knn

August 26, 2023

[32]:
```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive')
!ls "/content/drive/My Drive/cs231n/assignment1/"
# TODO: Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment1/'
FOLDERNAME = "cs231n/assignment1/"
assert FOLDERNAME is not None, "/content/drive/My Drive/cs231n/assignment1/"

# 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).
collect_submission.ipynb  features.ipynb  README.md      two_layer_net.ipynb
collectSubmission.sh      knn.ipynb       softmax.ipynb
cs231n                    makepdf.py      svm.ipynb
/content/drive/My Drive/cs231n/assignment1/cs231n/datasets
/content/drive/My Drive/cs231n/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.

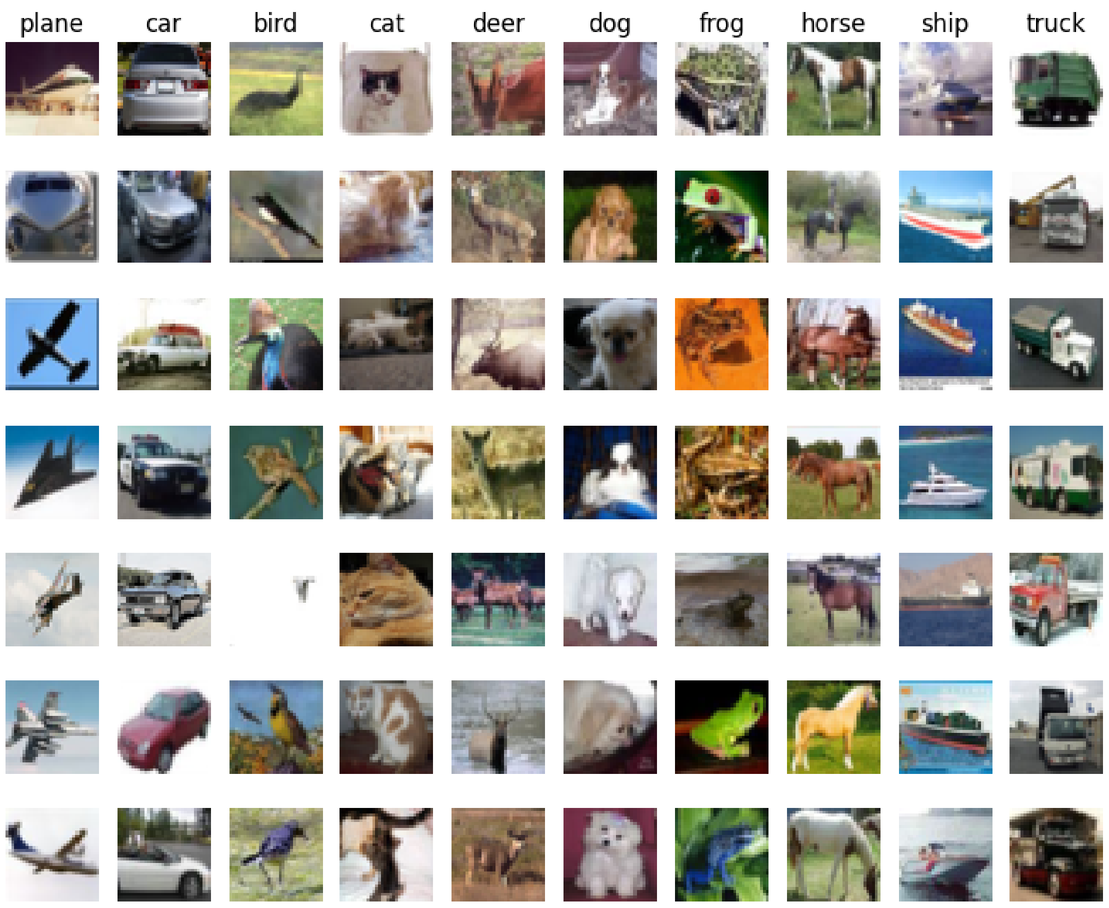
[33]:
```
# Run some setup code for this notebook.

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
```

[34]:
```
# 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,)
```

[35]:
```
# 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()
```



[36]:
```
# Subsample the data for more efficient code execution in this exercise
num_training = 5000
mask = list(range(num_training))
X_train = X_train[mask]
y_train = y_train[mask]

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

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

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

[37]:
```python
from cs231n.classifiers import KNearestNeighbor

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

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

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

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

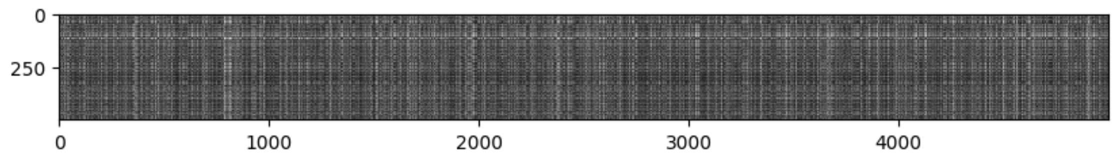
Note: For the three distance computations that we require you to implement in this notebook, you may not use the np.linalg.norm() function that numpy provides.

First, open `cs231n/classifiers/k_nearest_neighbor.py` and implement the function `compute_distances_two_loops` that uses a (very inefficient) double loop over all pairs of (test, train) examples and computes the distance matrix one element at a time.

[38]:
```python
# 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)
```

[39]:
```python
# 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 : The distinctly bright columns are caused by test points that are highly dissimilar to the training points.

The rows are similarily caused by data points that are highly dissimilar to the training points.

[40]:
```
# Now implement the function predict_labels and run the code below:
# We use k = 1 (which is Nearest Neighbor).
#print(y_test)
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`:

[41]:
```
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

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

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

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

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

Which of the following preprocessing steps will not change the performance of a Nearest Neighbor classifier that uses L1 distance? Select all that apply. To clarify, both training and test examples are preprocessed in the same way.

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, which means rotating all the images by the same angle. Empty regions in the image caused by rotation are padded with a same pixel value and no interpolation is performed.

$\color{blue}{\textit{Your Answer :}}$ 2,3,5

$\color{blue}{\textit{Your Explanatio}-n :}$ 2,3 - The same values are being subtracted and or divided for all pixels so (I1-I2) will make the mean cancel out. The division only scales the distances, so it will not affect performance as it maintains the original proportionality between the distances

5 - Padding value is same so it cancels out and the images still retain same distance.

```
[42]: # 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
```

```
[43]: # 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
```

[44]:
```
# 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 43.715152 seconds
One loop version took 36.869288 seconds
No loop version took 0.592264 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.

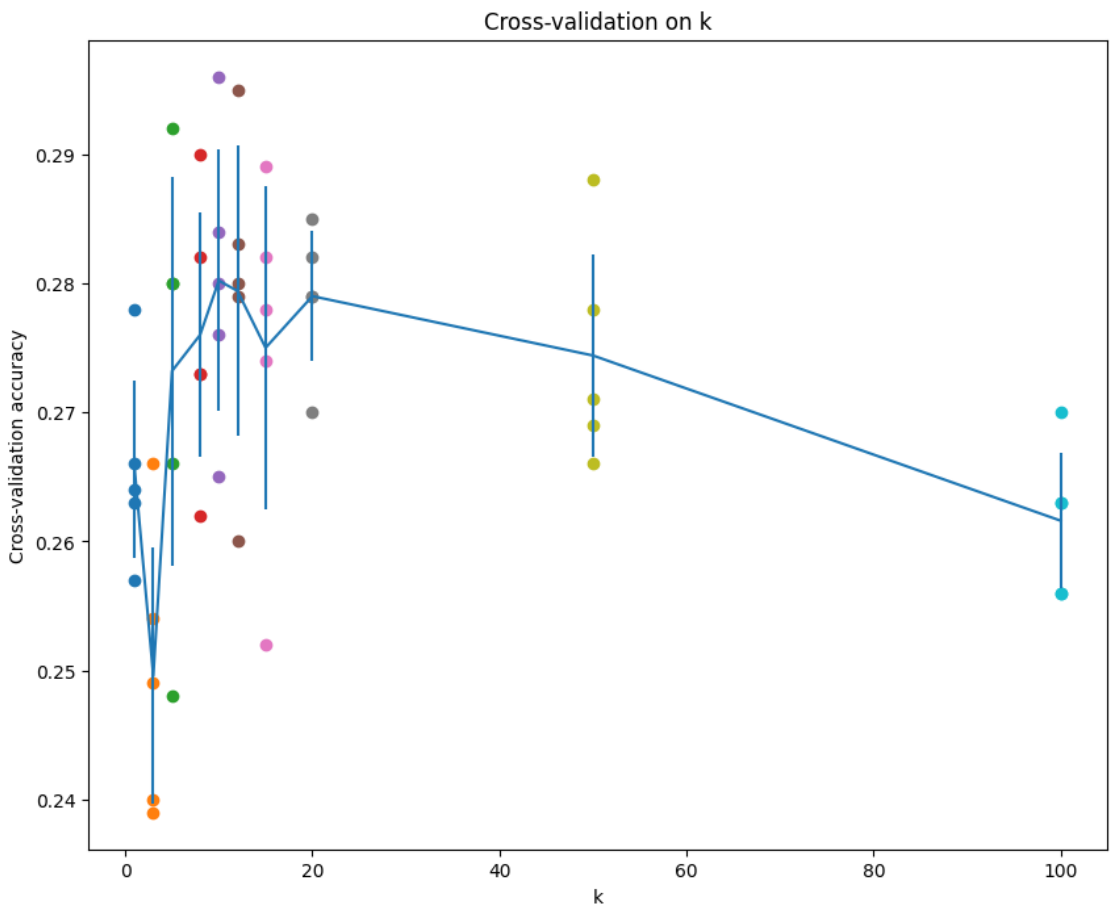
[45]:
```
num_folds = 5
k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50, 100]
```

```
X_train_folds = []
y_train_folds = []
################################################################################
# TODO:                                                                        #
# Split up the training data into folds. After splitting, X_train_folds and    #
# y_train_folds should each be lists of length num_folds, where                #
# y_train_folds[i] is the label vector for the points in X_train_folds[i].     #
# Hint: Look up the numpy array_split function.                                #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****


X_train_folds = np.array_split(X_train,num_folds)
y_train_folds = np.array_split(y_train,num_folds)

pass



# *****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:
  for i in range(0,num_folds):
    #setting up sets
    val_X = X_train_folds[i]
    val_y = y_train_folds[i]

    l = [0,1,2,3,4]
    l.pop(i)
    train_X = np.
↪concatenate((X_train_folds[(l[0])],X_train_folds[(l[1])],X_train_folds[(l[2])],X_train_fold
    train_y = np.
↪concatenate((y_train_folds[l[0]],y_train_folds[l[1]],y_train_folds[l[2]],y_train_folds[l[3]

    #testing the sets
    classifier.train(train_X,train_y)
    dists = classifier.compute_distances_no_loops(val_X)
    y_test_pred = classifier.predict_labels(dists, k=k)
    num_correct = np.sum(y_test_pred == val_y)
    acc = num_correct/1000

    if(i==0):
      k_to_accuracies[k] = [acc]
    else:
      k_to_accuracies[k].append(acc)

pass

# *****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
```

[46]:
```python
# 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()
```



[47]:
```
# 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.

$YourAnswer:$ 2,4

$YourExplanation:$

2 - 1-NN for each training point is zero as the nearest neighbor is itself so train error is zero.

