
(Iteration 201 / 4900) loss: 1.737930
(Epoch 1 / 20) train acc: 0.409000; val_acc: 0.410000
(Iteration 301 / 4900) loss: 1.551953
(Iteration 401 / 4900) loss: 1.504815
(Epoch 2 / 20) train acc: 0.496000; val_acc: 0.499000
(Iteration 501 / 4900) loss: 1.407880
```

```
(Iteration 601 / 4900) loss: 1.363506
(Iteration 701 / 4900) loss: 1.313112
(Epoch 3 / 20) train acc: 0.519000; val_acc: 0.510000
(Iteration 801 / 4900) loss: 1.265264
(Iteration 901 / 4900) loss: 1.262774
(Epoch 4 / 20) train acc: 0.505000; val_acc: 0.523000
(Iteration 1001 / 4900) loss: 1.276024
(Iteration 1101 / 4900) loss: 1.369477
(Iteration 1201 / 4900) loss: 1.305326
(Epoch 5 / 20) train acc: 0.569000; val_acc: 0.519000
(Iteration 1301 / 4900) loss: 1.156364
(Iteration 1401 / 4900) loss: 1.277893
(Epoch 6 / 20) train acc: 0.565000; val_acc: 0.544000
(Iteration 1501 / 4900) loss: 1.312396
(Iteration 1601 / 4900) loss: 1.257143
(Iteration 1701 / 4900) loss: 1.170644
(Epoch 7 / 20) train acc: 0.555000; val_acc: 0.542000
(Iteration 1801 / 4900) loss: 1.241511
(Iteration 1901 / 4900) loss: 1.201878
(Epoch 8 / 20) train acc: 0.555000; val_acc: 0.547000
(Iteration 2001 / 4900) loss: 1.136193
(Iteration 2101 / 4900) loss: 1.185465
(Iteration 2201 / 4900) loss: 1.214198
(Epoch 9 / 20) train acc: 0.578000; val_acc: 0.562000
(Iteration 2301 / 4900) loss: 1.131311
(Iteration 2401 / 4900) loss: 1.080305
(Epoch 10 / 20) train acc: 0.607000; val_acc: 0.572000
(Iteration 2501 / 4900) loss: 1.181659
(Iteration 2601 / 4900) loss: 1.116178
(Epoch 11 / 20) train acc: 0.592000; val_acc: 0.568000
(Iteration 2701 / 4900) loss: 1.159250
(Iteration 2801 / 4900) loss: 1.041445
(Iteration 2901 / 4900) loss: 1.110197
(Epoch 12 / 20) train acc: 0.591000; val_acc: 0.575000
(Iteration 3001 / 4900) loss: 1.053921
(Iteration 3101 / 4900) loss: 1.168308
(Epoch 13 / 20) train acc: 0.604000; val_acc: 0.575000
(Iteration 3201 / 4900) loss: 1.152245
(Iteration 3301 / 4900) loss: 0.990027
(Iteration 3401 / 4900) loss: 1.061325
(Epoch 14 / 20) train acc: 0.632000; val_acc: 0.583000
(Iteration 3501 / 4900) loss: 0.925185
(Iteration 3601 / 4900) loss: 1.088775
(Epoch 15 / 20) train acc: 0.629000; val_acc: 0.586000
(Iteration 3701 / 4900) loss: 1.160887
(Iteration 3801 / 4900) loss: 0.999682
(Iteration 3901 / 4900) loss: 0.975366
(Epoch 16 / 20) train acc: 0.622000; val_acc: 0.587000
```

```
(Iteration 4001 / 4900) loss: 1.018237
(Iteration 4101 / 4900) loss: 0.980045
(Epoch 17 / 20) train acc: 0.664000; val_acc: 0.586000
(Iteration 4201 / 4900) loss: 0.877141
(Iteration 4301 / 4900) loss: 1.130042
(Iteration 4401 / 4900) loss: 0.902301
(Epoch 18 / 20) train acc: 0.638000; val_acc: 0.593000
(Iteration 4501 / 4900) loss: 1.000542
(Iteration 4601 / 4900) loss: 0.982715
(Epoch 19 / 20) train acc: 0.636000; val_acc: 0.597000
(Iteration 4701 / 4900) loss: 1.021509
(Iteration 4801 / 4900) loss: 1.093519
(Epoch 20 / 20) train acc: 0.636000; val_acc: 0.595000

