knn

March 19, 2024

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

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

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

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

```
Mounted at /content/drive
/content/drive/My Drive/cs231n/assignments/assignment1/cs231n/datasets
--2024-03-06 02:39:06--  http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
Resolving www.cs.toronto.edu (www.cs.toronto.edu)… 128.100.3.30
Connecting to www.cs.toronto.edu (www.cs.toronto.edu)|128.100.3.30|:80…
connected.
HTTP request sent, awaiting response… 200 OK
Length: 170498071 (163M) [application/x-gzip]
Saving to: ‘cifar-10-python.tar.gz’

cifar-10-python.tar 100%[===================>] 162.60M  51.2MB/s    in 3.4s

2024-03-06 02:39:10 (47.3 MB/s) - ‘cifar-10-python.tar.gz’ saved
[170498071/170498071]

cifar-10-batches-py/
cifar-10-batches-py/data_batch_4
```

```
cifar-10-batches-py/readme.html
cifar-10-batches-py/test_batch
cifar-10-batches-py/data_batch_3
cifar-10-batches-py/batches.meta
cifar-10-batches-py/data_batch_2
cifar-10-batches-py/data_batch_5
cifar-10-batches-py/data_batch_1
--2024-03-06 02:39:14--  http://cs231n.stanford.edu/imagenet_val_25.npz
Resolving cs231n.stanford.edu (cs231n.stanford.edu)… 171.64.68.10
Connecting to cs231n.stanford.edu (cs231n.stanford.edu)|171.64.68.10|:80…
connected.
HTTP request sent, awaiting response… 200 OK
Length: 3940548 (3.8M) [text/plain]
Saving to: ‘imagenet_val_25.npz’

imagenet_val_25.npz 100%[===================>]   3.76M  4.61MB/s    in 0.8s

2024-03-06 02:39:15 (4.61 MB/s) - ‘imagenet_val_25.npz’ saved [3940548/3940548]

/content/drive/My Drive/cs231n/assignments/assignment1
```

1 k-Nearest Neighbor (kNN) exercise

Complete and hand in this completed worksheet (including its outputs and any supporting code outside of the worksheet) with your assignment submission. For more details see the assignments page on the course website.

The kNN classifier consists of two stages:

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

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

[ ]:
```
# 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
```

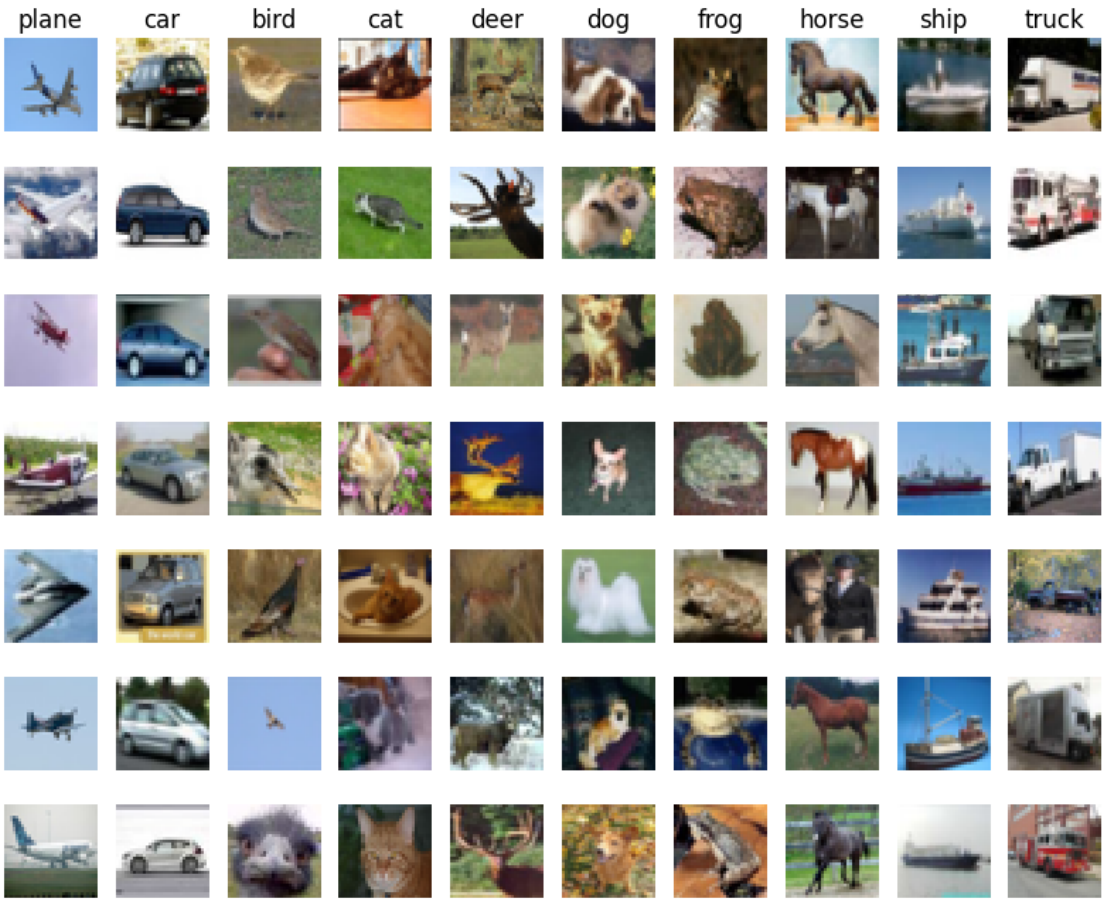
```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

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

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

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

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

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

[ ]:
```
# Visualize some examples from the dataset.
# We show a few examples of training images from each class.
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
  ↪'ship', 'truck']
num_classes = len(classes)
samples_per_class = 7
for y, cls in enumerate(classes):
    idxs = np.flatnonzero(y_train == y)
    idxs = np.random.choice(idxs, samples_per_class, replace=False)
```

```
    for i, idx in enumerate(idxs):
        plt_idx = i * num_classes + y + 1
        plt.subplot(samples_per_class, num_classes, plt_idx)
        plt.imshow(X_train[idx].astype('uint8'))
        plt.axis('off')
        if i == 0:
            plt.title(cls)
plt.show()
```



[ ]:
```
# 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)
```

[ ]:
```
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.

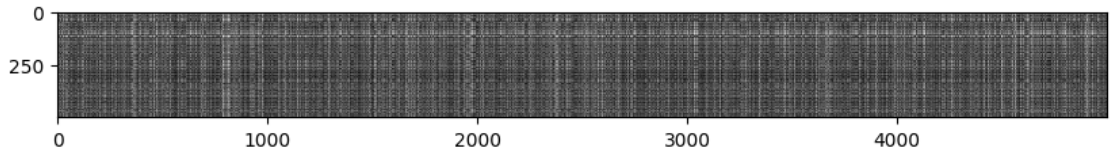
Note: For the three distance computations that we require you to implement in this notebook, you may not use the np.linalg.norm() function that numpy provides.

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.

[ ]:
```
# 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)
```

[ ]:
```
# 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 test data of this row is not similar to most of train data. - The train data of this row is not similar to most of test data.

[28]:
```
# Now implement the function predict_labels and run the code below:
# We use k = 1 (which is Nearest Neighbor).
y_test_pred = classifier.predict_labels(dists, k=1)

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

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

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

[34]:
```
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.

YourAnswer : 1,2,5

YourExplanation :

[38]:
```
# 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
```

[39]:
```
# 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
```

[40]:
```
# 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 34.606154 seconds
One loop version took 36.689680 seconds
No loop version took 0.546089 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.

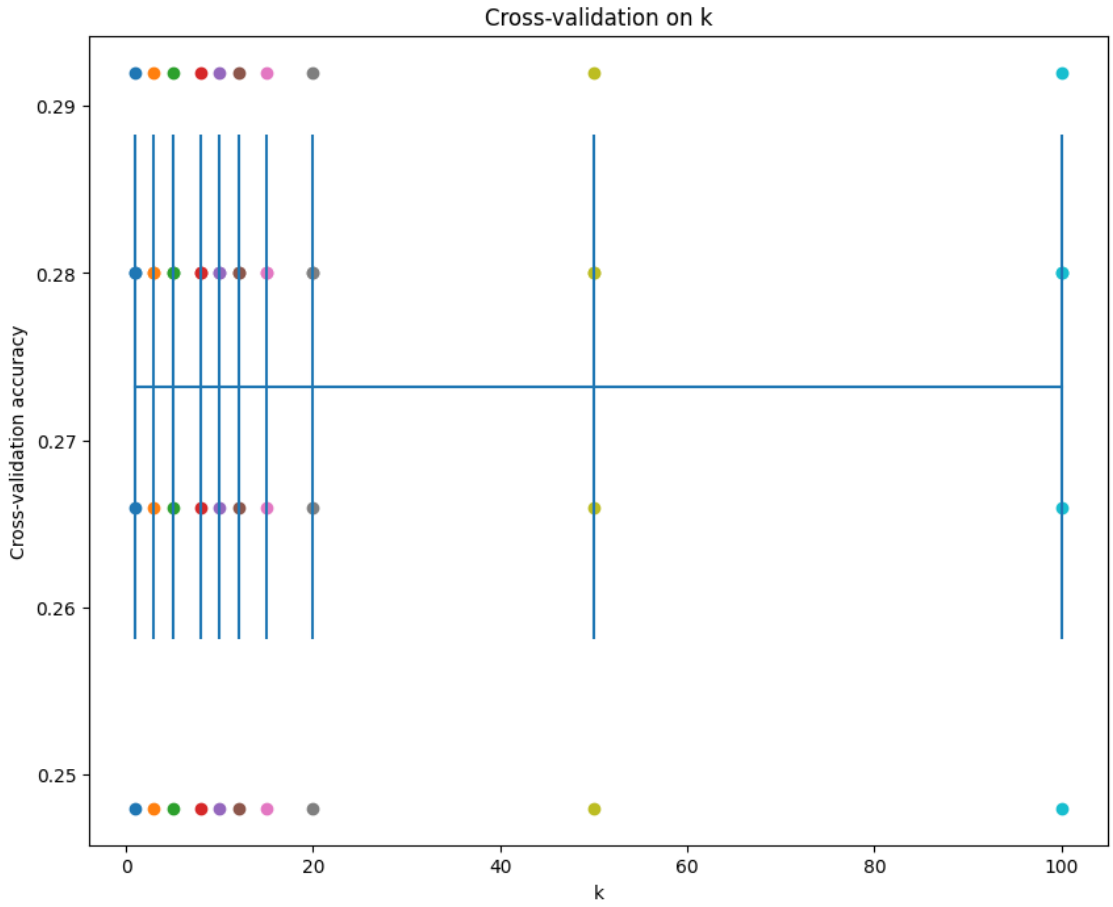
[55]:
```
num_folds = 5
k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50, 100]

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

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


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

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


################################################################################
# TODO:                                                                        #
# Perform k-fold cross validation to find the best value of k. For each        #
# possible value of k, run the k-nearest-neighbor algorithm num_folds times,   #
# where in each case you use all but one of the folds as training data and the #
# last fold as a validation set. Store the accuracies for all fold and all     #
# values of k in the k_to_accuracies dictionary.                               #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

for k in k_choices:
  accuracies = []

  for i in range(num_folds):
    X_test_cv = X_train_folds[i]
    y_test_cv = y_train_folds[i]
    X_train_cv = np.vstack(X_train_folds[0:i] + X_train_folds[i+1:])
    y_train_cv = np.hstack(y_train_folds[0:i] + y_train_folds[i+1:])
    num_valid = y_test_cv.shape[0]
    classifier.train(X_train_cv, y_train_cv)
```

```
    dists = classifier.compute_distances_no_loops(X_test_cv)
    y_test_pred = classifier.predict_labels(dists, k=5)
    num_correct = np.sum(y_test_pred == y_test_cv)
    accuracy = float(num_correct) / num_valid
    accuracies.append(accuracy)
  k_to_accuracies[k] = accuracies

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

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