4 - As the number of points increase the time taken to compute all the distances also increase.

svm

August 26, 2023

[1]:
```
# 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/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/assignment1/cs231n/datasets
/content/drive/My Drive/cs231n/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

[2]:
```
# Run some setup code for this notebook.
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

# This is a bit of magic to make matplotlib figures appear inline in the
# notebook rather than in a new window.
%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

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

1.1 CIFAR-10 Data Loading and Preprocessing

[3]:
```
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

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

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

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

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

[4]:
```python
# 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()
```

[5]:
```
# 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)
```

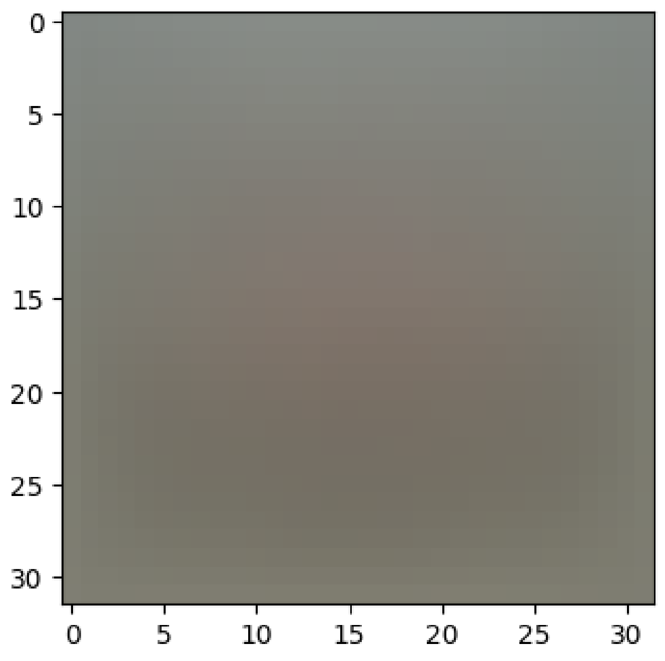
```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

[6]:
```python
# 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)
```

[7]:
```python
# Preprocessing: subtract the mean image
# first: compute the image mean based on the training data
mean_image = np.mean(X_train, axis=0)
print(mean_image[:10]) # print a few of the elements
plt.figure(figsize=(4,4))
plt.imshow(mean_image.reshape((32,32,3)).astype('uint8')) # visualize the mean␣
 ↪image
plt.show()

# second: subtract the mean image from train and test data
X_train -= mean_image
X_val -= mean_image
X_test -= mean_image
X_dev -= mean_image

# third: append the bias dimension of ones (i.e. bias trick) so that our SVM
# only has to worry about optimizing a single weight matrix W.
X_train = np.hstack([X_train, np.ones((X_train.shape[0], 1))])
X_val = np.hstack([X_val, np.ones((X_val.shape[0], 1))])
X_test = np.hstack([X_test, np.ones((X_test.shape[0], 1))])
X_dev = np.hstack([X_dev, np.ones((X_dev.shape[0], 1))])

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

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



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

1.2 SVM Classifier

Your code for this section will all be written inside `cs231n/classifiers/linear_svm.py`.

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

[8]:
```
# 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: 9.260003
```

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:

[104]:
```
# 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: -5.654452 analytic: -5.654452, relative error: 4.975813e-11
numerical: 43.585971 analytic: 43.585971, relative error: 7.401234e-12
numerical: 28.299253 analytic: 28.299253, relative error: 1.400271e-11
numerical: -24.772615 analytic: -24.772615, relative error: 8.944450e-12
numerical: -9.470119 analytic: -9.470119, relative error: 2.186111e-11
numerical: -1.200438 analytic: -1.200438, relative error: 1.321710e-11
numerical: -37.342561 analytic: -37.342561, relative error: 2.155837e-13
numerical: 26.597952 analytic: 26.597952, relative error: 1.494572e-11
numerical: -14.642281 analytic: -14.642281, relative error: 2.563185e-11
numerical: -66.536598 analytic: -66.536598, relative error: 8.607255e-13
numerical: 30.573219 analytic: 30.573219, relative error: 3.866380e-12
numerical: -16.673229 analytic: -16.673229, relative error: 6.534917e-12
numerical: -54.748533 analytic: -54.748533, relative error: 6.923143e-12
numerical: -39.221123 analytic: -39.221123, relative error: 4.984345e-12
numerical: -1.640401 analytic: -1.640401, relative error: 1.173521e-10
numerical: -35.849324 analytic: -35.849324, relative error: 4.320122e-12
numerical: 24.742604 analytic: 24.742604, relative error: 1.487675e-11
numerical: -12.119019 analytic: -12.119019, relative error: 1.441377e-11
numerical: 21.356389 analytic: 21.356389, relative error: 6.572634e-13
numerical: 11.620345 analytic: 11.620345, relative error: 3.447283e-12
```

Inline Question 1

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

$Your Answer:$ *fill this in.*

[105]:
```
# 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: 9.260003e+00 computed in 0.091412s
Vectorized loss: 9.260003e+00 computed in 0.012897s
difference: 0.000000
```

[106]:
```
# Complete the implementation of svm_loss_vectorized, and compute the gradient
# of the loss function in a vectorized way.

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

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

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

```
Naive loss and gradient: computed in 0.099214s
Vectorized loss and gradient: computed in 0.026862s
difference: 0.000000
```

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

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

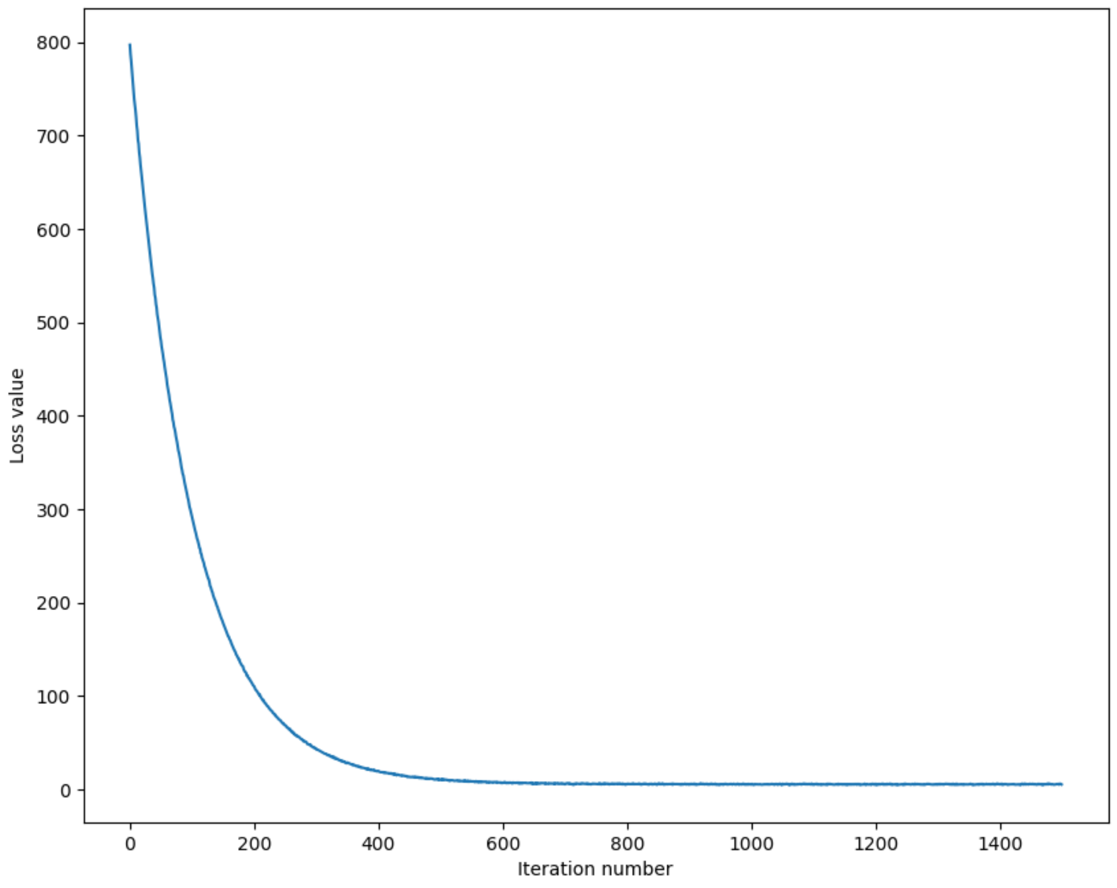
1.2.1 Stochastic Gradient Descent

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`.

```
[108]: # 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 797.112921
iteration 100 / 1500: loss 291.024551
iteration 200 / 1500: loss 110.122141
iteration 300 / 1500: loss 42.611639
iteration 400 / 1500: loss 19.500003
iteration 500 / 1500: loss 10.382832
iteration 600 / 1500: loss 7.127994
iteration 700 / 1500: loss 5.314368
iteration 800 / 1500: loss 5.700769
iteration 900 / 1500: loss 5.608419
iteration 1000 / 1500: loss 5.064672
iteration 1100 / 1500: loss 4.989330
iteration 1200 / 1500: loss 5.747474
iteration 1300 / 1500: loss 5.287791
iteration 1400 / 1500: loss 5.191813
That took 15.870582s
```

```
[109]: # 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()
```



[110]:
```
# 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: 0.359776
validation accuracy: 0.367000
```

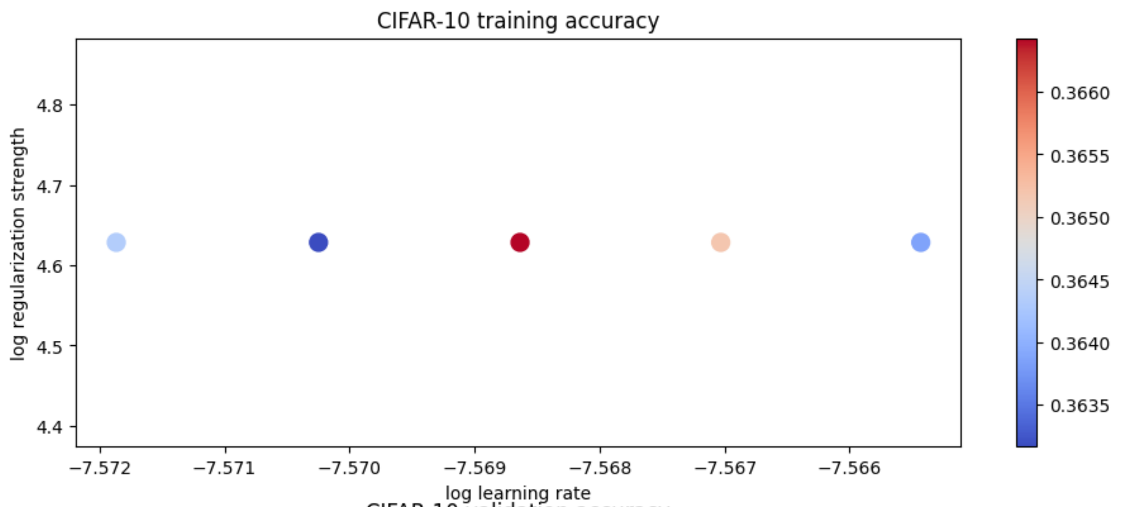
[114]:
```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of about 0.39 (> 0.385) 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 = [2.68e-8,2.69e-8,2.7e-8,2.71e-8,2.72e-8]
regularization_strengths = [4.25e4, 4.25e4,4.25e4,4.25e4,4.25e4]


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

for i in range(0,5):
  #train model
  svm = LinearSVM()
  print("model with lr: "+str(learning_rates[i])+" reg: "+␣
 ↪str(regularization_strengths[i]))
  loss_hist = svm.train(X_train, y_train, learning_rate=learning_rates[i],␣
 ↪reg=regularization_strengths[i],
                      num_iters=2500, verbose=True)
  #make preds
  y_train_pred = svm.predict(X_train)
  y_val_pred = svm.predict(X_val)

  #calc train/val acc
  train_acc = np.mean(y_train == y_train_pred)
  val_acc = np.mean(y_val == y_val_pred)
  if(val_acc>best_val):
        best_val = val_acc
        best_svm = svm
    #store in results
    results[(learning_rates[i], regularization_strengths[i])] =␣
    ↪(train_acc,val_acc)


pass

# *****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)
```