lr=[1.000000e-01], reg=[1.000000e-03]
(Iteration 1 / 4900) loss: 2.302628
(Epoch 0 / 20) train acc: 0.106000; val_acc: 0.078000
(Iteration 101 / 4900) loss: 2.245800
(Iteration 201 / 4900) loss: 1.817439
(Epoch 1 / 20) train acc: 0.418000; val_acc: 0.414000
(Iteration 301 / 4900) loss: 1.537593
(Iteration 401 / 4900) loss: 1.515550
(Epoch 2 / 20) train acc: 0.502000; val_acc: 0.495000
(Iteration 501 / 4900) loss: 1.389303
(Iteration 601 / 4900) loss: 1.456539
(Iteration 701 / 4900) loss: 1.296751
(Epoch 3 / 20) train acc: 0.515000; val_acc: 0.517000
(Iteration 801 / 4900) loss: 1.412949
(Iteration 901 / 4900) loss: 1.348960
(Epoch 4 / 20) train acc: 0.521000; val_acc: 0.518000
(Iteration 1001 / 4900) loss: 1.259809
(Iteration 1101 / 4900) loss: 1.200192
(Iteration 1201 / 4900) loss: 1.363640
(Epoch 5 / 20) train acc: 0.529000; val_acc: 0.528000
(Iteration 1301 / 4900) loss: 1.238229
(Iteration 1401 / 4900) loss: 1.279729
(Epoch 6 / 20) train acc: 0.549000; val_acc: 0.536000
(Iteration 1501 / 4900) loss: 1.327279
(Iteration 1601 / 4900) loss: 1.251899
(Iteration 1701 / 4900) loss: 1.257346
(Epoch 7 / 20) train acc: 0.575000; val_acc: 0.549000
(Iteration 1801 / 4900) loss: 1.147661
(Iteration 1901 / 4900) loss: 1.299044
(Epoch 8 / 20) train acc: 0.570000; val_acc: 0.550000
(Iteration 2001 / 4900) loss: 1.288339
(Iteration 2101 / 4900) loss: 1.293492
(Iteration 2201 / 4900) loss: 1.048936
(Epoch 9 / 20) train acc: 0.613000; val_acc: 0.549000
```

```
(Iteration 2301 / 4900) loss: 1.137331
(Iteration 2401 / 4900) loss: 1.110009
(Epoch 10 / 20) train acc: 0.590000; val_acc: 0.556000
(Iteration 2501 / 4900) loss: 1.191745
(Iteration 2601 / 4900) loss: 1.336939
(Epoch 11 / 20) train acc: 0.587000; val_acc: 0.565000
(Iteration 2701 / 4900) loss: 1.220373
(Iteration 2801 / 4900) loss: 1.166228
(Iteration 2901 / 4900) loss: 1.176944
(Epoch 12 / 20) train acc: 0.609000; val_acc: 0.560000
(Iteration 3001 / 4900) loss: 1.106674
(Iteration 3101 / 4900) loss: 1.195104
(Epoch 13 / 20) train acc: 0.605000; val_acc: 0.568000
(Iteration 3201 / 4900) loss: 1.176410
(Iteration 3301 / 4900) loss: 1.138381
(Iteration 3401 / 4900) loss: 1.088084
(Epoch 14 / 20) train acc: 0.606000; val_acc: 0.565000
(Iteration 3501 / 4900) loss: 1.048339
(Iteration 3601 / 4900) loss: 1.281443
(Epoch 15 / 20) train acc: 0.600000; val_acc: 0.584000
(Iteration 3701 / 4900) loss: 1.125791
(Iteration 3801 / 4900) loss: 1.185219
(Iteration 3901 / 4900) loss: 1.089864
(Epoch 16 / 20) train acc: 0.621000; val_acc: 0.582000
(Iteration 4001 / 4900) loss: 1.035556
(Iteration 4101 / 4900) loss: 1.073107
(Epoch 17 / 20) train acc: 0.634000; val_acc: 0.586000
(Iteration 4201 / 4900) loss: 1.046265
(Iteration 4301 / 4900) loss: 1.129060
(Iteration 4401 / 4900) loss: 1.130642
(Epoch 18 / 20) train acc: 0.637000; val_acc: 0.592000
(Iteration 4501 / 4900) loss: 1.194807
(Iteration 4601 / 4900) loss: 1.016915
(Epoch 19 / 20) train acc: 0.642000; val_acc: 0.597000
(Iteration 4701 / 4900) loss: 1.141588
(Iteration 4801 / 4900) loss: 1.266221
(Epoch 20 / 20) train acc: 0.629000; val_acc: 0.592000
```

[56]:
```
# 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)
```

```
0.571
```