```
k = 1, accuracy = 0.248000
k = 1, accuracy = 0.266000
k = 1, accuracy = 0.280000
k = 1, accuracy = 0.292000
k = 1, accuracy = 0.280000
k = 3, accuracy = 0.248000
k = 3, accuracy = 0.266000
k = 3, accuracy = 0.280000
k = 3, accuracy = 0.292000
k = 3, accuracy = 0.280000
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 = 0.248000
k = 8, accuracy = 0.266000
k = 8, accuracy = 0.280000
k = 8, accuracy = 0.292000
k = 8, accuracy = 0.280000
k = 10, accuracy = 0.248000
k = 10, accuracy = 0.266000
k = 10, accuracy = 0.280000
k = 10, accuracy = 0.292000
k = 10, accuracy = 0.280000
k = 12, accuracy = 0.248000
k = 12, accuracy = 0.266000
k = 12, accuracy = 0.280000
k = 12, accuracy = 0.292000
k = 12, accuracy = 0.280000
k = 15, accuracy = 0.248000
k = 15, accuracy = 0.266000
k = 15, accuracy = 0.280000
```

```
k = 15, accuracy = 0.292000
k = 15, accuracy = 0.280000
k = 20, accuracy = 0.248000
k = 20, accuracy = 0.266000
k = 20, accuracy = 0.280000
k = 20, accuracy = 0.292000
k = 20, accuracy = 0.280000
k = 50, accuracy = 0.248000
k = 50, accuracy = 0.266000
k = 50, accuracy = 0.280000
k = 50, accuracy = 0.292000
k = 50, accuracy = 0.280000
k = 100, accuracy = 0.248000
k = 100, accuracy = 0.266000
k = 100, accuracy = 0.280000
k = 100, accuracy = 0.292000
k = 100, accuracy = 0.280000
```

[47]:
```
# 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()
```



[48]:
```
# Based on the cross-validation results above, choose the best value for k,
# retrain the classifier using all the training data, and test it on the test
# data. You should be able to get above 28% accuracy on the test data.
best_k = 1

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

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

```
Got 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 :

svm

March 19, 2024

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

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

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

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

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

1 Multiclass Support Vector Machine exercise

Complete and hand in this completed worksheet (including its outputs and any supporting code outside of the worksheet) with your assignment submission. For more details see the assignments page on the course website.

In this exercise you will:

- implement a fully-vectorized **loss function** for the SVM
- implement the fully-vectorized expression for its **analytic gradient**
- **check your implementation** using numerical gradient
- use a validation set to **tune the learning rate and regularization** strength
- **optimize** the loss function with **SGD**

- **visualize** the final learned weights

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

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

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

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

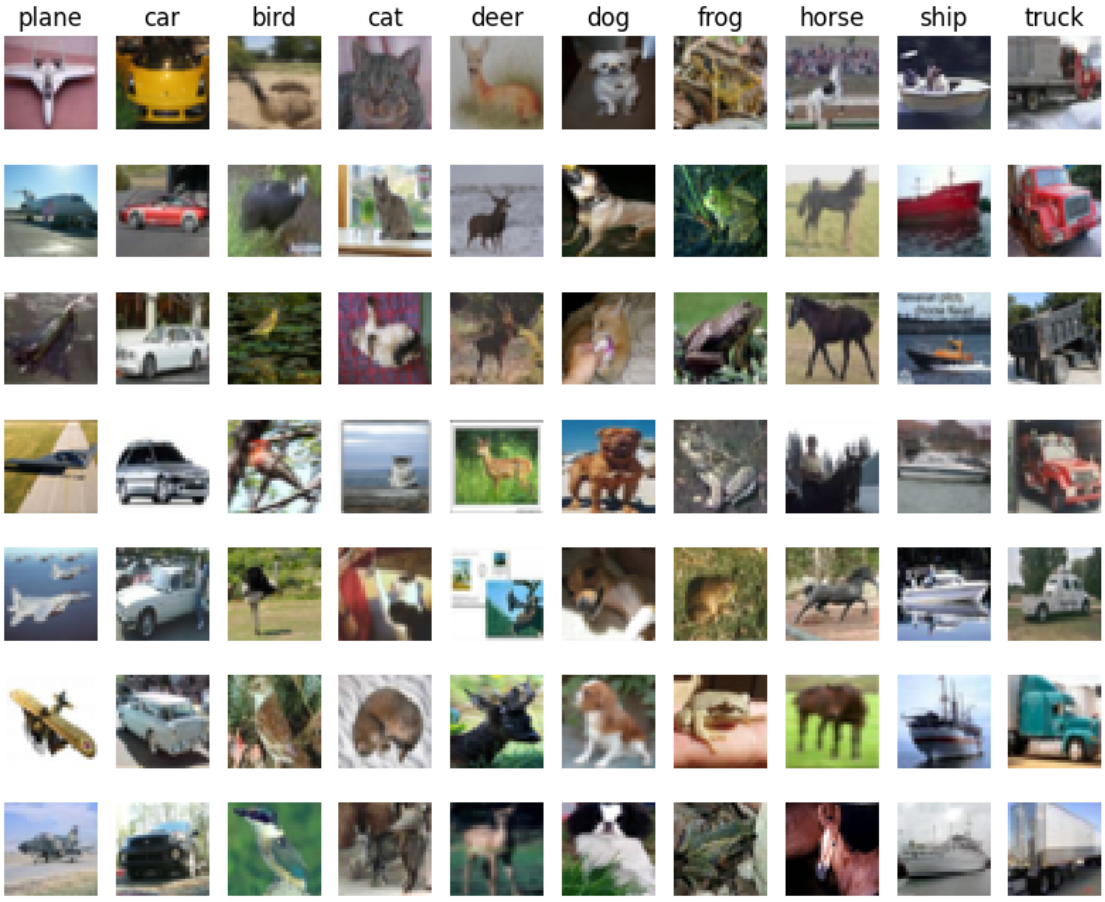
1.1 CIFAR-10 Data Loading and Preprocessing

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

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

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

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

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

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

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



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

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

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

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

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

print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

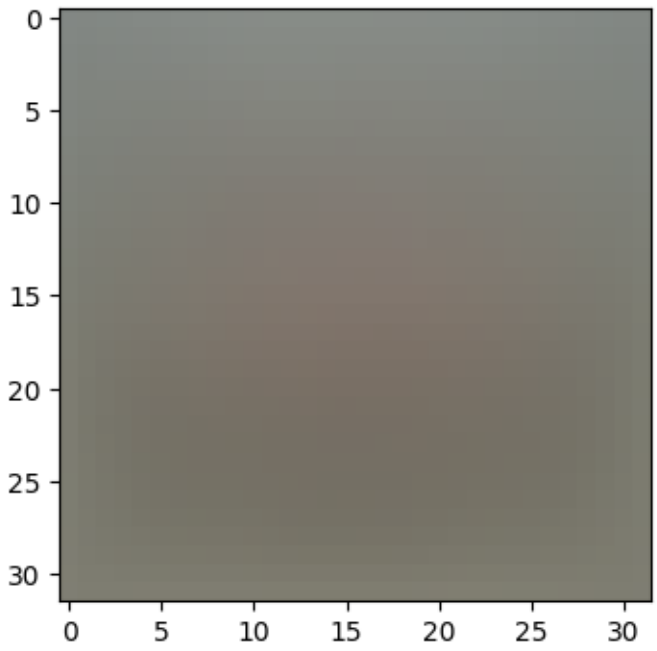
```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

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

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

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

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

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

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

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

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




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

1.2 SVM Classifier

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

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

[ ]:
```
# Evaluate the naive implementation of the loss we provided for you:
from cs231n.classifiers.linear_svm import svm_loss_naive
import time

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

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

```
loss: 8.859420
```

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

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

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

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

# Numerically compute the gradient along several randomly chosen dimensions, and
# compare them with your analytically computed gradient. The numbers should␣
 ↪match
# almost exactly along all dimensions.
from cs231n.gradient_check import grad_check_sparse
f = lambda w: svm_loss_naive(w, X_dev, y_dev, 0.0)[0]
grad_numerical = grad_check_sparse(f, W, grad)

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

```
numerical: 19.470268 analytic: 19.470268, relative error: 2.627375e-11
numerical: 0.877460 analytic: 0.877460, relative error: 2.060599e-10
numerical: 11.327523 analytic: 11.327523, relative error: 6.177052e-13
numerical: 0.582139 analytic: 0.582139, relative error: 9.589944e-10
numerical: 23.416507 analytic: 23.416507, relative error: 5.149548e-12
numerical: -15.853437 analytic: -15.853437, relative error: 1.101388e-11
numerical: 30.830512 analytic: 30.830512, relative error: 9.933029e-12
numerical: 11.695249 analytic: 11.695249, relative error: 2.753963e-11
numerical: -12.676781 analytic: -12.676781, relative error: 7.615612e-12
numerical: -3.031075 analytic: -3.031075, relative error: 1.216404e-10
numerical: 3.086279 analytic: 3.150939, relative error: 1.036682e-02
numerical: -19.920741 analytic: -19.920741, relative error: 1.062145e-11
numerical: 6.463232 analytic: 6.463232, relative error: 3.534584e-12
numerical: 21.800024 analytic: 21.800024, relative error: 2.049164e-12
numerical: -4.403479 analytic: -4.403479, relative error: 7.288765e-11
numerical: 1.063238 analytic: 1.063238, relative error: 3.541496e-11
numerical: -6.619567 analytic: -6.619567, relative error: 2.826424e-11
numerical: 7.085809 analytic: 7.111561, relative error: 1.813861e-03
numerical: 18.653016 analytic: 18.699660, relative error: 1.248732e-03
numerical: 3.796448 analytic: 3.796448, relative error: 1.292778e-10
```

Inline Question 1

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

YourAnswer : The SVM loss function is not differentiable at zero. So grad check will fail ar zero.

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

from cs231n.classifiers.linear_svm import svm_loss_vectorized
tic = time.time()
loss_vectorized, _ = svm_loss_vectorized(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Vectorized loss: %e computed in %fs' % (loss_vectorized, toc - tic))

# The losses should match but your vectorized implementation should be much faster.
print('difference: %f' % (loss_naive - loss_vectorized))
```

```
Naive loss: 8.859420e+00 computed in 0.248353s
Vectorized loss: 8.859420e+00 computed in 0.024733s
difference: -0.000000
```

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

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

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

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

```
Naive loss and gradient: computed in 0.201385s
Vectorized loss and gradient: computed in 0.025841s
difference: 0.000000
```