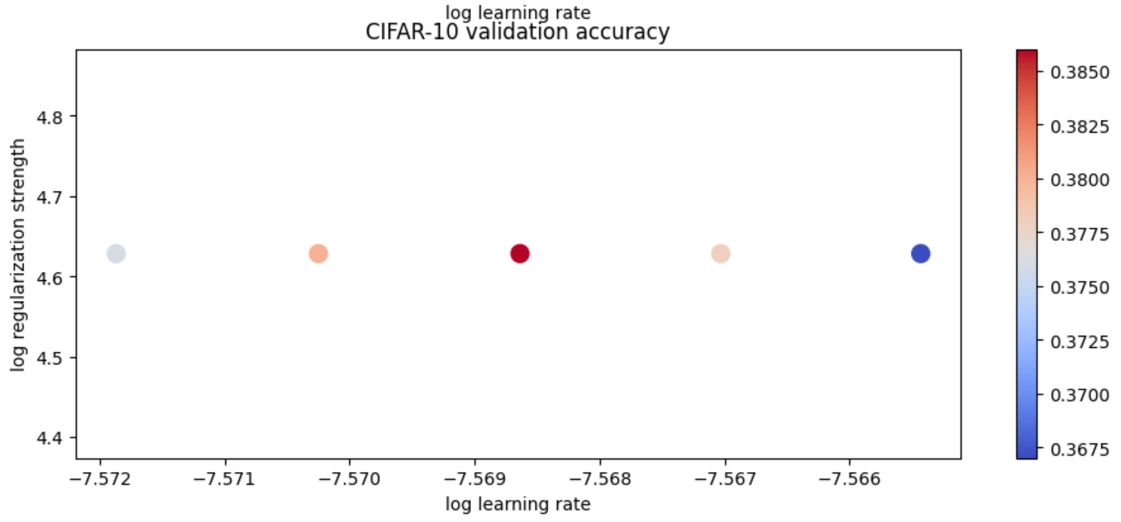
```
model with lr: 2.68e-08 reg: 42500.0
iteration 0 / 2500: loss 1320.764943
iteration 100 / 2500: loss 834.002598
iteration 200 / 2500: loss 528.925248
iteration 300 / 2500: loss 336.445796
iteration 400 / 2500: loss 214.243790
iteration 500 / 2500: loss 137.438221
iteration 600 / 2500: loss 88.896282
iteration 700 / 2500: loss 58.517347
iteration 800 / 2500: loss 39.049582
iteration 900 / 2500: loss 26.745296
iteration 1000 / 2500: loss 18.681211
iteration 1100 / 2500: loss 14.109073
iteration 1200 / 2500: loss 10.821628
iteration 1300 / 2500: loss 8.756656
iteration 1400 / 2500: loss 8.070923
iteration 1500 / 2500: loss 6.620173
iteration 1600 / 2500: loss 5.938848
iteration 1700 / 2500: loss 6.049881
iteration 1800 / 2500: loss 5.897083
iteration 1900 / 2500: loss 5.544029
iteration 2000 / 2500: loss 6.401948
iteration 2100 / 2500: loss 6.212023
iteration 2200 / 2500: loss 6.013684
iteration 2300 / 2500: loss 5.583563
iteration 2400 / 2500: loss 5.539701
model with lr: 2.69e-08 reg: 42500.0
```

```
iteration 0 / 2500: loss 1326.171704
iteration 100 / 2500: loss 835.835192
iteration 200 / 2500: loss 529.592124
iteration 300 / 2500: loss 336.104324
iteration 400 / 2500: loss 213.960515
iteration 500 / 2500: loss 137.916280
iteration 600 / 2500: loss 88.622150
iteration 700 / 2500: loss 58.104627
iteration 800 / 2500: loss 38.266499
iteration 900 / 2500: loss 26.048199
iteration 1000 / 2500: loss 18.679234
iteration 1100 / 2500: loss 13.689390
iteration 1200 / 2500: loss 10.822667
iteration 1300 / 2500: loss 8.720975
iteration 1400 / 2500: loss 7.462224
iteration 1500 / 2500: loss 6.659786
iteration 1600 / 2500: loss 6.467277
iteration 1700 / 2500: loss 6.043826
iteration 1800 / 2500: loss 5.515085
iteration 1900 / 2500: loss 5.331927
iteration 2000 / 2500: loss 6.044664
iteration 2100 / 2500: loss 5.413367
iteration 2200 / 2500: loss 6.233884
iteration 2300 / 2500: loss 5.129918
iteration 2400 / 2500: loss 5.241803
model with lr: 2.7e-08 reg: 42500.0
iteration 0 / 2500: loss 1326.176503
iteration 100 / 2500: loss 833.297165
iteration 200 / 2500: loss 525.787871
iteration 300 / 2500: loss 332.733592
iteration 400 / 2500: loss 212.023932
iteration 500 / 2500: loss 136.045619
iteration 600 / 2500: loss 87.210132
iteration 700 / 2500: loss 57.846409
iteration 800 / 2500: loss 37.718217
iteration 900 / 2500: loss 26.005085
iteration 1000 / 2500: loss 17.983122
iteration 1100 / 2500: loss 13.334739
iteration 1200 / 2500: loss 11.310689
iteration 1300 / 2500: loss 9.178774
iteration 1400 / 2500: loss 7.110467
iteration 1500 / 2500: loss 6.985372
iteration 1600 / 2500: loss 6.327061
iteration 1700 / 2500: loss 5.645074
iteration 1800 / 2500: loss 5.882890
iteration 1900 / 2500: loss 5.514758
iteration 2000 / 2500: loss 5.322593
iteration 2100 / 2500: loss 5.396829
iteration 2200 / 2500: loss 6.279607
iteration 2300 / 2500: loss 5.640155
iteration 2400 / 2500: loss 5.168483
model with lr: 2.71e-08 reg: 42500.0
iteration 0 / 2500: loss 1314.327281
iteration 100 / 2500: loss 826.774796
iteration 200 / 2500: loss 521.484596
iteration 300 / 2500: loss 330.377561
iteration 400 / 2500: loss 210.027160
iteration 500 / 2500: loss 133.719272
iteration 600 / 2500: loss 86.663279
iteration 700 / 2500: loss 56.200196
iteration 800 / 2500: loss 37.117349
iteration 900 / 2500: loss 25.182745
iteration 1000 / 2500: loss 17.865885
iteration 1100 / 2500: loss 13.627791
iteration 1200 / 2500: loss 10.455019
iteration 1300 / 2500: loss 8.765568
iteration 1400 / 2500: loss 7.739717
iteration 1500 / 2500: loss 6.708769
iteration 1600 / 2500: loss 6.256869
iteration 1700 / 2500: loss 6.693998
iteration 1800 / 2500: loss 6.023089
iteration 1900 / 2500: loss 6.102561
iteration 2000 / 2500: loss 5.352424
iteration 2100 / 2500: loss 5.508092
iteration 2200 / 2500: loss 5.848920
iteration 2300 / 2500: loss 5.449267
iteration 2400 / 2500: loss 5.057437
model with lr: 2.72e-08 reg: 42500.0
iteration 0 / 2500: loss 1325.323104
iteration 100 / 2500: loss 832.115726
iteration 200 / 2500: loss 523.237823
iteration 300 / 2500: loss 331.073109
iteration 400 / 2500: loss 209.950469
iteration 500 / 2500: loss 133.971665
iteration 600 / 2500: loss 86.300319
iteration 700 / 2500: loss 55.433274
iteration 800 / 2500: loss 37.302591
iteration 900 / 2500: loss 25.096538
iteration 1000 / 2500: loss 17.982404
iteration 1100 / 2500: loss 13.944073
iteration 1200 / 2500: loss 10.512447
iteration 1300 / 2500: loss 8.454511
iteration 1400 / 2500: loss 7.742112
iteration 1500 / 2500: loss 6.743620
iteration 1600 / 2500: loss 6.427155
iteration 1700 / 2500: loss 5.994082
iteration 1800 / 2500: loss 6.104318
iteration 1900 / 2500: loss 5.869412
iteration 2000 / 2500: loss 5.839761
iteration 2100 / 2500: loss 5.689155
iteration 2200 / 2500: loss 6.001094
iteration 2300 / 2500: loss 5.923258
iteration 2400 / 2500: loss 5.843511
lr 2.680000e-08 reg 4.250000e+04 train accuracy: 0.364347 val accuracy: 0.376000
lr 2.690000e-08 reg 4.250000e+04 train accuracy: 0.363163 val accuracy: 0.380000
lr 2.700000e-08 reg 4.250000e+04 train accuracy: 0.366429 val accuracy: 0.386000
lr 2.710000e-08 reg 4.250000e+04 train accuracy: 0.365184 val accuracy: 0.378000
lr 2.720000e-08 reg 4.250000e+04 train accuracy: 0.363857 val accuracy: 0.367000
best validation accuracy achieved during cross-validation: 0.386000
```

[115]:

```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()
```



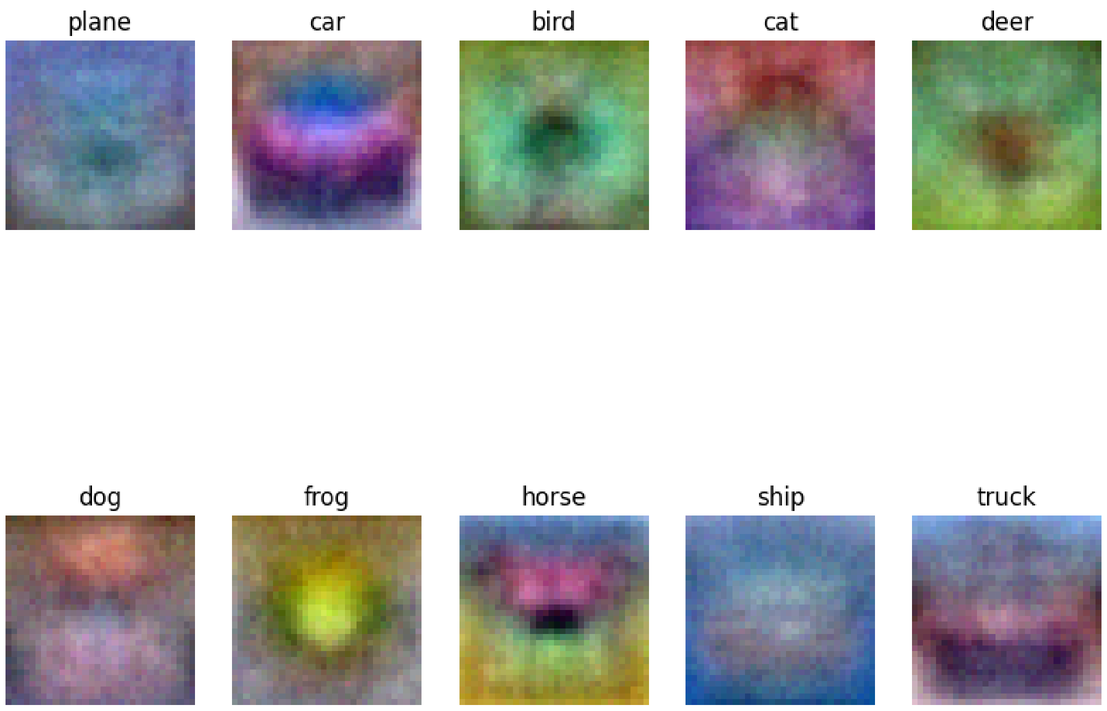


[116]:
```
# Evaluate the best svm on test set
y_test_pred = best_svm.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
```

```
linear SVM on raw pixels final test set accuracy: 0.369000
```

[117]:
```
# 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']
```

```python
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 : This is the image that generalizes a particular class from all the training data. It what the model expects a member of a particular class to look like based off what it has been optimized on.

softmax

August 26, 2023

[59]:
```
# 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/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/assignment1/cs231n/datasets
/content/drive/My Drive/cs231n/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

[60]:
```
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
```

[61]:
```
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`.

[52]:
```python
# 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: 2.346275
sanity check: 2.302585
```

Inline Question 1

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

$Your Answer:$ Weights are very close to zero making predictions (S) close to zero - > exponent of that gives value close to 1, computing L gives -log(1/(1+1+...+1)) = -log(1/10) = -log(1/C)

[62]:
```python
# 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: 1.773887 analytic: 1.773887, relative error: 9.357539e-09
```

```
numerical: -1.037751 analytic: -1.037751, relative error: 3.771695e-08
numerical: 1.073393 analytic: 1.073393, relative error: 5.789830e-08
numerical: -0.449861 analytic: -0.449861, relative error: 1.365268e-07
numerical: 0.459769 analytic: 0.459768, relative error: 1.100080e-07
numerical: 1.007710 analytic: 1.007709, relative error: 4.824643e-08
numerical: 2.529478 analytic: 2.529478, relative error: 2.142043e-08
numerical: 1.117490 analytic: 1.117490, relative error: 4.499801e-08
numerical: 2.436630 analytic: 2.436630, relative error: 1.579256e-08
numerical: 0.079457 analytic: 0.079457, relative error: 8.315314e-08
numerical: 3.494144 analytic: 3.494144, relative error: 1.069329e-08
numerical: -0.753010 analytic: -0.753010, relative error: 1.103002e-08
numerical: -0.116484 analytic: -0.116484, relative error: 1.675815e-07
numerical: 0.762472 analytic: 0.762472, relative error: 7.075627e-08
numerical: -0.419722 analytic: -0.419722, relative error: 8.492282e-08
numerical: -1.853717 analytic: -1.853717, relative error: 3.877640e-08
numerical: 4.099734 analytic: 4.099734, relative error: 7.639685e-09
numerical: 3.188594 analytic: 3.188594, relative error: 3.200801e-09
numerical: 1.214247 analytic: 1.214247, relative error: 5.096399e-08
numerical: 2.104101 analytic: 2.104101, relative error: 2.430960e-08
```

[91]:
```python
# 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: 2.355327e+00 computed in 0.442115s
(500, 10)
(500, 10)
```

```
(3073, 10)
vectorized loss: 2.355327e+00 computed in 0.025778s
Loss difference: 0.000000
Gradient difference: 0.000000
```

[100]:
```
# 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-7,1e-7,1e-7,1e-7]
regularization_strengths = [1.5e4, 2.0e4,2.5e4,3e4,3.5e4]

for i in range(0,5):
  #train model
  softmax = Softmax()
  print("model with lr: "+str(learning_rates[i])+" reg: "+␣
 ↪str(regularization_strengths[i]))
  loss_hist = softmax.train(X_train, y_train, learning_rate=learning_rates[i],␣
 ↪reg=regularization_strengths[i],
                      num_iters=2500, verbose=True)
  #make preds
  y_train_pred = softmax.predict(X_train)
  y_val_pred = softmax.predict(X_val)

  #calc train/val acc
  train_acc = np.mean(y_train == y_train_pred)
  val_acc = np.mean(y_val == y_val_pred)
  if(val_acc>best_val):
    best_val = val_acc
    best_softmax = softmax
  #store in results
```

```
        results[(learning_rates[i], regularization_strengths[i])] =␣
↪(train_acc,val_acc)


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

pass

# *****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)
```

```
model with lr: 1e-07 reg: 15000.0
iteration 0 / 2500: loss 933.878842
iteration 100 / 2500: loss 511.319441
iteration 200 / 2500: loss 280.423958
iteration 300 / 2500: loss 154.372205
iteration 400 / 2500: loss 85.419264
iteration 500 / 2500: loss 47.776781
iteration 600 / 2500: loss 27.066344
iteration 700 / 2500: loss 15.755652
iteration 800 / 2500: loss 9.611614
iteration 900 / 2500: loss 6.236905
iteration 1000 / 2500: loss 4.376144
iteration 1100 / 2500: loss 3.319126
iteration 1200 / 2500: loss 2.801058
iteration 1300 / 2500: loss 2.614131
iteration 1400 / 2500: loss 2.351742
iteration 1500 / 2500: loss 2.182451
iteration 1600 / 2500: loss 2.142170
iteration 1700 / 2500: loss 2.213256
iteration 1800 / 2500: loss 2.135185
iteration 1900 / 2500: loss 2.149198
iteration 2000 / 2500: loss 2.174494
iteration 2100 / 2500: loss 2.085774
iteration 2200 / 2500: loss 2.064387
iteration 2300 / 2500: loss 2.131866
iteration 2400 / 2500: loss 2.157387
model with lr: 5e-07 reg: 20000.0
iteration 0 / 2500: loss 1229.465087
iteration 100 / 2500: loss 23.454773
iteration 200 / 2500: loss 2.518201
iteration 300 / 2500: loss 2.287700
iteration 400 / 2500: loss 2.263231
iteration 500 / 2500: loss 2.114817
iteration 600 / 2500: loss 2.139965
iteration 700 / 2500: loss 2.136896
iteration 800 / 2500: loss 2.163296
iteration 900 / 2500: loss 2.192549
iteration 1000 / 2500: loss 2.108856
iteration 1100 / 2500: loss 2.116117
iteration 1200 / 2500: loss 2.164136
iteration 1300 / 2500: loss 2.188129
iteration 1400 / 2500: loss 2.162043
iteration 1500 / 2500: loss 2.169182
iteration 1600 / 2500: loss 2.174048
iteration 1700 / 2500: loss 2.130676
iteration 1800 / 2500: loss 2.155567
iteration 1900 / 2500: loss 2.212658
iteration 2000 / 2500: loss 2.219973
iteration 2100 / 2500: loss 2.204301
iteration 2200 / 2500: loss 2.182443
iteration 2300 / 2500: loss 2.165058
iteration 2400 / 2500: loss 2.204527
model with lr: 1e-07 reg: 25000.0
iteration 0 / 2500: loss 1522.171638
iteration 100 / 2500: loss 558.111086
iteration 200 / 2500: loss 205.680453
iteration 300 / 2500: loss 76.634231
iteration 400 / 2500: loss 29.414175
iteration 500 / 2500: loss 12.125787
iteration 600 / 2500: loss 5.802581
iteration 700 / 2500: loss 3.514320
iteration 800 / 2500: loss 2.639531
iteration 900 / 2500: loss 2.339378
iteration 1000 / 2500: loss 2.306471
iteration 1100 / 2500: loss 2.235888
iteration 1200 / 2500: loss 2.172299
iteration 1300 / 2500: loss 2.180579
iteration 1400 / 2500: loss 2.154472
iteration 1500 / 2500: loss 2.118483
iteration 1600 / 2500: loss 2.148106
iteration 1700 / 2500: loss 2.151270
iteration 1800 / 2500: loss 2.141910
iteration 1900 / 2500: loss 2.164047
iteration 2000 / 2500: loss 2.195140
iteration 2100 / 2500: loss 2.219355
iteration 2200 / 2500: loss 2.136609
iteration 2300 / 2500: loss 2.176771
iteration 2400 / 2500: loss 2.142882
model with lr: 1e-07 reg: 30000.0
iteration 0 / 2500: loss 1839.216052
iteration 100 / 2500: loss 551.781357
iteration 200 / 2500: loss 166.690329
iteration 300 / 2500: loss 51.370735
iteration 400 / 2500: loss 16.890672
iteration 500 / 2500: loss 6.642282
iteration 600 / 2500: loss 3.510025
iteration 700 / 2500: loss 2.583809
iteration 800 / 2500: loss 2.366420
iteration 900 / 2500: loss 2.175474
iteration 1000 / 2500: loss 2.193159
iteration 1100 / 2500: loss 2.183610
iteration 1200 / 2500: loss 2.203484
iteration 1300 / 2500: loss 2.224459
iteration 1400 / 2500: loss 2.186467
iteration 1500 / 2500: loss 2.167720
iteration 1600 / 2500: loss 2.160941
iteration 1700 / 2500: loss 2.198329
iteration 1800 / 2500: loss 2.191530
iteration 1900 / 2500: loss 2.176645
iteration 2000 / 2500: loss 2.178826
iteration 2100 / 2500: loss 2.161271
iteration 2200 / 2500: loss 2.199485
iteration 2300 / 2500: loss 2.177308
iteration 2400 / 2500: loss 2.227324
model with lr: 1e-07 reg: 35000.0
iteration 0 / 2500: loss 2145.916612
iteration 100 / 2500: loss 526.669853
iteration 200 / 2500: loss 130.496548
iteration 300 / 2500: loss 33.559380
iteration 400 / 2500: loss 9.912133
iteration 500 / 2500: loss 4.120230
iteration 600 / 2500: loss 2.654955
iteration 700 / 2500: loss 2.312548
iteration 800 / 2500: loss 2.227502
iteration 900 / 2500: loss 2.146994
iteration 1000 / 2500: loss 2.157033
iteration 1100 / 2500: loss 2.197371
iteration 1200 / 2500: loss 2.268923
iteration 1300 / 2500: loss 2.230314
iteration 1400 / 2500: loss 2.177205
iteration 1500 / 2500: loss 2.225661
iteration 1600 / 2500: loss 2.238512
iteration 1700 / 2500: loss 2.199799
iteration 1800 / 2500: loss 2.216480
iteration 1900 / 2500: loss 2.164264
iteration 2000 / 2500: loss 2.197753
iteration 2100 / 2500: loss 2.206698
iteration 2200 / 2500: loss 2.208068
iteration 2300 / 2500: loss 2.151890
iteration 2400 / 2500: loss 2.170927
lr 1.000000e-07 reg 1.500000e+04 train accuracy: 0.346163 val accuracy: 0.365000
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.329776 val accuracy: 0.342000
lr 1.000000e-07 reg 3.000000e+04 train accuracy: 0.321224 val accuracy: 0.341000
lr 1.000000e-07 reg 3.500000e+04 train accuracy: 0.310714 val accuracy: 0.325000
lr 5.000000e-07 reg 2.000000e+04 train accuracy: 0.339612 val accuracy: 0.351000
best validation accuracy achieved during cross-validation: 0.365000
```

[101]:
```python
# 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: 0.363000
```

Inline Question 2 - *True or False*

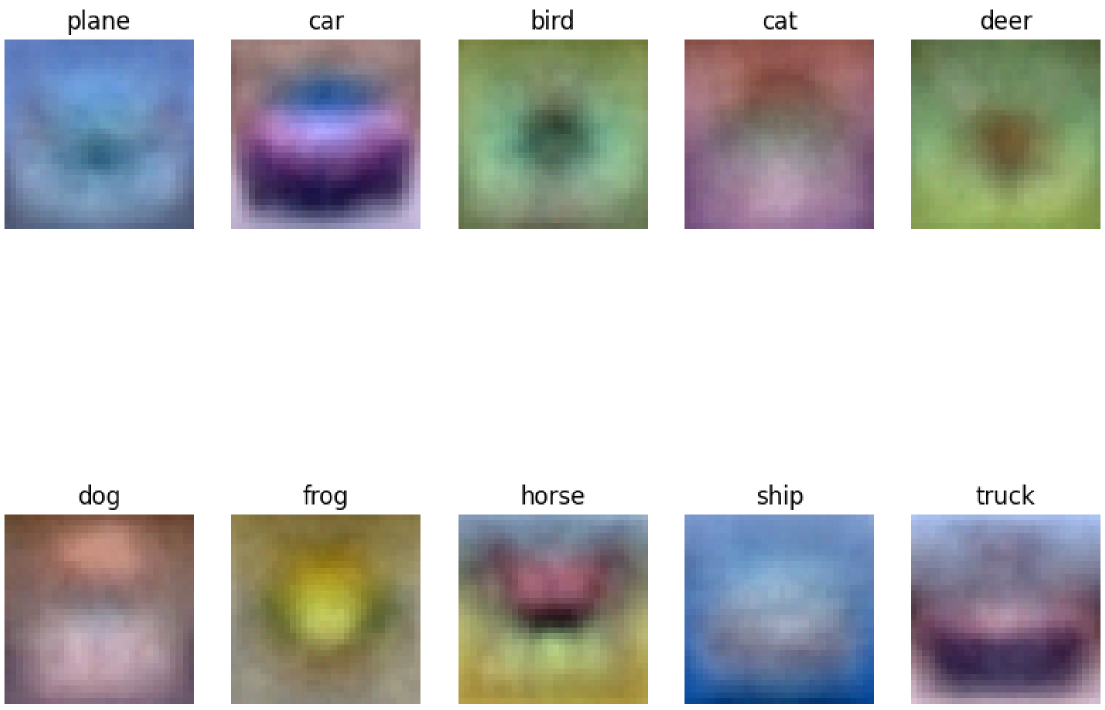
Suppose the overall training loss is defined as the sum of the per-datapoint loss over all training examples. It is possible to add a new datapoint to a training set that would leave the SVM loss unchanged, but this is not the case with the Softmax classifier loss.

$\color{blue}{\textit{Your Answer :}}$ True

$\color{blue}{\textit{Your Explanation :}}$ SVM - if we add a point that makes the margin 0 or less than zero it does not change the sum

Softmax - Addition of any new point changes the sum as the output of softmax is always >0. Moreover it changes the probablilites associated with each point so the sum definitely always changes.

[ ]:
```python
# 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])
```