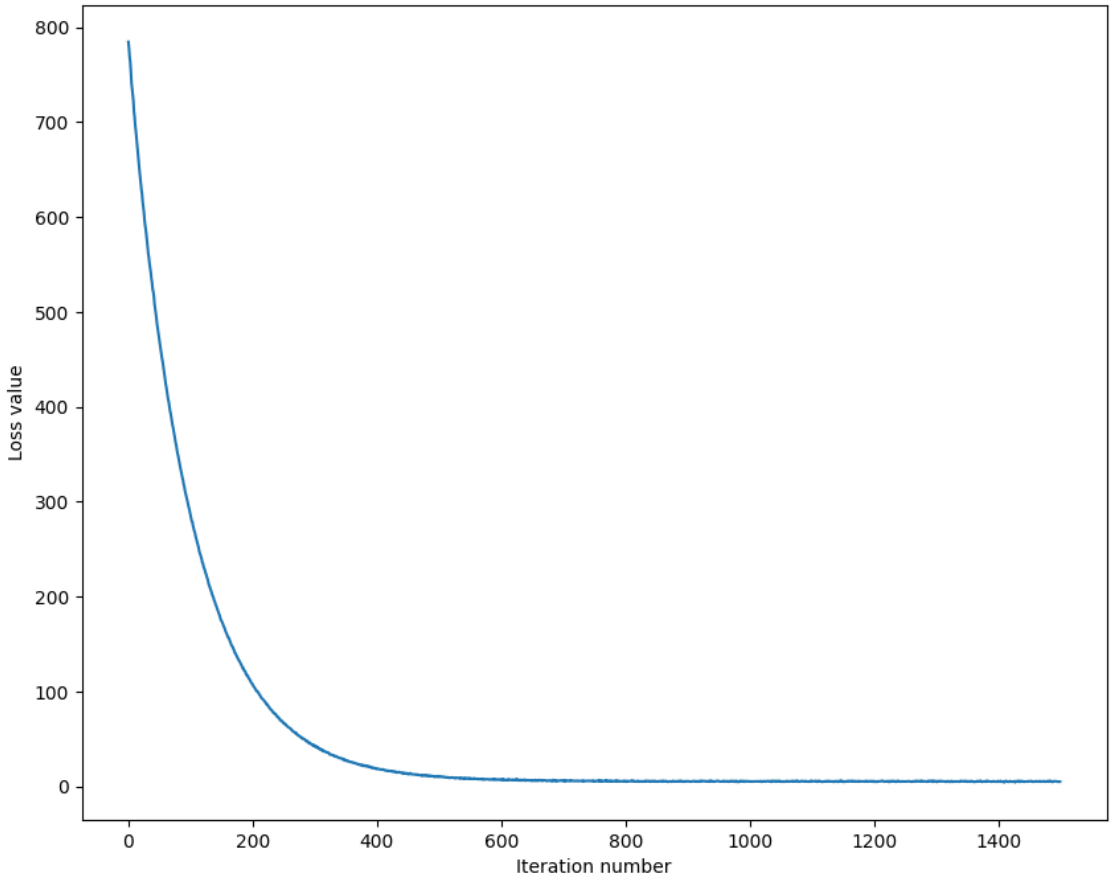
1.2.1 Stochastic Gradient Descent

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

[ ]:
```
# In the file linear_classifier.py, implement SGD in the function
# LinearClassifier.train() and then run it with the code below.
from cs231n.classifiers import LinearSVM
svm = LinearSVM()
tic = time.time()
loss_hist = svm.train(X_train, y_train, learning_rate=1e-7, reg=2.5e4,
                      num_iters=1500, verbose=True)
toc = time.time()
print('That took %fs' % (toc - tic))
```

```
iteration 0 / 1500: loss 784.654223
iteration 100 / 1500: loss 285.273528
iteration 200 / 1500: loss 107.164650
iteration 300 / 1500: loss 42.235987
iteration 400 / 1500: loss 18.380694
iteration 500 / 1500: loss 10.343449
iteration 600 / 1500: loss 7.117850
iteration 700 / 1500: loss 6.677269
iteration 800 / 1500: loss 5.351055
iteration 900 / 1500: loss 5.527846
iteration 1000 / 1500: loss 5.539836
iteration 1100 / 1500: loss 5.617272
iteration 1200 / 1500: loss 5.443100
iteration 1300 / 1500: loss 5.635666
iteration 1400 / 1500: loss 5.385941
That took 8.951911s
```

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



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

```
training accuracy: 0.365327
validation accuracy: 0.385000
```

[ ]:
```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of about 0.39 (> 0.385) on the validation set.

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

# results is dictionary mapping tuples of the form
```

```
# (learning_rate, regularization_strength) to tuples of the form
# (training_accuracy, validation_accuracy). The accuracy is simply the fraction
# of data points that are correctly classified.
results = {}
best_val = -1   # The highest validation accuracy that we have seen so far.
best_svm = None # The LinearSVM object that achieved the highest validation␣
 ↪rate.

################################################################################
# TODO:                                                                        #
# Write code that chooses the best hyperparameters by tuning on the validation #
# set. For each combination of hyperparameters, train a linear SVM on the      #
# training set, compute its accuracy on the training and validation sets, and  #
# store these numbers in the results dictionary. In addition, store the best   #
# validation accuracy in best_val and the LinearSVM object that achieves this  #
# accuracy in best_svm.                                                        #
#                                                                              #
# Hint: You should use a small value for num_iters as you develop your         #
# validation code so that the SVMs don't take much time to train; once you are #
# confident that your validation code works, you should rerun the validation   #
# code with a larger value for num_iters.                                      #
################################################################################

# Provided as a reference. You may or may not want to change these␣
 ↪hyperparameters
learning_rates = [1e-7, 5e-5]
regularization_strengths = [2.5e4, 5e4]

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

for lr in learning_rates:
    for rs in regularization_strengths:
        svm = LinearSVM()
        loss_hist = svm.train(X_train, y_train, learning_rate=lr, reg=rs,
                              num_iters=1500, verbose=True)
        y_train_pred = svm.predict(X_train)
        train_acc = np.mean(y_train == y_train_pred);
        y_val_pred = svm.predict(X_val)
        val_acc = np.mean(y_val == y_val_pred);
        results[(lr,rs)] = (train_acc,val_acc)
        if val_acc > best_val:
          best_val = val_acc
          best_svm = svm
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

# Print out results.
for lr, reg in sorted(results):
```

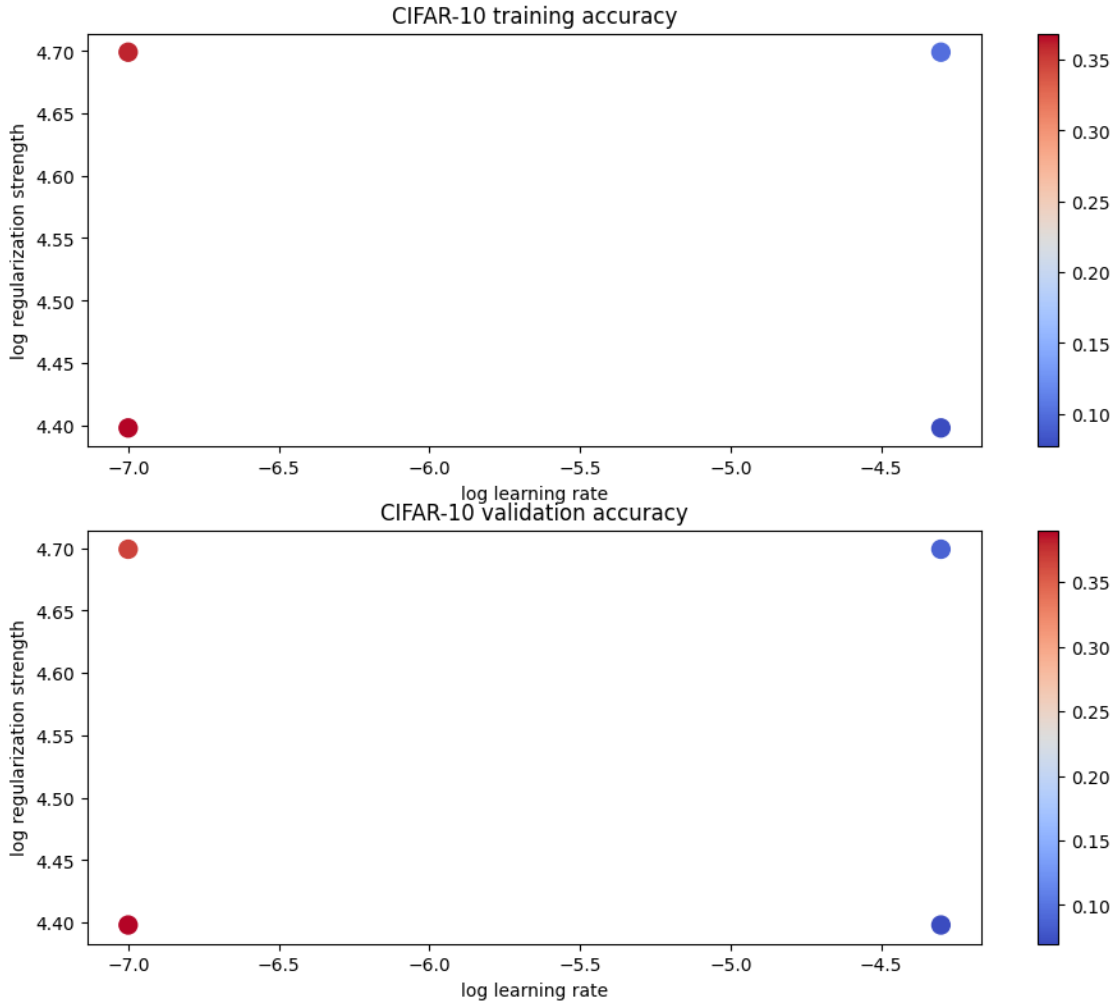
```
    train_accuracy, val_accuracy = results[(lr, reg)]
    print('lr %e reg %e train accuracy: %f val accuracy: %f' % (
                lr, reg, train_accuracy, val_accuracy))

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

```
iteration 0 / 1500: loss 792.970005
iteration 100 / 1500: loss 288.619050
iteration 200 / 1500: loss 108.250582
iteration 300 / 1500: loss 42.907838
iteration 400 / 1500: loss 19.197884
iteration 500 / 1500: loss 9.908823
iteration 600 / 1500: loss 7.261680
iteration 700 / 1500: loss 5.609485
iteration 800 / 1500: loss 6.310782
iteration 900 / 1500: loss 5.398565
iteration 1000 / 1500: loss 5.660579
iteration 1100 / 1500: loss 5.466796
iteration 1200 / 1500: loss 5.286075
iteration 1300 / 1500: loss 4.983028
iteration 1400 / 1500: loss 5.319333
iteration 0 / 1500: loss 1554.658253
iteration 100 / 1500: loss 210.883542
iteration 200 / 1500: loss 32.789331
iteration 300 / 1500: loss 9.591418
iteration 400 / 1500: loss 5.536076
iteration 500 / 1500: loss 6.011186
iteration 600 / 1500: loss 5.695168
iteration 700 / 1500: loss 5.405209
iteration 800 / 1500: loss 6.022510
iteration 900 / 1500: loss 5.627982
iteration 1000 / 1500: loss 5.554128
iteration 1100 / 1500: loss 5.920214
iteration 1200 / 1500: loss 5.949302
iteration 1300 / 1500: loss 5.841053
iteration 1400 / 1500: loss 5.577686
iteration 0 / 1500: loss 786.949592
iteration 100 / 1500: loss 42217010421549062052422346791804574105 6.000000
iteration 200 / 1500: loss 69781337099732849630306418147750068587194270906322505
969880618525765140480.000000
iteration 300 / 1500: loss 11534296149357411708001771067432744360585580694626311
639218796010596652447600501082294968853012003338321920000.000000
iteration 400 / 1500: loss 19065267762200930585073939101708809094538426601632274
029413495992406538767233114010695392851344140059579174464647062786285656241972 07
083800592384.000000
iteration 500 / 1500: loss 31513360688650966458225671804256678752195872607455981
```

```
640743135362899395634988054783638122672400429610887718323148253221191349441603969853291852370885138665274343116660587639917772 8.000000
iteration 600 / 1500: loss 52089061338122403831545700659922675899583638836524105662515986511348635146769624891995021459428815371072277248538112079786478622695679925269754891727477881525124511847206737586177194768147247331526676864184136761344.000000
iteration 700 / 1500: loss 86099046620052147697347248683765030460572998252911495743962631344703439172962312183846377897964511867714511190865641394103554192102673947760282641900222098104975080175816225152853561718785564642275132717089273050083208022914145712062564041408894730 24.000000
iteration 800 / 1500: loss 14231482845816864338793605900551211501756538752708132416972242335399850671501405670746738440454301770788629074609528520839650670379571685234956661033044887445396890780128845667425782370929486986063923431773536550465367479005474403317768606449048109233128259015078636390125198857207808 00.000000
iteration 900 / 1500: loss inf
iteration 1000 / 1500: loss inf
iteration 1100 / 1500: loss inf
iteration 1200 / 1500: loss inf
iteration 1300 / 1500: loss inf
iteration 1400 / 1500: loss inf
iteration 0 / 1500: loss 1569.975247
iteration 100 / 1500: loss 42549824035216895869564348800621078993826586685773581273088966624276386655677742685282535158255503883718325450405825174568 96.000000
iteration 200 / 1500: loss 10987427792755824225972947568259031305692292250883421951172590457777955649496006908430296352704569033102098540922130400580885806255497787521457041589783117567526116759094374792585729527907146403977370632405158669656130511045086268299346125520896.000000
iteration 300 / 1500: loss inf
iteration 400 / 1500: loss inf
iteration 500 / 1500: loss inf
iteration 600 / 1500: loss nan
iteration 700 / 1500: loss nan
iteration 800 / 1500: loss nan
iteration 900 / 1500: loss nan
iteration 1000 / 1500: loss nan
iteration 1100 / 1500: loss nan
iteration 1200 / 1500: loss nan
iteration 1300 / 1500: loss nan
iteration 1400 / 1500: loss nan
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.368571 val accuracy: 0.390000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.360347 val accuracy: 0.367000
lr 5.000000e-05 reg 2.500000e+04 train accuracy: 0.076633 val accuracy: 0.070000
lr 5.000000e-05 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
best validation accuracy achieved during cross-validation: 0.390000
```

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

```
import pdb

# pdb.set_trace()

x_scatter = [math.log10(x[0]) for x in results]
y_scatter = [math.log10(x[1]) for x in results]

# plot training accuracy
marker_size = 100
colors = [results[x][0] for x in results]
plt.subplot(2, 1, 1)
plt.tight_layout(pad=3)
plt.scatter(x_scatter, y_scatter, marker_size, c=colors, cmap=plt.cm.coolwarm)
plt.colorbar()
plt.xlabel('log learning rate')
plt.ylabel('log regularization strength')
plt.title('CIFAR-10 training accuracy')

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



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

```
linear SVM on raw pixels final test set accuracy: 0.367000
```

[ ]:
```
# Visualize the learned weights for each class.
# Depending on your choice of learning rate and regularization strength, these␣
 ↪may
# or may not be nice to look at.
w = best_svm.W[:-1,:] # strip out the bias
w = w.reshape(32, 32, 3, 10)
w_min, w_max = np.min(w), np.max(w)
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
 ↪'ship', 'truck']
```

```
for i in range(10):
    plt.subplot(2, 5, i + 1)

    # Rescale the weights to be between 0 and 255
    wimg = 255.0 * (w[:, :, :, i].squeeze() - w_min) / (w_max - w_min)
    plt.imshow(wimg.astype('uint8'))
    plt.axis('off')
    plt.title(classes[i])
```

Inline question 2

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

YourAnswer : The SVM weights are like templates for the corresponding classes.

softmax

March 19, 2024

[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/assignments/assignment1/'
assert FOLDERNAME is not None, "[!] Enter the foldername."