```
[ ]:
```

two_layer_net

August 26, 2023

```
[ ]: # 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/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/assignment1/cs231n/datasets
/content/drive/My Drive/cs231n/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 - Sigmoid - in saturation kills gradient eg:[-20,0,20] only 0 gives a reasonable gradient while the rest give near zero gradient.

2 - ReLU - in saturation kills gradient eg:[-20,0,20] only 20 gives a reasonable gradient while the rest give near zero gradient , it effectively functions properly in the linear regime (this can cause certain parts of the data cloud that are negative to always be zero, producing improperly trained models).

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:  2.3025458445007376
dx error:  0.3333313285673707
```

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: 1.83e-08
W2 relative error: 3.31e-10
b1 relative error: 9.83e-09
b2 relative error: 4.33e-10
Running numeric gradient check with reg =  0.7
W1 relative error: 2.53e-07
W2 relative error: 2.85e-08
b1 relative error: 1.56e-08
b2 relative error: 7.76e-10
```

9 Solver

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

[ ]:
```python
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()
pass

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

```
(Iteration 1 / 1225) loss: 2.302049
(Epoch 0 / 5) train acc: 0.083000; val_acc: 0.075000
(Iteration 101 / 1225) loss: 2.283631
(Iteration 201 / 1225) loss: 2.136619
(Epoch 1 / 5) train acc: 0.222000; val_acc: 0.257000
(Iteration 301 / 1225) loss: 2.040722
(Iteration 401 / 1225) loss: 2.008701
```

```
(Epoch 2 / 5) train acc: 0.275000; val_acc: 0.298000
(Iteration 501 / 1225) loss: 1.944057
(Iteration 601 / 1225) loss: 1.919385
(Iteration 701 / 1225) loss: 1.984568
(Epoch 3 / 5) train acc: 0.309000; val_acc: 0.322000
(Iteration 801 / 1225) loss: 1.822463
(Iteration 901 / 1225) loss: 1.795208
(Epoch 4 / 5) train acc: 0.379000; val_acc: 0.349000
(Iteration 1001 / 1225) loss: 1.785058
(Iteration 1101 / 1225) loss: 1.808511
(Iteration 1201 / 1225) loss: 1.711310
(Epoch 5 / 5) train acc: 0.367000; val_acc: 0.375000
```

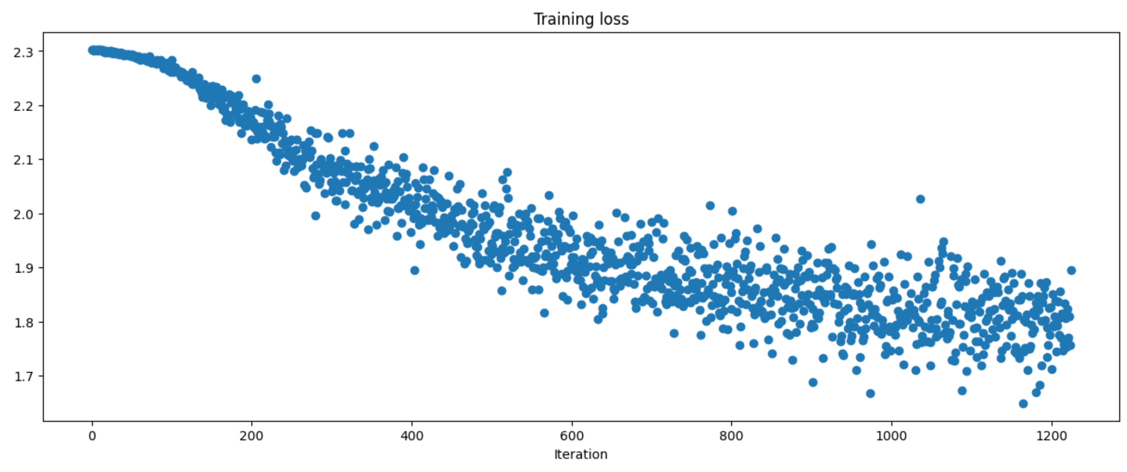
10 Debug the training

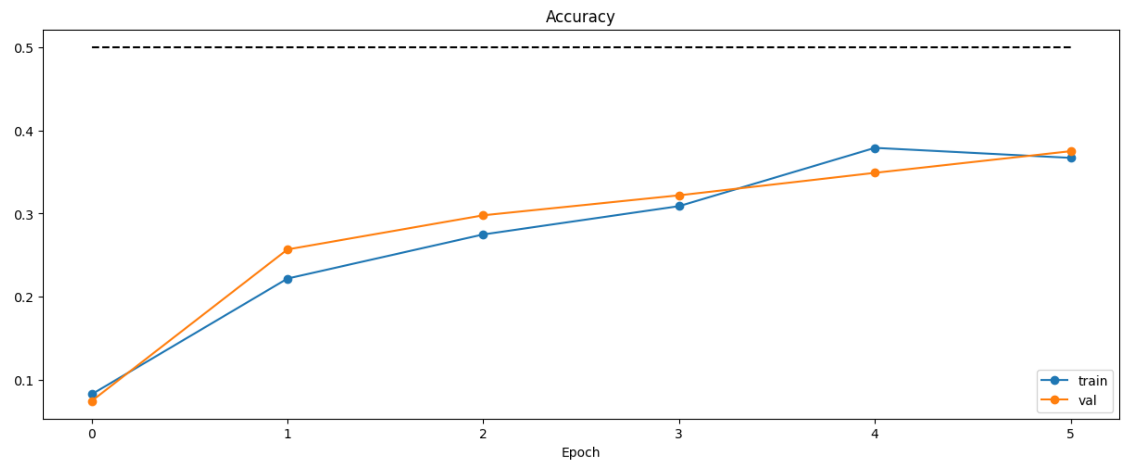
With the default parameters we provided above, you should get a validation accuracy of about 0.36 on the validation set. This isn't very good.

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.

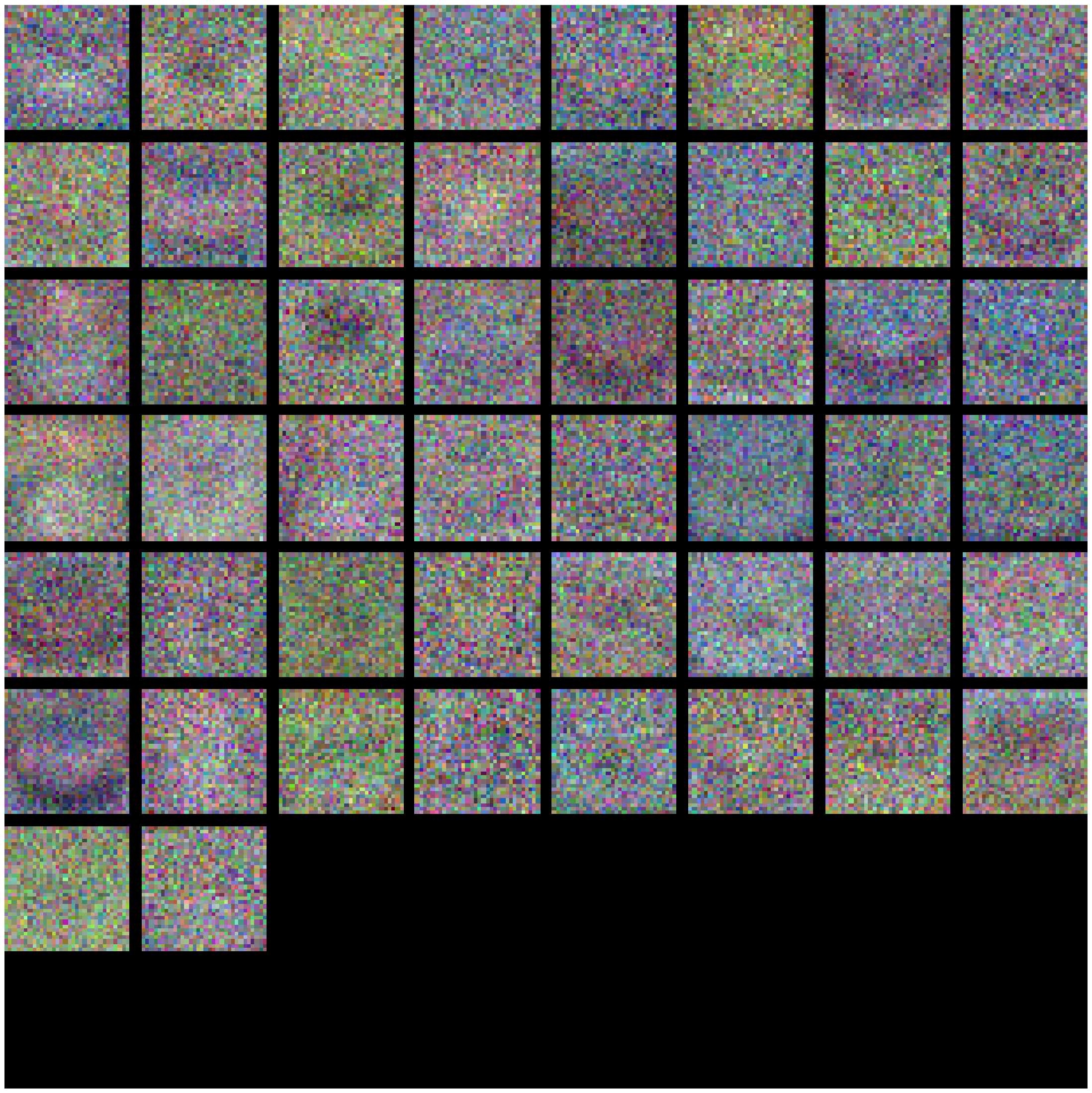
Another strategy is to visualize the weights that were learned in the first layer of the network. In most neural networks trained on visual data, the first layer weights typically show some visible structure when visualized.

[ ]:
```python
# 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
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
best_val = 0