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

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

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

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

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

1 Softmax exercise

Complete and hand in this completed worksheet (including its outputs and any supporting code outside of the worksheet) with your assignment submission. For more details see the assignments page on the course website.

This exercise is analogous to the SVM exercise. You will:

- implement a fully-vectorized **loss function** for the Softmax classifier
- implement the fully-vectorized expression for its **analytic gradient**
- **check your implementation** with numerical gradient
- use a validation set to **tune the learning rate and regularization** strength
- **optimize** the loss function with **SGD**
- **visualize** the final learned weights

[3]:
```
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

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

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

[4]:
```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000,␣
 ↪num_dev=500):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    it for the linear classifier. These are the same steps as we used for the
    SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

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

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

    # subsample the data
    mask = list(range(num_training, num_training + num_validation))
    X_val = X_train[mask]
```

```
    y_val = y_train[mask]
    mask = list(range(num_training))
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = list(range(num_test))
    X_test = X_test[mask]
    y_test = y_test[mask]
    mask = np.random.choice(num_training, num_dev, replace=False)
    X_dev = X_train[mask]
    y_dev = y_train[mask]

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

    # Normalize the data: subtract the mean image
    mean_image = np.mean(X_train, axis = 0)
    X_train -= mean_image
    X_val -= mean_image
    X_test -= mean_image
    X_dev -= mean_image

    # add bias dimension and transform into columns
    X_train = np.hstack([X_train, np.ones((X_train.shape[0], 1))])
    X_val = np.hstack([X_val, np.ones((X_val.shape[0], 1))])
    X_test = np.hstack([X_test, np.ones((X_test.shape[0], 1))])
    X_dev = np.hstack([X_dev, np.ones((X_dev.shape[0], 1))])

    return X_train, y_train, X_val, y_val, X_test, y_test, X_dev, y_dev


# Invoke the above function to get our data.
X_train, y_train, X_val, y_val, X_test, y_test, X_dev, y_dev =␣
 ↪get_CIFAR10_data()
print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
print('dev data shape: ', X_dev.shape)
print('dev labels shape: ', y_dev.shape)
```

```
Train data shape:  (49000, 3073)
Train labels shape:  (49000,)
```

```
Validation data shape:  (1000, 3073)
Validation labels shape:  (1000,)
Test data shape:  (1000, 3073)
Test labels shape:  (1000,)
dev data shape:  (500, 3073)
dev labels shape:  (500,)
```

1.1 Softmax Classifier

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

[5]:
```
# 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: 2.311215
sanity check: 2.302585
```

Inline Question 1

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

YourAnswer : At initialization all s will be approximately equal. Loss = -log(1/C) = -log(1/10) (The regularization term in the loss is ignored)

[12]:
```
# Complete the implementation of softmax_loss_naive and implement a (naive)
# version of the gradient that uses nested loops.
loss, grad = softmax_loss_naive(W, X_dev, y_dev, 0.0)

# As we did for the SVM, use numeric gradient checking as a debugging tool.
# The numeric gradient should be close to the analytic gradient.
from cs231n.gradient_check import grad_check_sparse
f = lambda w: softmax_loss_naive(w, X_dev, y_dev, 0.0)[0]
grad_numerical = grad_check_sparse(f, W, grad, 10)

# similar to SVM case, do another gradient check with regularization
loss, grad = softmax_loss_naive(W, X_dev, y_dev, 5e1)
f = lambda w: softmax_loss_naive(w, X_dev, y_dev, 5e1)[0]
grad_numerical = grad_check_sparse(f, W, grad, 10)
```

```
numerical: 0.681730 analytic: 0.681730, relative error: 8.908594e-09
numerical: -2.259816 analytic: -2.259816, relative error: 9.870641e-09
numerical: 0.109759 analytic: 0.109759, relative error: 2.477685e-07
numerical: -1.062487 analytic: -1.062487, relative error: 3.718729e-08
numerical: 0.497654 analytic: 0.497654, relative error: 8.316044e-08
numerical: 0.578221 analytic: 0.578221, relative error: 2.557639e-09
numerical: -0.308373 analytic: -0.308373, relative error: 3.269435e-08
numerical: -0.880146 analytic: -0.880146, relative error: 1.652522e-08
numerical: -0.724372 analytic: -0.724372, relative error: 1.530291e-08
numerical: -0.827719 analytic: -0.827719, relative error: 7.452018e-09
numerical: 0.688518 analytic: 0.688518, relative error: 3.531677e-08
numerical: 1.130295 analytic: 1.130295, relative error: 3.351763e-08
numerical: -0.388108 analytic: -0.388108, relative error: 1.213262e-07
numerical: 2.127733 analytic: 2.127733, relative error: 7.326745e-09
numerical: -0.028263 analytic: -0.028263, relative error: 6.221455e-07
numerical: 0.232665 analytic: 0.232665, relative error: 9.635498e-08
numerical: 0.509197 analytic: 0.509197, relative error: 3.078888e-08
numerical: 2.539579 analytic: 2.539578, relative error: 1.700143e-08
numerical: -2.071840 analytic: -2.071840, relative error: 3.372725e-08
numerical: -1.790176 analytic: -1.790176, relative error: 1.104137e-08
```

[15]:
```
# 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: 2.311215e+00 computed in 0.155301s
vectorized loss: 2.311215e+00 computed in 0.010526s
```

```
Loss difference: 0.000000
Gradient difference: 0.000000
```

[21]:
```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of over 0.35 on the validation set.

from cs231n.classifiers import Softmax
results = {}
best_val = -1
best_softmax = None

################################################################################
# TODO:                                                                        #
# Use the validation set to set the learning rate and regularization strength. #
# This should be identical to the validation that you did for the SVM; save    #
# the best trained softmax classifer in best_softmax.                          #
################################################################################

# Provided as a reference. You may or may not want to change these␣
 ↪hyperparameters
learning_rates = [1e-7, 5e-7]
regularization_strengths = [2.5e4, 5e4]

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

for lr in learning_rates:
    for rs in regularization_strengths:
        softmax = Softmax()
        loss_hist = softmax.train(X_train, y_train, learning_rate=lr, reg=rs,
                      num_iters=1500, verbose=True)
        y_train_pred = softmax.predict(X_train)
        train_acc = np.mean(y_train == y_train_pred);
        y_val_pred = softmax.predict(X_val)
        val_acc = np.mean(y_val == y_val_pred);
        results[(lr,rs)] = (train_acc,val_acc)
        if val_acc > best_val:
          best_val = val_acc
          best_softmax = softmax

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

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

```
                lr, reg, train_accuracy, val_accuracy))

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

```
iteration 0 / 1500: loss 767.654122
iteration 100 / 1500: loss 281.251100
iteration 200 / 1500: loss 104.286539
iteration 300 / 1500: loss 39.462425
iteration 400 / 1500: loss 15.836921
iteration 500 / 1500: loss 7.166378
iteration 600 / 1500: loss 3.937910
iteration 700 / 1500: loss 2.736180
iteration 800 / 1500: loss 2.328301
iteration 900 / 1500: loss 2.211726
iteration 1000 / 1500: loss 2.140761
iteration 1100 / 1500: loss 2.190938
iteration 1200 / 1500: loss 2.077003
iteration 1300 / 1500: loss 2.109160
iteration 1400 / 1500: loss 2.029783
iteration 0 / 1500: loss 1538.234799
iteration 100 / 1500: loss 207.264576
iteration 200 / 1500: loss 29.540196
iteration 300 / 1500: loss 5.851931
iteration 400 / 1500: loss 2.642459
iteration 500 / 1500: loss 2.250087
iteration 600 / 1500: loss 2.178043
iteration 700 / 1500: loss 2.132787
iteration 800 / 1500: loss 2.119321
iteration 900 / 1500: loss 2.193799
iteration 1000 / 1500: loss 2.136943
iteration 1100 / 1500: loss 2.168282
iteration 1200 / 1500: loss 2.175777
iteration 1300 / 1500: loss 2.121294
iteration 1400 / 1500: loss 2.115354
iteration 0 / 1500: loss 773.123065
iteration 100 / 1500: loss 6.833510
iteration 200 / 1500: loss 2.114473
iteration 300 / 1500: loss 2.096209
iteration 400 / 1500: loss 2.141556
iteration 500 / 1500: loss 2.054993
iteration 600 / 1500: loss 2.093020
iteration 700 / 1500: loss 2.109584
iteration 800 / 1500: loss 2.078541
iteration 900 / 1500: loss 2.090271
iteration 1000 / 1500: loss 2.067577
iteration 1100 / 1500: loss 2.096640
```

```
iteration 1200 / 1500: loss 2.075847
iteration 1300 / 1500: loss 2.054227
iteration 1400 / 1500: loss 2.119331
iteration 0 / 1500: loss 1530.544097
iteration 100 / 1500: loss 2.168288
iteration 200 / 1500: loss 2.166799
iteration 300 / 1500: loss 2.193247
iteration 400 / 1500: loss 2.148061
iteration 500 / 1500: loss 2.146403
iteration 600 / 1500: loss 2.128293
iteration 700 / 1500: loss 2.161879
iteration 800 / 1500: loss 2.117300
iteration 900 / 1500: loss 2.118126
iteration 1000 / 1500: loss 2.116482
iteration 1100 / 1500: loss 2.149414
iteration 1200 / 1500: loss 2.099915
iteration 1300 / 1500: loss 2.151582
iteration 1400 / 1500: loss 2.203879
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.326918 val accuracy: 0.340000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.301265 val accuracy: 0.315000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.329041 val accuracy: 0.347000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.298204 val accuracy: 0.313000
best validation accuracy achieved during cross-validation: 0.347000
```

[22]:
```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: 0.350000
```

Inline Question 2 - *True or False*

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

YourAnswer : True

YourExplanation : When a data point is added to the SVM, its corresponding loss may be 0. But this does not hold for softmax.

[23]:
```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

March 19, 2024

[33]:
```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive')

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

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

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

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

1 Fully-Connected Neural Nets

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

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

```
  out = # the output

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

  return out, cache
```

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

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

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

  return dx, dw
```

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

[34]:
```
# 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
```

[35]:
```
# 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:

[36]:
```
# 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.

[37]:
```
# 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:

[38]:
```
# 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:

[39]:
```
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: x -> $+\infty$ or x -> $-\infty$
2. ReLU: x < 0
3. Leaky ReLU: none

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:

[40]:
```
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:

[41]:
```
np.random.seed(231)
num_classes, num_inputs = 10, 50
x = 0.001 * np.random.randn(num_inputs, num_classes)
```

```
y = np.random.randint(num_classes, size=num_inputs)

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

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

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

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

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

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

8 Two-layer network

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

[42]:
```
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.

[43]:
```
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,optim_config={'learning_rate':␣
 ↪1e-3},print_every=1000)
solver.train()

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

```
(Iteration 1 / 4900) loss: 2.300089
(Epoch 0 / 10) train acc: 0.171000; val_acc: 0.170000
(Epoch 1 / 10) train acc: 0.399000; val_acc: 0.428000
(Epoch 2 / 10) train acc: 0.468000; val_acc: 0.444000
(Iteration 1001 / 4900) loss: 1.339096
(Epoch 3 / 10) train acc: 0.466000; val_acc: 0.429000
(Epoch 4 / 10) train acc: 0.504000; val_acc: 0.451000
(Iteration 2001 / 4900) loss: 1.452116
(Epoch 5 / 10) train acc: 0.493000; val_acc: 0.465000
```

```
(Epoch 6 / 10) train acc: 0.537000; val_acc: 0.488000
(Iteration 3001 / 4900) loss: 1.375969
(Epoch 7 / 10) train acc: 0.531000; val_acc: 0.468000
(Epoch 8 / 10) train acc: 0.541000; val_acc: 0.463000
(Iteration 4001 / 4900) loss: 1.189466
(Epoch 9 / 10) train acc: 0.580000; val_acc: 0.469000
(Epoch 10 / 10) train acc: 0.524000; val_acc: 0.456000
```

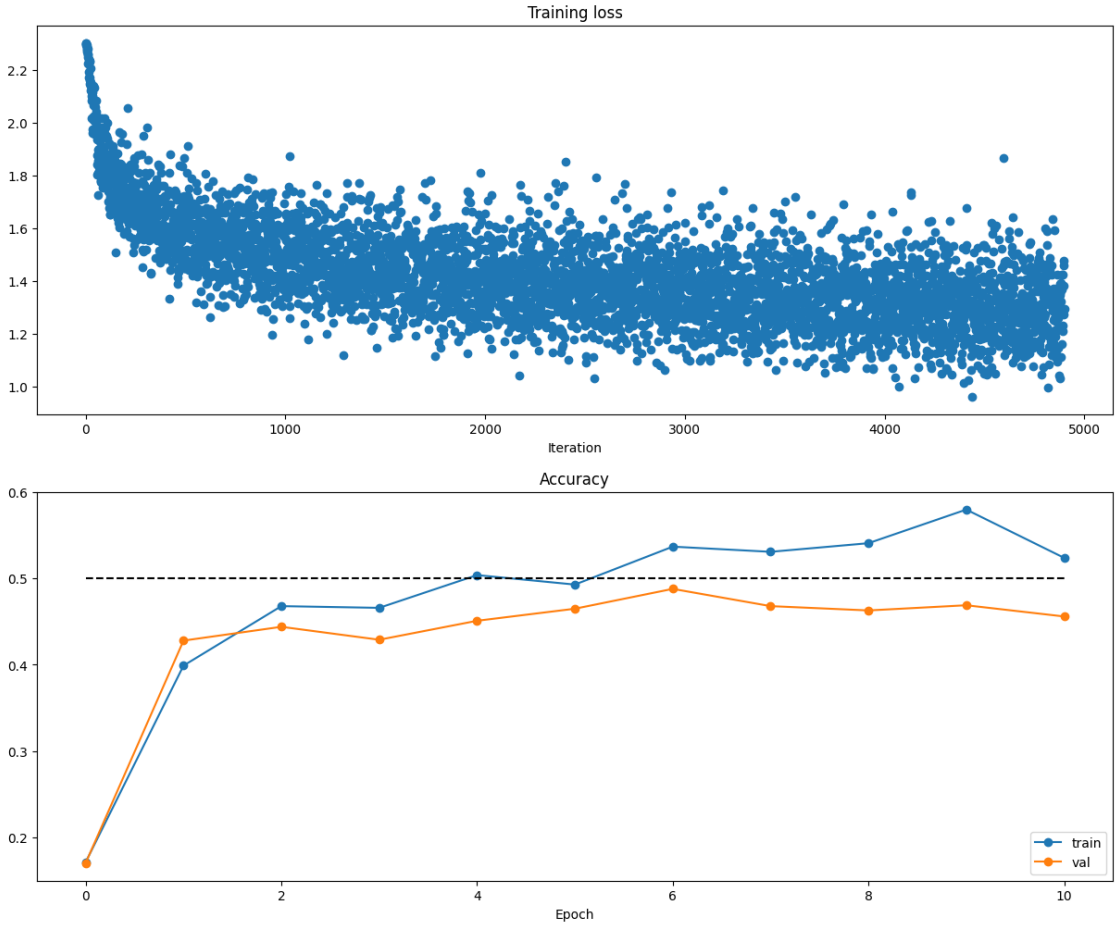
10 Debug the training

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.

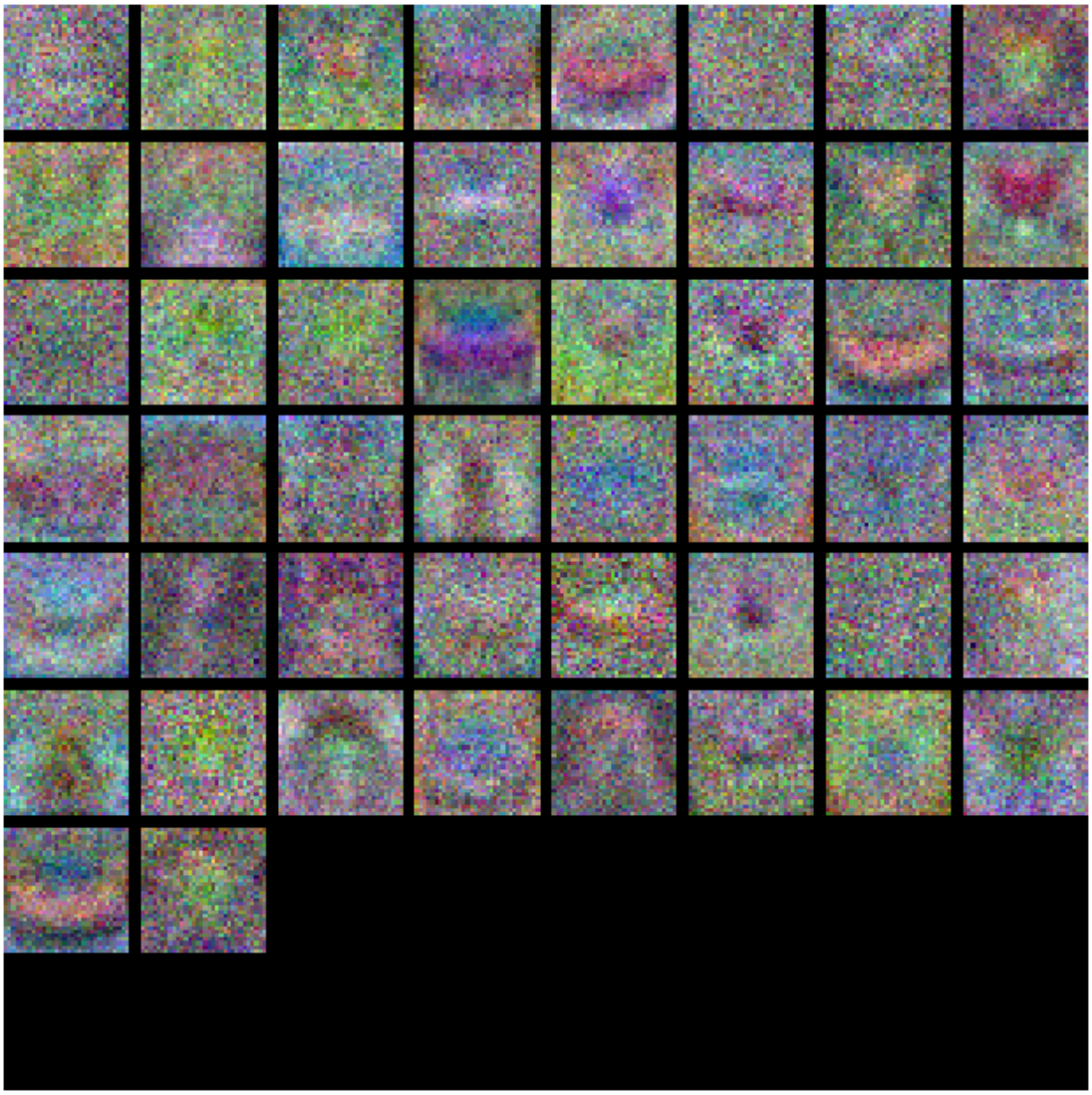
One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

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.

[44]:
```
# 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()
```



[ ]:
```
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.).

[46]:
```
best_model = None
best_val = -1


################################################################################
# TODO: Tune hyperparameters using the validation set. Store your best trained␣
 ↪#
# model in best_model.                                                         ␣
 ↪#
#                                                                              ␣
 ↪#
# To help debug your network, it may help to use visualizations similar to the␣
 ↪#
# ones we used above; these visualizations will have significant qualitative   ␣
 ↪#
# differences from the ones we saw above for the poorly tuned network.         ␣
 ↪#
#                                                                              ␣
 ↪#
# Tweaking hyperparameters by hand can be fun, but you might find it useful to ␣
 ↪#
# write code to sweep through possible combinations of hyperparameters         ␣
 ↪#
# automatically like we did on thexs previous exercises.                       ␣
 ↪  #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

input_size = 32 * 32 * 3
hidden_sizes = [20,200]
num_classes = 10
learning_rates = [1e-4, 1e-2]
for lr in learning_rates:
    for hs in hidden_sizes:
      model = TwoLayerNet(input_size, hs, num_classes)
      solver = Solver(model,data,optim_config={'learning_rate':␣
 ↪lr},print_every=1000)
```

```
      solver.train()
      val_acc = solver.best_val_acc
      if val_acc > best_val:
        best_val = val_acc
        best_model = model

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

```
(Iteration 1 / 4900) loss: 2.301616
(Epoch 0 / 10) train acc: 0.130000; val_acc: 0.116000
(Epoch 1 / 10) train acc: 0.268000; val_acc: 0.288000
(Epoch 2 / 10) train acc: 0.339000; val_acc: 0.344000
(Iteration 1001 / 4900) loss: 1.943473
(Epoch 3 / 10) train acc: 0.384000; val_acc: 0.375000
(Epoch 4 / 10) train acc: 0.393000; val_acc: 0.395000
(Iteration 2001 / 4900) loss: 1.956623
(Epoch 5 / 10) train acc: 0.436000; val_acc: 0.419000
(Epoch 6 / 10) train acc: 0.441000; val_acc: 0.429000
(Iteration 3001 / 4900) loss: 1.584258
(Epoch 7 / 10) train acc: 0.458000; val_acc: 0.426000
(Epoch 8 / 10) train acc: 0.435000; val_acc: 0.434000
(Iteration 4001 / 4900) loss: 1.519249
(Epoch 9 / 10) train acc: 0.439000; val_acc: 0.443000
(Epoch 10 / 10) train acc: 0.423000; val_acc: 0.450000
(Iteration 1 / 4900) loss: 2.308725
(Epoch 0 / 10) train acc: 0.090000; val_acc: 0.092000
(Epoch 1 / 10) train acc: 0.324000; val_acc: 0.329000
(Epoch 2 / 10) train acc: 0.394000; val_acc: 0.372000
(Iteration 1001 / 4900) loss: 1.716837
(Epoch 3 / 10) train acc: 0.408000; val_acc: 0.408000
(Epoch 4 / 10) train acc: 0.466000; val_acc: 0.438000
(Iteration 2001 / 4900) loss: 1.432769
(Epoch 5 / 10) train acc: 0.481000; val_acc: 0.455000
(Epoch 6 / 10) train acc: 0.480000; val_acc: 0.470000
(Iteration 3001 / 4900) loss: 1.592710
(Epoch 7 / 10) train acc: 0.476000; val_acc: 0.475000
(Epoch 8 / 10) train acc: 0.494000; val_acc: 0.473000
(Iteration 4001 / 4900) loss: 1.468212
(Epoch 9 / 10) train acc: 0.503000; val_acc: 0.481000
(Epoch 10 / 10) train acc: 0.498000; val_acc: 0.492000
(Iteration 1 / 4900) loss: 2.303037
(Epoch 0 / 10) train acc: 0.174000; val_acc: 0.162000
```