#################################################################################
# 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)*****
learning_rates = [3.90e-4,3.95e-4,4e-4,4.05e-4,4.10e-4]
regularization_strengths = [0.68, 0.69,0.7,0.71,0.72]
learning_rate_decay = [0.98,0.982,0.984,0.986,0.988]

    # solver = Solver(model, data,
    #                 update_rule='sgd',
    #                 optim_config={
    #                   'learning_rate': 1e-4,
    #                  },
    #                  lr_decay=0.95,
    #                  num_epochs=7, batch_size=200,
    #                  print_every=100)

for i in range(0,5):
  #train model

  model = TwoLayerNet(input_size, hidden_size,␣
 ↪num_classes,reg=regularization_strengths[i])

  solver = Solver(model, data,
                  update_rule='sgd',
                  optim_config={
                    'learning_rate': learning_rates[i],
                  },
                  lr_decay=0.95,
                  num_epochs=8, batch_size=200,
                  print_every=500)

  print("model with lr: "+str(learning_rates[i])+" reg: "+␣
 ↪str(regularization_strengths[i])+ " decay: " +str(learning_rate_decay[i]))
  loss_hist = solver.train()
  #make preds
  y_train_pred = np.argmax(model.loss(data['X_train']), axis=1)
  y_val_pred = np.argmax(model.loss(data['X_val']), axis=1)

  #calc train/val acc
  train_acc = np.mean(data['y_train'] == y_train_pred)
  val_acc = np.mean(data['y_val'] == y_val_pred)
  if(val_acc>best_val):
    best_val = val_acc
    best_model = model
  #store in results
pass

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

```
model with lr: 0.00039 reg: 0.68 decay: 0.98
(Iteration 1 / 1960) loss: 2.357124
(Epoch 0 / 8) train acc: 0.089000; val_acc: 0.117000
(Epoch 1 / 8) train acc: 0.344000; val_acc: 0.370000
(Epoch 2 / 8) train acc: 0.415000; val_acc: 0.418000
(Iteration 501 / 1960) loss: 1.680884
```

```
(Epoch 3 / 8) train acc: 0.459000; val_acc: 0.445000
(Epoch 4 / 8) train acc: 0.456000; val_acc: 0.451000
(Iteration 1001 / 1960) loss: 1.648468
(Epoch 5 / 8) train acc: 0.462000; val_acc: 0.479000
(Epoch 6 / 8) train acc: 0.500000; val_acc: 0.473000
(Iteration 1501 / 1960) loss: 1.534539
(Epoch 7 / 8) train acc: 0.488000; val_acc: 0.474000
(Epoch 8 / 8) train acc: 0.507000; val_acc: 0.475000
model with lr: 0.000395 reg: 0.69 decay: 0.982
(Iteration 1 / 1960) loss: 2.357483
(Epoch 0 / 8) train acc: 0.084000; val_acc: 0.096000
(Epoch 1 / 8) train acc: 0.356000; val_acc: 0.351000
(Epoch 2 / 8) train acc: 0.386000; val_acc: 0.425000
(Iteration 501 / 1960) loss: 1.656732
(Epoch 3 / 8) train acc: 0.411000; val_acc: 0.432000
(Epoch 4 / 8) train acc: 0.464000; val_acc: 0.458000
(Iteration 1001 / 1960) loss: 1.563367
(Epoch 5 / 8) train acc: 0.471000; val_acc: 0.459000
(Epoch 6 / 8) train acc: 0.463000; val_acc: 0.460000
(Iteration 1501 / 1960) loss: 1.467282
(Epoch 7 / 8) train acc: 0.469000; val_acc: 0.463000
(Epoch 8 / 8) train acc: 0.503000; val_acc: 0.487000
model with lr: 0.0004 reg: 0.7 decay: 0.984
(Iteration 1 / 1960) loss: 2.358284
(Epoch 0 / 8) train acc: 0.139000; val_acc: 0.114000
(Epoch 1 / 8) train acc: 0.349000; val_acc: 0.367000
(Epoch 2 / 8) train acc: 0.424000; val_acc: 0.423000
(Iteration 501 / 1960) loss: 1.677587
(Epoch 3 / 8) train acc: 0.433000; val_acc: 0.437000
(Epoch 4 / 8) train acc: 0.441000; val_acc: 0.458000
(Iteration 1001 / 1960) loss: 1.655844
(Epoch 5 / 8) train acc: 0.446000; val_acc: 0.464000
(Epoch 6 / 8) train acc: 0.453000; val_acc: 0.457000
(Iteration 1501 / 1960) loss: 1.570089
(Epoch 7 / 8) train acc: 0.509000; val_acc: 0.465000
(Epoch 8 / 8) train acc: 0.485000; val_acc: 0.480000
model with lr: 0.000405 reg: 0.71 decay: 0.986
(Iteration 1 / 1960) loss: 2.357155
(Epoch 0 / 8) train acc: 0.120000; val_acc: 0.131000
(Epoch 1 / 8) train acc: 0.373000; val_acc: 0.364000
(Epoch 2 / 8) train acc: 0.393000; val_acc: 0.410000
(Iteration 501 / 1960) loss: 1.689633
(Epoch 3 / 8) train acc: 0.434000; val_acc: 0.440000
(Epoch 4 / 8) train acc: 0.467000; val_acc: 0.450000
(Iteration 1001 / 1960) loss: 1.648809
(Epoch 5 / 8) train acc: 0.462000; val_acc: 0.458000
(Epoch 6 / 8) train acc: 0.487000; val_acc: 0.471000
(Iteration 1501 / 1960) loss: 1.523355
```

```
(Epoch 7 / 8) train acc: 0.463000; val_acc: 0.469000
(Epoch 8 / 8) train acc: 0.481000; val_acc: 0.467000
model with lr: 0.00041 reg: 0.72 decay: 0.988
(Iteration 1 / 1960) loss: 2.356841
(Epoch 0 / 8) train acc: 0.123000; val_acc: 0.092000
(Epoch 1 / 8) train acc: 0.355000; val_acc: 0.357000
(Epoch 2 / 8) train acc: 0.424000; val_acc: 0.416000
(Iteration 501 / 1960) loss: 1.717587
(Epoch 3 / 8) train acc: 0.431000; val_acc: 0.432000
(Epoch 4 / 8) train acc: 0.477000; val_acc: 0.455000
(Iteration 1001 / 1960) loss: 1.658536
(Epoch 5 / 8) train acc: 0.433000; val_acc: 0.455000
(Epoch 6 / 8) train acc: 0.466000; val_acc: 0.469000
(Iteration 1501 / 1960) loss: 1.520789
(Epoch 7 / 8) train acc: 0.493000; val_acc: 0.477000
(Epoch 8 / 8) train acc: 0.478000; val_acc: 0.469000
```

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:  0.487
```

[ ]:
```
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:  0.477
```

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 : 1 - More data makes the model robust to a more varied test set, as the final weights will generalize the classes better.

3 - Increasing the regularization strength penalizes the classifier if overfits the training set, ensuring that the gap between the train and test accuracy is reduced.

features

August 26, 2023

[1]:
```
# 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/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/assignment1/cs231n/datasets
/content/drive/My Drive/cs231n/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.

[12]:
```
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.

[13]:
```
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.

[14]:
```
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.

[27]:
```
# Use the validation set to tune the learning rate and regularization strength

from cs231n.classifiers.linear_classifier import LinearSVM

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

learning_rates = [1.5e-8,1.75e-8,2e-8,2.25e-8,2.5e-8]
regularization_strengths = [4.6e5, 4.7e5,4.8e5,5e5,5.1e5]

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 i in range(0,5):
    #train model
    svm = LinearSVM()
    print("model with lr: "+str(learning_rates[i])+" reg: "+␣
 ↪str(regularization_strengths[i]))
    loss_hist = svm.train(X_train_feats, y_train,␣
 ↪learning_rate=learning_rates[i],␣
 ↪reg=regularization_strengths[i],num_iters=2500, verbose=True)
    #make preds
    y_train_pred = svm.predict(X_train_feats)
    y_val_pred = svm.predict(X_val_feats)

    #calc train/val acc
```

```
  train_acc = np.mean(y_train == y_train_pred)
  val_acc = np.mean(y_val == y_val_pred)
  if(val_acc>best_val):
    best_val = val_acc
    best_svm = svm
  #store in results
  results[(learning_rates[i], regularization_strengths[i])] =␣
 ↪(train_acc,val_acc)