```
/content/drive/My Drive/cs231n/assignments/assignment1/cs231n/layers.py:827:
RuntimeWarning: divide by zero encountered in log
```

```
  loss = np.sum(-np.log(p)) / num_train
/content/drive/My Drive/cs231n/assignments/assignment1/cs231n/layers.py:825:
RuntimeWarning: overflow encountered in exp
  scores = np.exp(scores)
/content/drive/My Drive/cs231n/assignments/assignment1/cs231n/layers.py:826:
RuntimeWarning: invalid value encountered in divide
  p = scores[range(num_train),y]/np.sum(scores,axis = 1)
/content/drive/My Drive/cs231n/assignments/assignment1/cs231n/layers.py:829:
RuntimeWarning: invalid value encountered in divide
  counts = scores / np.sum(scores,axis = 1).reshape(-1,1)

(Epoch 1 / 10) train acc: 0.114000; val_acc: 0.087000
(Epoch 2 / 10) train acc: 0.104000; val_acc: 0.087000
(Iteration 1001 / 4900) loss: nan
(Epoch 3 / 10) train acc: 0.099000; val_acc: 0.087000
(Epoch 4 / 10) train acc: 0.106000; val_acc: 0.087000
(Iteration 2001 / 4900) loss: nan
(Epoch 5 / 10) train acc: 0.100000; val_acc: 0.087000
(Epoch 6 / 10) train acc: 0.097000; val_acc: 0.087000
(Iteration 3001 / 4900) loss: nan
(Epoch 7 / 10) train acc: 0.089000; val_acc: 0.087000
(Epoch 8 / 10) train acc: 0.089000; val_acc: 0.087000
(Iteration 4001 / 4900) loss: nan
(Epoch 9 / 10) train acc: 0.107000; val_acc: 0.087000
(Epoch 10 / 10) train acc: 0.100000; val_acc: 0.087000
(Iteration 1 / 4900) loss: 2.300248
(Epoch 0 / 10) train acc: 0.179000; val_acc: 0.196000
(Epoch 1 / 10) train acc: 0.109000; val_acc: 0.087000
(Epoch 2 / 10) train acc: 0.086000; val_acc: 0.087000
(Iteration 1001 / 4900) loss: nan
(Epoch 3 / 10) train acc: 0.095000; val_acc: 0.087000
(Epoch 4 / 10) train acc: 0.098000; val_acc: 0.087000
(Iteration 2001 / 4900) loss: nan
(Epoch 5 / 10) train acc: 0.085000; val_acc: 0.087000
(Epoch 6 / 10) train acc: 0.100000; val_acc: 0.087000
(Iteration 3001 / 4900) loss: nan
(Epoch 7 / 10) train acc: 0.097000; val_acc: 0.087000
(Epoch 8 / 10) train acc: 0.097000; val_acc: 0.087000
(Iteration 4001 / 4900) loss: nan
(Epoch 9 / 10) train acc: 0.094000; val_acc: 0.087000
(Epoch 10 / 10) train acc: 0.104000; val_acc: 0.087000
```

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.

[47]:
```
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:  0.492
```

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

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 : 3

YourExplanation : Testing accuracy is much lower than the training accuracy, indicating that there is an overfitting problem. The overfitting problem can be solved by increasing the regularization strength.

[ ]:

features

March 19, 2024

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

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

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

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

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

1 Image features exercise

Complete and hand in this completed worksheet (including its outputs and any supporting code outside of the worksheet) with your assignment submission. For more details see the assignments page on the course website.

We have seen that we can achieve reasonable performance on an image classification task by training a linear classifier on the pixels of the input image. In this exercise we will show that we can improve our classification performance by training linear classifiers not on raw pixels but on features that are computed from the raw pixels.

All of your work for this exercise will be done in this notebook.

[ ]:
```
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.

[ ]:
```
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.

[ ]:
```
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.

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

from cs231n.classifiers.linear_classifier import LinearSVM

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

results = {}
best_val = -1
best_svm = None

################################################################################
# TODO:                                                                        #
# Use the validation set to set the learning rate and regularization strength. #
# This should be identical to the validation that you did for the SVM; save    #
# the best trained classifer in best_svm. You might also want to play          #
# with different numbers of bins in the color histogram. If you are careful    #
# you should be able to get accuracy of near 0.44 on the validation set.       #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

for lr in learning_rates:
    for rs in regularization_strengths:
        svm = LinearSVM()
        loss_hist = svm.train(X_train_feats, y_train, learning_rate=lr, reg=rs,
                              num_iters=1500, verbose=True)
        y_train_pred = svm.predict(X_train_feats)
        train_acc = np.mean(y_train == y_train_pred);
        y_val_pred = svm.predict(X_val_feats)
        val_acc = np.mean(y_val == y_val_pred);
        results[(lr,rs)] = (train_acc,val_acc)
        if val_acc > best_val:
          best_val = val_acc
          best_svm = svm

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

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

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

```
iteration 0 / 1500: loss 87.799822
iteration 100 / 1500: loss 86.220622
iteration 200 / 1500: loss 84.710986
iteration 300 / 1500: loss 83.201032
iteration 400 / 1500: loss 81.736079
iteration 500 / 1500: loss 80.300227
iteration 600 / 1500: loss 78.888262
iteration 700 / 1500: loss 77.500871
iteration 800 / 1500: loss 76.136424
iteration 900 / 1500: loss 74.811267
iteration 1000 / 1500: loss 73.525294
iteration 1100 / 1500: loss 72.241071
iteration 1200 / 1500: loss 70.987798
iteration 1300 / 1500: loss 69.749820
iteration 1400 / 1500: loss 68.552501
iteration 0 / 1500: loss 818.057877
iteration 100 / 1500: loss 671.342985
iteration 200 / 1500: loss 551.231272
iteration 300 / 1500: loss 452.886163
iteration 400 / 1500: loss 372.386616
iteration 500 / 1500: loss 306.488368
iteration 600 / 1500: loss 252.536851
iteration 700 / 1500: loss 208.366630
iteration 800 / 1500: loss 172.223399
iteration 900 / 1500: loss 142.615450
iteration 1000 / 1500: loss 118.379542
iteration 1100 / 1500: loss 98.546368
iteration 1200 / 1500: loss 82.307960
iteration 1300 / 1500: loss 69.011373
iteration 1400 / 1500: loss 58.127720
iteration 0 / 1500: loss 7464.196944
iteration 100 / 1500: loss 1007.843814
iteration 200 / 1500: loss 142.824448
iteration 300 / 1500: loss 26.930019
iteration 400 / 1500: loss 11.401929
iteration 500 / 1500: loss 9.321870
iteration 600 / 1500: loss 9.043113
iteration 700 / 1500: loss 9.005768
```

```
iteration 800 / 1500: loss 9.000770
iteration 900 / 1500: loss 9.000100
iteration 1000 / 1500: loss 9.000011
iteration 1100 / 1500: loss 8.999998
iteration 1200 / 1500: loss 8.999996
iteration 1300 / 1500: loss 8.999996
iteration 1400 / 1500: loss 8.999997
iteration 0 / 1500: loss 90.498397
iteration 100 / 1500: loss 75.714226
iteration 200 / 1500: loss 63.612658
iteration 300 / 1500: loss 53.709864
iteration 400 / 1500: loss 45.602253
iteration 500 / 1500: loss 38.958486
iteration 600 / 1500: loss 33.526658
iteration 700 / 1500: loss 29.080767
iteration 800 / 1500: loss 25.446115
iteration 900 / 1500: loss 22.461828
iteration 1000 / 1500: loss 20.015323
iteration 1100 / 1500: loss 18.016971
iteration 1200 / 1500: loss 16.381528
iteration 1300 / 1500: loss 15.049196
iteration 1400 / 1500: loss 13.948071
iteration 0 / 1500: loss 801.910809
iteration 100 / 1500: loss 115.237476
iteration 200 / 1500: loss 23.234653
iteration 300 / 1500: loss 10.907593
iteration 400 / 1500: loss 9.255502
iteration 500 / 1500: loss 9.034126
iteration 600 / 1500: loss 9.004551
iteration 700 / 1500: loss 9.000579
iteration 800 / 1500: loss 9.000051
iteration 900 / 1500: loss 8.999985
iteration 1000 / 1500: loss 8.999960
iteration 1100 / 1500: loss 8.999966
iteration 1200 / 1500: loss 8.999968
iteration 1300 / 1500: loss 8.999962
iteration 1400 / 1500: loss 8.999964
iteration 0 / 1500: loss 7607.385734
iteration 100 / 1500: loss 9.000001
iteration 200 / 1500: loss 8.999996
iteration 300 / 1500: loss 8.999997
iteration 400 / 1500: loss 8.999996
iteration 500 / 1500: loss 8.999998
iteration 600 / 1500: loss 8.999997
iteration 700 / 1500: loss 8.999997
iteration 800 / 1500: loss 8.999997
iteration 900 / 1500: loss 8.999998
iteration 1000 / 1500: loss 8.999997
```