# *****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)
```

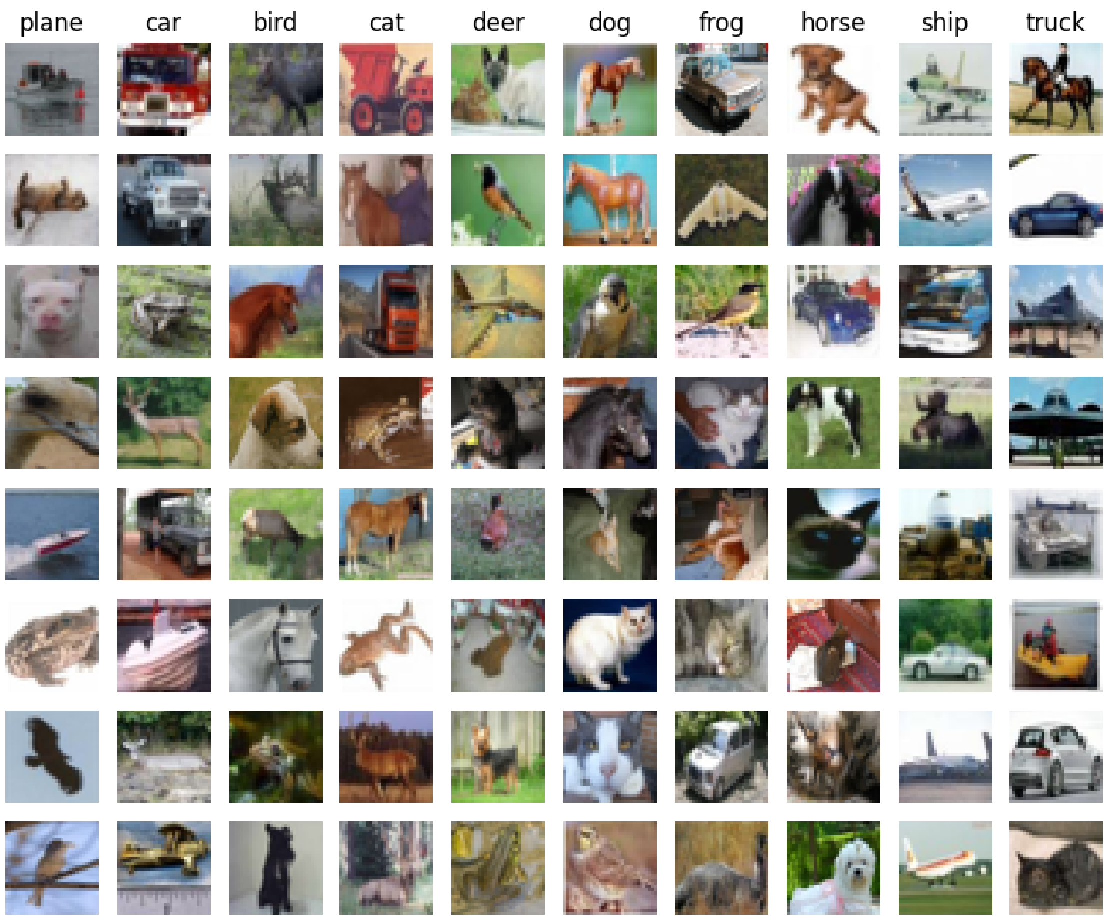
```
model with lr: 1.5e-08 reg: 460000.0
iteration 0 / 2500: loss 702.463239
iteration 100 / 2500: loss 52.057347
iteration 200 / 2500: loss 11.673427
iteration 300 / 2500: loss 9.165999
iteration 400 / 2500: loss 9.010291
iteration 500 / 2500: loss 9.000614
iteration 600 / 2500: loss 8.999992
iteration 700 / 2500: loss 8.999973
iteration 800 / 2500: loss 8.999954
iteration 900 / 2500: loss 8.999960
iteration 1000 / 2500: loss 8.999962
iteration 1100 / 2500: loss 8.999963
iteration 1200 / 2500: loss 8.999957
iteration 1300 / 2500: loss 8.999968
iteration 1400 / 2500: loss 8.999962
iteration 1500 / 2500: loss 8.999964
iteration 1600 / 2500: loss 8.999954
iteration 1700 / 2500: loss 8.999955
iteration 1800 / 2500: loss 8.999966
iteration 1900 / 2500: loss 8.999968
iteration 2000 / 2500: loss 8.999963
iteration 2100 / 2500: loss 8.999965
iteration 2200 / 2500: loss 8.999972
iteration 2300 / 2500: loss 8.999960
iteration 2400 / 2500: loss 8.999964
model with lr: 1.75e-08 reg: 470000.0
```

```
iteration 0 / 2500: loss 740.748966
iteration 100 / 2500: loss 35.523866
iteration 200 / 2500: loss 9.961782
iteration 300 / 2500: loss 9.034910
iteration 400 / 2500: loss 9.001221
iteration 500 / 2500: loss 9.000007
iteration 600 / 2500: loss 8.999964
iteration 700 / 2500: loss 8.999958
iteration 800 / 2500: loss 8.999966
iteration 900 / 2500: loss 8.999959
iteration 1000 / 2500: loss 8.999963
iteration 1100 / 2500: loss 8.999960
iteration 1200 / 2500: loss 8.999960
iteration 1300 / 2500: loss 8.999959
iteration 1400 / 2500: loss 8.999966
iteration 1500 / 2500: loss 8.999962
iteration 1600 / 2500: loss 8.999960
iteration 1700 / 2500: loss 8.999962
iteration 1800 / 2500: loss 8.999970
iteration 1900 / 2500: loss 8.999967
iteration 2000 / 2500: loss 8.999955
iteration 2100 / 2500: loss 8.999966
iteration 2200 / 2500: loss 8.999966
iteration 2300 / 2500: loss 8.999969
iteration 2400 / 2500: loss 8.999965
model with lr: 2e-08 reg: 480000.0
iteration 0 / 2500: loss 763.795961
iteration 100 / 2500: loss 24.629967
iteration 200 / 2500: loss 9.323462
iteration 300 / 2500: loss 9.006665
iteration 400 / 2500: loss 9.000103
iteration 500 / 2500: loss 8.999973
iteration 600 / 2500: loss 8.999964
iteration 700 / 2500: loss 8.999963
iteration 800 / 2500: loss 8.999966
iteration 900 / 2500: loss 8.999956
iteration 1000 / 2500: loss 8.999972
iteration 1100 / 2500: loss 8.999957
iteration 1200 / 2500: loss 8.999963
iteration 1300 / 2500: loss 8.999965
iteration 1400 / 2500: loss 8.999960
iteration 1500 / 2500: loss 8.999962
iteration 1600 / 2500: loss 8.999958
iteration 1700 / 2500: loss 8.999964
iteration 1800 / 2500: loss 8.999962
iteration 1900 / 2500: loss 8.999960
iteration 2000 / 2500: loss 8.999969
iteration 2100 / 2500: loss 8.999966
iteration 2200 / 2500: loss 8.999958
iteration 2300 / 2500: loss 8.999960
iteration 2400 / 2500: loss 8.999960
model with lr: 2.25e-08 reg: 500000.0
iteration 0 / 2500: loss 822.229771
iteration 100 / 2500: loss 17.582804
iteration 200 / 2500: loss 9.090561
iteration 300 / 2500: loss 9.000914
iteration 400 / 2500: loss 8.999965
iteration 500 / 2500: loss 8.999963
iteration 600 / 2500: loss 8.999968
iteration 700 / 2500: loss 8.999970
iteration 800 / 2500: loss 8.999966
iteration 900 / 2500: loss 8.999965
iteration 1000 / 2500: loss 8.999970
iteration 1100 / 2500: loss 8.999963
iteration 1200 / 2500: loss 8.999963
iteration 1300 / 2500: loss 8.999962
iteration 1400 / 2500: loss 8.999962
iteration 1500 / 2500: loss 8.999963
iteration 1600 / 2500: loss 8.999968
iteration 1700 / 2500: loss 8.999958
iteration 1800 / 2500: loss 8.999968
iteration 1900 / 2500: loss 8.999970
iteration 2000 / 2500: loss 8.999963
iteration 2100 / 2500: loss 8.999971
iteration 2200 / 2500: loss 8.999963
iteration 2300 / 2500: loss 8.999974
iteration 2400 / 2500: loss 8.999969
model with lr: 2.5e-08 reg: 510000.0
iteration 0 / 2500: loss 815.501200
iteration 100 / 2500: loss 13.602118
iteration 200 / 2500: loss 9.026284
iteration 300 / 2500: loss 9.000113
iteration 400 / 2500: loss 8.999970
iteration 500 / 2500: loss 8.999977
iteration 600 / 2500: loss 8.999966
iteration 700 / 2500: loss 8.999971
iteration 800 / 2500: loss 8.999960
iteration 900 / 2500: loss 8.999960
iteration 1000 / 2500: loss 8.999963
iteration 1100 / 2500: loss 8.999972
iteration 1200 / 2500: loss 8.999958
iteration 1300 / 2500: loss 8.999965
iteration 1400 / 2500: loss 8.999969
iteration 1500 / 2500: loss 8.999964
iteration 1600 / 2500: loss 8.999962
iteration 1700 / 2500: loss 8.999967
iteration 1800 / 2500: loss 8.999966
iteration 1900 / 2500: loss 8.999961
iteration 2000 / 2500: loss 8.999957
iteration 2100 / 2500: loss 8.999966
iteration 2200 / 2500: loss 8.999958
iteration 2300 / 2500: loss 8.999964
iteration 2400 / 2500: loss 8.999970
lr 1.500000e-08 reg 4.600000e+05 train accuracy: 0.410694 val accuracy: 0.416000
lr 1.750000e-08 reg 4.700000e+05 train accuracy: 0.415878 val accuracy: 0.417000
lr 2.000000e-08 reg 4.800000e+05 train accuracy: 0.415510 val accuracy: 0.422000
lr 2.250000e-08 reg 5.000000e+05 train accuracy: 0.410082 val accuracy: 0.419000
lr 2.500000e-08 reg 5.100000e+05 train accuracy: 0.413551 val accuracy: 0.422000
best validation accuracy achieved: 0.422000
```

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

```
0.43
```

[29]:
```
# An important way to gain intuition about how an algorithm works is to
# visualize the mistakes that it makes. In this visualization, we show examples
# of images that are misclassified by our current system. The first column
# shows images that our system labeled as "plane" but whose true label is
# something other than "plane".

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



1.3.1 Inline question 1:

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

Your Answer :Yes they make sense as the classifier picks the images closest in pixel space (images with common elements - eg: planes and ships both have blue backrounds and have sails/aerodynamic surfaces) so a classifier can get confused. We can avoid this by finding the class boundaries in a higher dimensional space (called feature space) then visualising it in 2-d/3-d.

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.