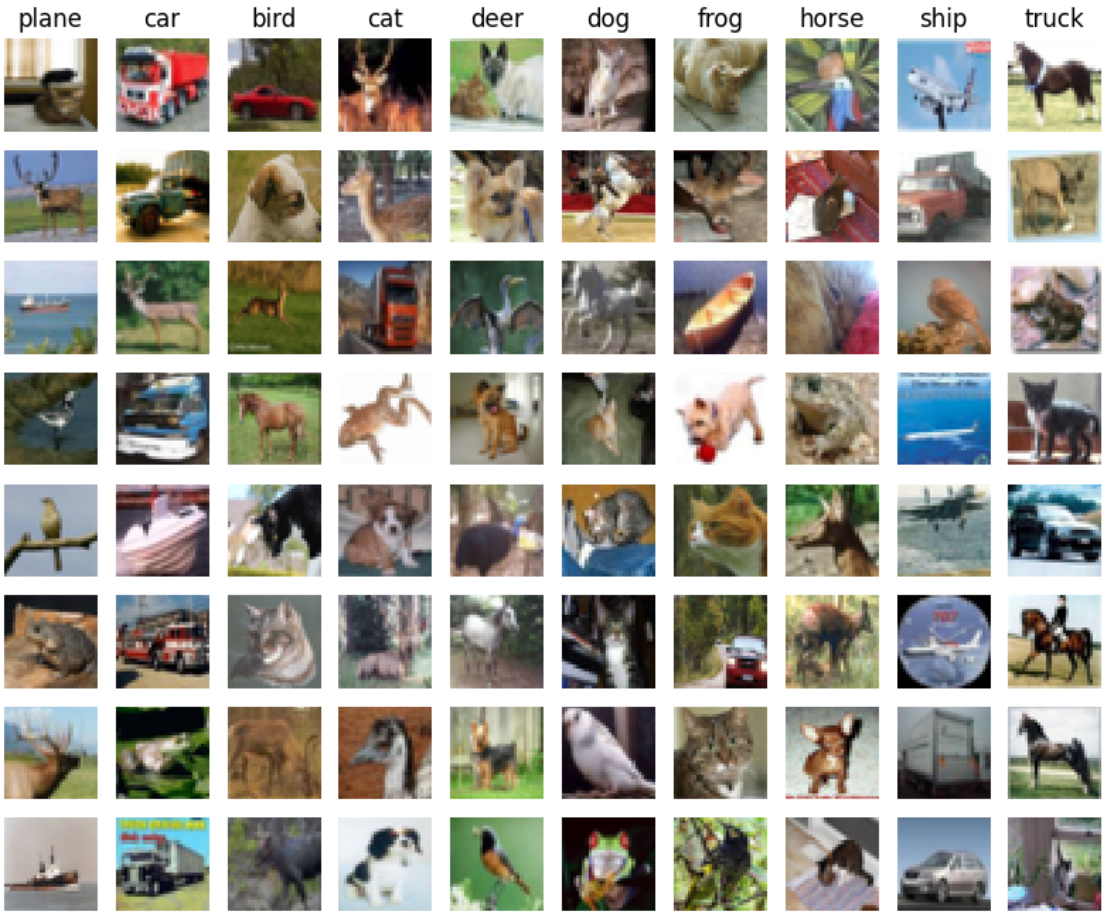
```
iteration 1100 / 1500: loss 8.999998
iteration 1200 / 1500: loss 8.999997
iteration 1300 / 1500: loss 8.999997
iteration 1400 / 1500: loss 8.999998
iteration 0 / 1500: loss 86.913019
iteration 100 / 1500: loss 19.448467
iteration 200 / 1500: loss 10.396473
iteration 300 / 1500: loss 9.186833
iteration 400 / 1500: loss 9.024887
iteration 500 / 1500: loss 9.002990
iteration 600 / 1500: loss 9.000152
iteration 700 / 1500: loss 8.999741
iteration 800 / 1500: loss 8.999657
iteration 900 / 1500: loss 8.999582
iteration 1000 / 1500: loss 8.999615
iteration 1100 / 1500: loss 8.999613
iteration 1200 / 1500: loss 8.999667
iteration 1300 / 1500: loss 8.999522
iteration 1400 / 1500: loss 8.999603
iteration 0 / 1500: loss 757.558004
iteration 100 / 1500: loss 8.999971
iteration 200 / 1500: loss 8.999977
iteration 300 / 1500: loss 8.999963
iteration 400 / 1500: loss 8.999970
iteration 500 / 1500: loss 8.999965
iteration 600 / 1500: loss 8.999964
iteration 700 / 1500: loss 8.999979
iteration 800 / 1500: loss 8.999966
iteration 900 / 1500: loss 8.999961
iteration 1000 / 1500: loss 8.999968
iteration 1100 / 1500: loss 8.999963
iteration 1200 / 1500: loss 8.999967
iteration 1300 / 1500: loss 8.999967
iteration 1400 / 1500: loss 8.999969
iteration 0 / 1500: loss 8127.457776
iteration 100 / 1500: loss 9.000000
iteration 200 / 1500: loss 8.999999
iteration 300 / 1500: loss 9.000000
iteration 400 / 1500: loss 9.000001
iteration 500 / 1500: loss 8.999999
iteration 600 / 1500: loss 9.000000
iteration 700 / 1500: loss 8.999999
iteration 800 / 1500: loss 9.000000
iteration 900 / 1500: loss 9.000000
iteration 1000 / 1500: loss 9.000000
iteration 1100 / 1500: loss 9.000001
iteration 1200 / 1500: loss 9.000000
iteration 1300 / 1500: loss 8.999999
```

```
iteration 1400 / 1500: loss 9.000000
lr 1.000000e-09 reg 5.000000e+04 train accuracy: 0.094939 val accuracy: 0.076000
lr 1.000000e-09 reg 5.000000e+05 train accuracy: 0.079388 val accuracy: 0.085000
lr 1.000000e-09 reg 5.000000e+06 train accuracy: 0.415816 val accuracy: 0.421000
lr 1.000000e-08 reg 5.000000e+04 train accuracy: 0.084184 val accuracy: 0.078000
lr 1.000000e-08 reg 5.000000e+05 train accuracy: 0.416980 val accuracy: 0.424000
lr 1.000000e-08 reg 5.000000e+06 train accuracy: 0.397082 val accuracy: 0.388000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.415898 val accuracy: 0.411000
lr 1.000000e-07 reg 5.000000e+05 train accuracy: 0.400429 val accuracy: 0.394000
lr 1.000000e-07 reg 5.000000e+06 train accuracy: 0.315837 val accuracy: 0.303000
best validation accuracy achieved: 0.424000
```

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

```
0.428
```

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

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



1.3.1 Inline question 1:

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

YourAnswer : Some of the misclassified examples are similar to the identified class, but others do not look similar.

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.

[ ]:
```
# 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)
```

[70]:
```
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 = -1
best_prams = None

################################################################################
# TODO: Train a two-layer neural network on image features. You may want to    #
# cross-validate various parameters as in previous sections. Store your best   #
# model in the best_net variable.                                              #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

input_size = X_train_feats.shape[1]
hidden_sizes = [500]
num_classes = 10
learning_rates =  [1e-2 ,1e-1, 5e-1, 1, 5]
regs = [1e-3, 5e-3, 1e-2, 1e-1, 0.5, 1]
for lr in learning_rates:
    for hs in hidden_sizes:
      for reg in regs:
        net = TwoLayerNet(input_dim, hidden_dim, num_classes, reg = reg)
```

```
        solver = Solver(net,data,optim_config={'learning_rate':␣
 ↪lr},print_every=1000)
        solver.train()
        val_acc = solver.best_val_acc
        if val_acc > best_val:
          best_val = val_acc
          best_net = net
          best_prams = [lr,hs,reg]

print('best validation accuracy achieved during cross-validation: %f' %␣
 ↪best_val)
print(best_prams)
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

```
(Iteration 1 / 4900) loss: 2.302606
(Epoch 0 / 10) train acc: 0.091000; val_acc: 0.112000
(Epoch 1 / 10) train acc: 0.212000; val_acc: 0.233000
(Epoch 2 / 10) train acc: 0.247000; val_acc: 0.268000
(Iteration 1001 / 4900) loss: 2.233199
(Epoch 3 / 10) train acc: 0.252000; val_acc: 0.286000
(Epoch 4 / 10) train acc: 0.346000; val_acc: 0.348000
(Iteration 2001 / 4900) loss: 1.773577
(Epoch 5 / 10) train acc: 0.405000; val_acc: 0.395000
(Epoch 6 / 10) train acc: 0.438000; val_acc: 0.427000
(Iteration 3001 / 4900) loss: 1.389160
(Epoch 7 / 10) train acc: 0.464000; val_acc: 0.456000
(Epoch 8 / 10) train acc: 0.477000; val_acc: 0.475000
(Iteration 4001 / 4900) loss: 1.472585
(Epoch 9 / 10) train acc: 0.506000; val_acc: 0.486000
(Epoch 10 / 10) train acc: 0.517000; val_acc: 0.502000
(Iteration 1 / 4900) loss: 2.302832
(Epoch 0 / 10) train acc: 0.089000; val_acc: 0.070000
(Epoch 1 / 10) train acc: 0.158000; val_acc: 0.180000
(Epoch 2 / 10) train acc: 0.304000; val_acc: 0.299000
(Iteration 1001 / 4900) loss: 2.252328
(Epoch 3 / 10) train acc: 0.292000; val_acc: 0.278000
(Epoch 4 / 10) train acc: 0.343000; val_acc: 0.353000
(Iteration 2001 / 4900) loss: 1.910919
(Epoch 5 / 10) train acc: 0.430000; val_acc: 0.394000
(Epoch 6 / 10) train acc: 0.464000; val_acc: 0.420000
(Iteration 3001 / 4900) loss: 1.634541
(Epoch 7 / 10) train acc: 0.467000; val_acc: 0.450000
(Epoch 8 / 10) train acc: 0.473000; val_acc: 0.463000
(Iteration 4001 / 4900) loss: 1.441512
(Epoch 9 / 10) train acc: 0.494000; val_acc: 0.486000
(Epoch 10 / 10) train acc: 0.497000; val_acc: 0.499000
(Iteration 1 / 4900) loss: 2.303022
```

```
(Epoch 0 / 10) train acc: 0.107000; val_acc: 0.086000
(Epoch 1 / 10) train acc: 0.169000; val_acc: 0.193000
(Epoch 2 / 10) train acc: 0.268000; val_acc: 0.243000
(Iteration 1001 / 4900) loss: 2.235219
(Epoch 3 / 10) train acc: 0.261000; val_acc: 0.275000
(Epoch 4 / 10) train acc: 0.348000; val_acc: 0.322000
(Iteration 2001 / 4900) loss: 1.887166
(Epoch 5 / 10) train acc: 0.380000; val_acc: 0.385000
(Epoch 6 / 10) train acc: 0.418000; val_acc: 0.416000
(Iteration 3001 / 4900) loss: 1.720815
(Epoch 7 / 10) train acc: 0.473000; val_acc: 0.448000
(Epoch 8 / 10) train acc: 0.480000; val_acc: 0.464000
(Iteration 4001 / 4900) loss: 1.601102
(Epoch 9 / 10) train acc: 0.493000; val_acc: 0.488000
(Epoch 10 / 10) train acc: 0.495000; val_acc: 0.486000
(Iteration 1 / 4900) loss: 2.306647
(Epoch 0 / 10) train acc: 0.090000; val_acc: 0.108000
(Epoch 1 / 10) train acc: 0.166000; val_acc: 0.166000
(Epoch 2 / 10) train acc: 0.228000; val_acc: 0.223000
(Iteration 1001 / 4900) loss: 2.296142
(Epoch 3 / 10) train acc: 0.254000; val_acc: 0.246000
(Epoch 4 / 10) train acc: 0.265000; val_acc: 0.274000
(Iteration 2001 / 4900) loss: 2.176844
(Epoch 5 / 10) train acc: 0.291000; val_acc: 0.293000
(Epoch 6 / 10) train acc: 0.283000; val_acc: 0.312000
(Iteration 3001 / 4900) loss: 2.089430
(Epoch 7 / 10) train acc: 0.358000; val_acc: 0.336000
(Epoch 8 / 10) train acc: 0.378000; val_acc: 0.369000
(Iteration 4001 / 4900) loss: 2.049766
(Epoch 9 / 10) train acc: 0.372000; val_acc: 0.369000
(Epoch 10 / 10) train acc: 0.396000; val_acc: 0.388000
(Iteration 1 / 4900) loss: 2.323279
(Epoch 0 / 10) train acc: 0.082000; val_acc: 0.101000
(Epoch 1 / 10) train acc: 0.104000; val_acc: 0.113000
(Epoch 2 / 10) train acc: 0.103000; val_acc: 0.107000
(Iteration 1001 / 4900) loss: 2.302405
(Epoch 3 / 10) train acc: 0.122000; val_acc: 0.079000
(Epoch 4 / 10) train acc: 0.109000; val_acc: 0.112000
(Iteration 2001 / 4900) loss: 2.302925
(Epoch 5 / 10) train acc: 0.109000; val_acc: 0.098000
(Epoch 6 / 10) train acc: 0.106000; val_acc: 0.098000
(Iteration 3001 / 4900) loss: 2.302606
(Epoch 7 / 10) train acc: 0.092000; val_acc: 0.112000
(Epoch 8 / 10) train acc: 0.095000; val_acc: 0.105000
(Iteration 4001 / 4900) loss: 2.302143
(Epoch 9 / 10) train acc: 0.081000; val_acc: 0.102000
(Epoch 10 / 10) train acc: 0.105000; val_acc: 0.105000
(Iteration 1 / 4900) loss: 2.343501
```