[34]:
```
# 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)
```

[53]:
```
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,
}

best_net = None
best_val = 0

################################################################################
# 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)*****
learning_rates = [1e-2,2e-2,3e-2,4e-2,5e-2]
regularization_strengths = [1e-4, 1e-4,1e-4,1e-4,1e-4]
#learning_rate_decay = [0.97,0.97,0.97,0.97,0.97]


for i in range(0,5):
  #train model

  model = TwoLayerNet(input_dim, hidden_dim,␣
 ↪num_classes,reg=regularization_strengths[i])
```

```
  solver = Solver(model, data,
                  update_rule='sgd',
                  optim_config={
                    'learning_rate': learning_rates[i],
                  },
                  num_epochs=20, batch_size=200,
                  print_every=500)

  print("model with lr: "+str(learning_rates[i])+" reg: "+␣
↪str(regularization_strengths[i])+ " decay: " +str(learning_rate_decay[i]))
  solver.train()
  #make preds
  y_train_pred = np.argmax(model.loss(data['X_train']), axis=1)
  y_val_pred = np.argmax(model.loss(data['X_val']), axis=1)

  #calc train/val acc
  train_acc = np.mean(data['y_train'] == y_train_pred)
  val_acc = np.mean(data['y_val'] == y_val_pred)

  if(val_acc>best_val):
    best_val = val_acc
    best_net = model
pass

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

```
model with lr: 0.01 reg: 0.0001 decay: 0.97
(Iteration 1 / 4900) loss: 2.302595
(Epoch 0 / 20) train acc: 0.113000; val_acc: 0.109000
(Epoch 1 / 20) train acc: 0.223000; val_acc: 0.229000
(Epoch 2 / 20) train acc: 0.126000; val_acc: 0.135000
(Iteration 501 / 4900) loss: 2.299839
(Epoch 3 / 20) train acc: 0.233000; val_acc: 0.234000
(Epoch 4 / 20) train acc: 0.249000; val_acc: 0.269000
(Iteration 1001 / 4900) loss: 2.232559
(Epoch 5 / 20) train acc: 0.272000; val_acc: 0.278000
(Epoch 6 / 20) train acc: 0.283000; val_acc: 0.295000
(Iteration 1501 / 4900) loss: 1.968033
(Epoch 7 / 20) train acc: 0.300000; val_acc: 0.326000
(Epoch 8 / 20) train acc: 0.346000; val_acc: 0.352000
(Iteration 2001 / 4900) loss: 1.720908
(Epoch 9 / 20) train acc: 0.399000; val_acc: 0.384000
(Epoch 10 / 20) train acc: 0.397000; val_acc: 0.398000
(Iteration 2501 / 4900) loss: 1.670844
(Epoch 11 / 20) train acc: 0.449000; val_acc: 0.417000
(Epoch 12 / 20) train acc: 0.422000; val_acc: 0.428000
```

```
(Iteration 3001 / 4900) loss: 1.454615
(Epoch 13 / 20) train acc: 0.453000; val_acc: 0.435000
(Epoch 14 / 20) train acc: 0.467000; val_acc: 0.451000
(Iteration 3501 / 4900) loss: 1.388624
(Epoch 15 / 20) train acc: 0.473000; val_acc: 0.465000
(Epoch 16 / 20) train acc: 0.500000; val_acc: 0.482000
(Iteration 4001 / 4900) loss: 1.476390
(Epoch 17 / 20) train acc: 0.512000; val_acc: 0.483000
(Epoch 18 / 20) train acc: 0.492000; val_acc: 0.491000
(Iteration 4501 / 4900) loss: 1.448513
(Epoch 19 / 20) train acc: 0.508000; val_acc: 0.502000
(Epoch 20 / 20) train acc: 0.524000; val_acc: 0.505000
model with lr: 0.02 reg: 0.0001 decay: 0.97
(Iteration 1 / 4900) loss: 2.302562
(Epoch 0 / 20) train acc: 0.112000; val_acc: 0.099000
(Epoch 1 / 20) train acc: 0.184000; val_acc: 0.163000
(Epoch 2 / 20) train acc: 0.261000; val_acc: 0.257000
(Iteration 501 / 4900) loss: 2.230282
(Epoch 3 / 20) train acc: 0.276000; val_acc: 0.314000
(Epoch 4 / 20) train acc: 0.339000; val_acc: 0.364000
(Iteration 1001 / 4900) loss: 1.778247
(Epoch 5 / 20) train acc: 0.398000; val_acc: 0.398000
(Epoch 6 / 20) train acc: 0.442000; val_acc: 0.431000
(Iteration 1501 / 4900) loss: 1.517946
(Epoch 7 / 20) train acc: 0.474000; val_acc: 0.451000
(Epoch 8 / 20) train acc: 0.481000; val_acc: 0.478000
(Iteration 2001 / 4900) loss: 1.314878
(Epoch 9 / 20) train acc: 0.484000; val_acc: 0.488000
(Epoch 10 / 20) train acc: 0.481000; val_acc: 0.500000
(Iteration 2501 / 4900) loss: 1.419105
(Epoch 11 / 20) train acc: 0.510000; val_acc: 0.503000
(Epoch 12 / 20) train acc: 0.536000; val_acc: 0.513000
(Iteration 3001 / 4900) loss: 1.354447
(Epoch 13 / 20) train acc: 0.522000; val_acc: 0.519000
(Epoch 14 / 20) train acc: 0.536000; val_acc: 0.514000
(Iteration 3501 / 4900) loss: 1.246371
(Epoch 15 / 20) train acc: 0.533000; val_acc: 0.513000
(Epoch 16 / 20) train acc: 0.507000; val_acc: 0.517000
(Iteration 4001 / 4900) loss: 1.317808
(Epoch 17 / 20) train acc: 0.533000; val_acc: 0.515000
(Epoch 18 / 20) train acc: 0.526000; val_acc: 0.521000
(Iteration 4501 / 4900) loss: 1.259956
(Epoch 19 / 20) train acc: 0.524000; val_acc: 0.520000
(Epoch 20 / 20) train acc: 0.555000; val_acc: 0.520000
model with lr: 0.03 reg: 0.0001 decay: 0.97
(Iteration 1 / 4900) loss: 2.302597
(Epoch 0 / 20) train acc: 0.100000; val_acc: 0.103000
(Epoch 1 / 20) train acc: 0.225000; val_acc: 0.245000
```

```
(Epoch 2 / 20) train acc: 0.277000; val_acc: 0.298000
(Iteration 501 / 4900) loss: 1.954633
(Epoch 3 / 20) train acc: 0.358000; val_acc: 0.376000
(Epoch 4 / 20) train acc: 0.434000; val_acc: 0.423000
(Iteration 1001 / 4900) loss: 1.659837
(Epoch 5 / 20) train acc: 0.466000; val_acc: 0.462000
(Epoch 6 / 20) train acc: 0.487000; val_acc: 0.487000
(Iteration 1501 / 4900) loss: 1.487654
(Epoch 7 / 20) train acc: 0.514000; val_acc: 0.501000
(Epoch 8 / 20) train acc: 0.532000; val_acc: 0.506000
(Iteration 2001 / 4900) loss: 1.261374
(Epoch 9 / 20) train acc: 0.515000; val_acc: 0.507000
(Epoch 10 / 20) train acc: 0.553000; val_acc: 0.514000
(Iteration 2501 / 4900) loss: 1.256254
(Epoch 11 / 20) train acc: 0.574000; val_acc: 0.520000
(Epoch 12 / 20) train acc: 0.510000; val_acc: 0.522000
(Iteration 3001 / 4900) loss: 1.374747
(Epoch 13 / 20) train acc: 0.540000; val_acc: 0.520000
(Epoch 14 / 20) train acc: 0.538000; val_acc: 0.517000
(Iteration 3501 / 4900) loss: 1.253474
(Epoch 15 / 20) train acc: 0.541000; val_acc: 0.523000
(Epoch 16 / 20) train acc: 0.559000; val_acc: 0.534000
(Iteration 4001 / 4900) loss: 1.326269
(Epoch 17 / 20) train acc: 0.522000; val_acc: 0.526000
(Epoch 18 / 20) train acc: 0.589000; val_acc: 0.538000
(Iteration 4501 / 4900) loss: 1.186855
(Epoch 19 / 20) train acc: 0.544000; val_acc: 0.541000
(Epoch 20 / 20) train acc: 0.600000; val_acc: 0.543000
model with lr: 0.04 reg: 0.0001 decay: 0.97
(Iteration 1 / 4900) loss: 2.302584
(Epoch 0 / 20) train acc: 0.090000; val_acc: 0.106000
(Epoch 1 / 20) train acc: 0.237000; val_acc: 0.237000
(Epoch 2 / 20) train acc: 0.341000; val_acc: 0.335000
(Iteration 501 / 4900) loss: 1.831810
(Epoch 3 / 20) train acc: 0.429000; val_acc: 0.420000
(Epoch 4 / 20) train acc: 0.514000; val_acc: 0.475000
(Iteration 1001 / 4900) loss: 1.550674
(Epoch 5 / 20) train acc: 0.479000; val_acc: 0.497000
(Epoch 6 / 20) train acc: 0.491000; val_acc: 0.503000
(Iteration 1501 / 4900) loss: 1.419282
(Epoch 7 / 20) train acc: 0.520000; val_acc: 0.510000
(Epoch 8 / 20) train acc: 0.504000; val_acc: 0.514000
(Iteration 2001 / 4900) loss: 1.245271
(Epoch 9 / 20) train acc: 0.508000; val_acc: 0.507000
(Epoch 10 / 20) train acc: 0.530000; val_acc: 0.514000
(Iteration 2501 / 4900) loss: 1.365764
(Epoch 11 / 20) train acc: 0.536000; val_acc: 0.521000
(Epoch 12 / 20) train acc: 0.572000; val_acc: 0.534000
```

```
(Iteration 3001 / 4900) loss: 1.264313
(Epoch 13 / 20) train acc: 0.543000; val_acc: 0.538000
(Epoch 14 / 20) train acc: 0.532000; val_acc: 0.542000
(Iteration 3501 / 4900) loss: 1.110682
(Epoch 15 / 20) train acc: 0.554000; val_acc: 0.539000
(Epoch 16 / 20) train acc: 0.576000; val_acc: 0.545000
(Iteration 4001 / 4900) loss: 1.272701
(Epoch 17 / 20) train acc: 0.537000; val_acc: 0.550000
(Epoch 18 / 20) train acc: 0.557000; val_acc: 0.559000
(Iteration 4501 / 4900) loss: 1.232045
(Epoch 19 / 20) train acc: 0.587000; val_acc: 0.566000
(Epoch 20 / 20) train acc: 0.591000; val_acc: 0.566000
model with lr: 0.05 reg: 0.0001 decay: 0.97
(Iteration 1 / 4900) loss: 2.302608
(Epoch 0 / 20) train acc: 0.102000; val_acc: 0.078000
(Epoch 1 / 20) train acc: 0.240000; val_acc: 0.254000
(Epoch 2 / 20) train acc: 0.440000; val_acc: 0.411000
(Iteration 501 / 4900) loss: 1.561863
(Epoch 3 / 20) train acc: 0.489000; val_acc: 0.466000
(Epoch 4 / 20) train acc: 0.523000; val_acc: 0.497000
(Iteration 1001 / 4900) loss: 1.463038
(Epoch 5 / 20) train acc: 0.508000; val_acc: 0.515000
(Epoch 6 / 20) train acc: 0.531000; val_acc: 0.512000
(Iteration 1501 / 4900) loss: 1.285360
(Epoch 7 / 20) train acc: 0.533000; val_acc: 0.519000
(Epoch 8 / 20) train acc: 0.561000; val_acc: 0.523000
(Iteration 2001 / 4900) loss: 1.272114
(Epoch 9 / 20) train acc: 0.586000; val_acc: 0.529000
(Epoch 10 / 20) train acc: 0.553000; val_acc: 0.542000
(Iteration 2501 / 4900) loss: 1.273705
(Epoch 11 / 20) train acc: 0.546000; val_acc: 0.539000
(Epoch 12 / 20) train acc: 0.549000; val_acc: 0.551000
(Iteration 3001 / 4900) loss: 1.373189
(Epoch 13 / 20) train acc: 0.558000; val_acc: 0.560000
(Epoch 14 / 20) train acc: 0.597000; val_acc: 0.559000
(Iteration 3501 / 4900) loss: 1.123090
(Epoch 15 / 20) train acc: 0.588000; val_acc: 0.555000
(Epoch 16 / 20) train acc: 0.593000; val_acc: 0.557000
(Iteration 4001 / 4900) loss: 1.163375
(Epoch 17 / 20) train acc: 0.608000; val_acc: 0.564000
(Epoch 18 / 20) train acc: 0.604000; val_acc: 0.567000
(Iteration 4501 / 4900) loss: 1.035225
(Epoch 19 / 20) train acc: 0.596000; val_acc: 0.575000
(Epoch 20 / 20) train acc: 0.622000; val_acc: 0.574000
```

[54]:

```
# 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)
```

```
0.564
```