```
(Epoch 0 / 10) train acc: 0.105000; val_acc: 0.100000
(Epoch 1 / 10) train acc: 0.101000; val_acc: 0.119000
(Epoch 2 / 10) train acc: 0.104000; val_acc: 0.119000
(Iteration 1001 / 4900) loss: 2.302834
(Epoch 3 / 10) train acc: 0.104000; val_acc: 0.113000
(Epoch 4 / 10) train acc: 0.113000; val_acc: 0.113000
(Iteration 2001 / 4900) loss: 2.302558
(Epoch 5 / 10) train acc: 0.110000; val_acc: 0.079000
(Epoch 6 / 10) train acc: 0.093000; val_acc: 0.113000
(Iteration 3001 / 4900) loss: 2.302918
(Epoch 7 / 10) train acc: 0.093000; val_acc: 0.079000
(Epoch 8 / 10) train acc: 0.101000; val_acc: 0.098000
(Iteration 4001 / 4900) loss: 2.303311
(Epoch 9 / 10) train acc: 0.096000; val_acc: 0.098000
(Epoch 10 / 10) train acc: 0.095000; val_acc: 0.098000
(Iteration 1 / 4900) loss: 2.302628
(Epoch 0 / 10) train acc: 0.119000; val_acc: 0.102000
(Epoch 1 / 10) train acc: 0.493000; val_acc: 0.500000
(Epoch 2 / 10) train acc: 0.539000; val_acc: 0.526000
(Iteration 1001 / 4900) loss: 1.438289
(Epoch 3 / 10) train acc: 0.550000; val_acc: 0.538000
(Epoch 4 / 10) train acc: 0.551000; val_acc: 0.546000
(Iteration 2001 / 4900) loss: 1.184881
(Epoch 5 / 10) train acc: 0.585000; val_acc: 0.556000
(Epoch 6 / 10) train acc: 0.594000; val_acc: 0.572000
(Iteration 3001 / 4900) loss: 0.928839
(Epoch 7 / 10) train acc: 0.628000; val_acc: 0.591000
(Epoch 8 / 10) train acc: 0.672000; val_acc: 0.586000
(Iteration 4001 / 4900) loss: 1.051755
(Epoch 9 / 10) train acc: 0.654000; val_acc: 0.572000
(Epoch 10 / 10) train acc: 0.688000; val_acc: 0.595000
(Iteration 1 / 4900) loss: 2.302849
(Epoch 0 / 10) train acc: 0.086000; val_acc: 0.102000
(Epoch 1 / 10) train acc: 0.498000; val_acc: 0.497000
(Epoch 2 / 10) train acc: 0.513000; val_acc: 0.506000
(Iteration 1001 / 4900) loss: 1.531231
(Epoch 3 / 10) train acc: 0.556000; val_acc: 0.527000
(Epoch 4 / 10) train acc: 0.545000; val_acc: 0.546000
(Iteration 2001 / 4900) loss: 1.408112
(Epoch 5 / 10) train acc: 0.547000; val_acc: 0.550000
(Epoch 6 / 10) train acc: 0.603000; val_acc: 0.555000
(Iteration 3001 / 4900) loss: 1.308340
(Epoch 7 / 10) train acc: 0.595000; val_acc: 0.572000
(Epoch 8 / 10) train acc: 0.582000; val_acc: 0.563000
(Iteration 4001 / 4900) loss: 1.197786
(Epoch 9 / 10) train acc: 0.607000; val_acc: 0.581000
(Epoch 10 / 10) train acc: 0.608000; val_acc: 0.577000
(Iteration 1 / 4900) loss: 2.303029
```

```
(Epoch 0 / 10) train acc: 0.114000; val_acc: 0.110000
(Epoch 1 / 10) train acc: 0.508000; val_acc: 0.482000
(Epoch 2 / 10) train acc: 0.538000; val_acc: 0.516000
(Iteration 1001 / 4900) loss: 1.385247
(Epoch 3 / 10) train acc: 0.540000; val_acc: 0.531000
(Epoch 4 / 10) train acc: 0.520000; val_acc: 0.537000
(Iteration 2001 / 4900) loss: 1.379279
(Epoch 5 / 10) train acc: 0.547000; val_acc: 0.540000
(Epoch 6 / 10) train acc: 0.547000; val_acc: 0.532000
(Iteration 3001 / 4900) loss: 1.488798
(Epoch 7 / 10) train acc: 0.597000; val_acc: 0.529000
(Epoch 8 / 10) train acc: 0.581000; val_acc: 0.555000
(Iteration 4001 / 4900) loss: 1.512713
(Epoch 9 / 10) train acc: 0.551000; val_acc: 0.541000
(Epoch 10 / 10) train acc: 0.565000; val_acc: 0.547000
(Iteration 1 / 4900) loss: 2.306665
(Epoch 0 / 10) train acc: 0.116000; val_acc: 0.128000
(Epoch 1 / 10) train acc: 0.380000; val_acc: 0.390000
(Epoch 2 / 10) train acc: 0.404000; val_acc: 0.423000
(Iteration 1001 / 4900) loss: 1.877190
(Epoch 3 / 10) train acc: 0.455000; val_acc: 0.422000
(Epoch 4 / 10) train acc: 0.412000; val_acc: 0.422000
(Iteration 2001 / 4900) loss: 1.981235
(Epoch 5 / 10) train acc: 0.437000; val_acc: 0.420000
(Epoch 6 / 10) train acc: 0.451000; val_acc: 0.446000
(Iteration 3001 / 4900) loss: 1.957586
(Epoch 7 / 10) train acc: 0.423000; val_acc: 0.439000
(Epoch 8 / 10) train acc: 0.434000; val_acc: 0.442000
(Iteration 4001 / 4900) loss: 1.974286
(Epoch 9 / 10) train acc: 0.459000; val_acc: 0.449000
(Epoch 10 / 10) train acc: 0.449000; val_acc: 0.412000
(Iteration 1 / 4900) loss: 2.322859
(Epoch 0 / 10) train acc: 0.118000; val_acc: 0.105000
(Epoch 1 / 10) train acc: 0.100000; val_acc: 0.078000
(Epoch 2 / 10) train acc: 0.094000; val_acc: 0.078000
(Iteration 1001 / 4900) loss: 2.302610
(Epoch 3 / 10) train acc: 0.102000; val_acc: 0.098000
(Epoch 4 / 10) train acc: 0.093000; val_acc: 0.102000
(Iteration 2001 / 4900) loss: 2.304563
(Epoch 5 / 10) train acc: 0.101000; val_acc: 0.079000
(Epoch 6 / 10) train acc: 0.100000; val_acc: 0.113000
(Iteration 3001 / 4900) loss: 2.304299
(Epoch 7 / 10) train acc: 0.095000; val_acc: 0.098000
(Epoch 8 / 10) train acc: 0.131000; val_acc: 0.126000
(Iteration 4001 / 4900) loss: 2.300758
(Epoch 9 / 10) train acc: 0.141000; val_acc: 0.112000
(Epoch 10 / 10) train acc: 0.106000; val_acc: 0.135000
(Iteration 1 / 4900) loss: 2.343296
```

```
(Epoch 0 / 10) train acc: 0.106000; val_acc: 0.107000
(Epoch 1 / 10) train acc: 0.087000; val_acc: 0.107000
(Epoch 2 / 10) train acc: 0.092000; val_acc: 0.079000
(Iteration 1001 / 4900) loss: 2.302638
(Epoch 3 / 10) train acc: 0.108000; val_acc: 0.078000
(Epoch 4 / 10) train acc: 0.097000; val_acc: 0.079000
(Iteration 2001 / 4900) loss: 2.303614
(Epoch 5 / 10) train acc: 0.096000; val_acc: 0.087000
(Epoch 6 / 10) train acc: 0.105000; val_acc: 0.079000
(Iteration 3001 / 4900) loss: 2.303041
(Epoch 7 / 10) train acc: 0.103000; val_acc: 0.098000
(Epoch 8 / 10) train acc: 0.091000; val_acc: 0.119000
(Iteration 4001 / 4900) loss: 2.303484
(Epoch 9 / 10) train acc: 0.092000; val_acc: 0.087000
(Epoch 10 / 10) train acc: 0.078000; val_acc: 0.119000
(Iteration 1 / 4900) loss: 2.302627
(Epoch 0 / 10) train acc: 0.090000; val_acc: 0.112000
(Epoch 1 / 10) train acc: 0.566000; val_acc: 0.517000
(Epoch 2 / 10) train acc: 0.593000; val_acc: 0.564000
(Iteration 1001 / 4900) loss: 1.011052
(Epoch 3 / 10) train acc: 0.624000; val_acc: 0.566000
(Epoch 4 / 10) train acc: 0.657000; val_acc: 0.574000
(Iteration 2001 / 4900) loss: 1.020858
(Epoch 5 / 10) train acc: 0.666000; val_acc: 0.540000
(Epoch 6 / 10) train acc: 0.700000; val_acc: 0.587000
(Iteration 3001 / 4900) loss: 1.030499
(Epoch 7 / 10) train acc: 0.703000; val_acc: 0.579000
(Epoch 8 / 10) train acc: 0.676000; val_acc: 0.580000
(Iteration 4001 / 4900) loss: 1.270558
(Epoch 9 / 10) train acc: 0.711000; val_acc: 0.601000
(Epoch 10 / 10) train acc: 0.701000; val_acc: 0.587000
(Iteration 1 / 4900) loss: 2.302781
(Epoch 0 / 10) train acc: 0.086000; val_acc: 0.098000
(Epoch 1 / 10) train acc: 0.539000; val_acc: 0.517000
(Epoch 2 / 10) train acc: 0.556000; val_acc: 0.512000
(Iteration 1001 / 4900) loss: 1.529207
(Epoch 3 / 10) train acc: 0.574000; val_acc: 0.542000
(Epoch 4 / 10) train acc: 0.570000; val_acc: 0.534000
(Iteration 2001 / 4900) loss: 1.530710
(Epoch 5 / 10) train acc: 0.548000; val_acc: 0.521000
(Epoch 6 / 10) train acc: 0.571000; val_acc: 0.524000
(Iteration 3001 / 4900) loss: 1.408861
(Epoch 7 / 10) train acc: 0.574000; val_acc: 0.531000
(Epoch 8 / 10) train acc: 0.598000; val_acc: 0.544000
(Iteration 4001 / 4900) loss: 1.400433
(Epoch 9 / 10) train acc: 0.585000; val_acc: 0.550000
(Epoch 10 / 10) train acc: 0.600000; val_acc: 0.529000
(Iteration 1 / 4900) loss: 2.303014
```

```
(Epoch 0 / 10) train acc: 0.185000; val_acc: 0.148000
(Epoch 1 / 10) train acc: 0.493000; val_acc: 0.493000
(Epoch 2 / 10) train acc: 0.566000; val_acc: 0.513000
(Iteration 1001 / 4900) loss: 1.677159
(Epoch 3 / 10) train acc: 0.554000; val_acc: 0.503000
(Epoch 4 / 10) train acc: 0.518000; val_acc: 0.512000
(Iteration 2001 / 4900) loss: 1.353801
(Epoch 5 / 10) train acc: 0.518000; val_acc: 0.518000
(Epoch 6 / 10) train acc: 0.516000; val_acc: 0.495000
(Iteration 3001 / 4900) loss: 1.331727
(Epoch 7 / 10) train acc: 0.537000; val_acc: 0.523000
(Epoch 8 / 10) train acc: 0.541000; val_acc: 0.513000
(Iteration 4001 / 4900) loss: 1.566304
(Epoch 9 / 10) train acc: 0.522000; val_acc: 0.518000
(Epoch 10 / 10) train acc: 0.581000; val_acc: 0.522000
(Iteration 1 / 4900) loss: 2.306691
(Epoch 0 / 10) train acc: 0.097000; val_acc: 0.087000
(Epoch 1 / 10) train acc: 0.393000; val_acc: 0.360000
(Epoch 2 / 10) train acc: 0.384000; val_acc: 0.370000
(Iteration 1001 / 4900) loss: 1.998896
(Epoch 3 / 10) train acc: 0.395000; val_acc: 0.389000
(Epoch 4 / 10) train acc: 0.349000; val_acc: 0.362000
(Iteration 2001 / 4900) loss: 2.068132
(Epoch 5 / 10) train acc: 0.408000; val_acc: 0.366000
(Epoch 6 / 10) train acc: 0.365000; val_acc: 0.378000
(Iteration 3001 / 4900) loss: 2.050387
(Epoch 7 / 10) train acc: 0.399000; val_acc: 0.354000
(Epoch 8 / 10) train acc: 0.358000; val_acc: 0.366000
(Iteration 4001 / 4900) loss: 2.041253
(Epoch 9 / 10) train acc: 0.383000; val_acc: 0.420000
(Epoch 10 / 10) train acc: 0.377000; val_acc: 0.363000
(Iteration 1 / 4900) loss: 2.322888
(Epoch 0 / 10) train acc: 0.161000; val_acc: 0.145000
(Epoch 1 / 10) train acc: 0.093000; val_acc: 0.087000
(Epoch 2 / 10) train acc: 0.141000; val_acc: 0.165000
(Iteration 1001 / 4900) loss: 2.298798
(Epoch 3 / 10) train acc: 0.189000; val_acc: 0.199000
(Epoch 4 / 10) train acc: 0.129000; val_acc: 0.127000
(Iteration 2001 / 4900) loss: 2.314968
(Epoch 5 / 10) train acc: 0.179000; val_acc: 0.169000
(Epoch 6 / 10) train acc: 0.168000; val_acc: 0.124000
(Iteration 3001 / 4900) loss: 2.272961
(Epoch 7 / 10) train acc: 0.122000; val_acc: 0.119000
(Epoch 8 / 10) train acc: 0.169000; val_acc: 0.182000
(Iteration 4001 / 4900) loss: 2.282503
(Epoch 9 / 10) train acc: 0.168000; val_acc: 0.181000
(Epoch 10 / 10) train acc: 0.165000; val_acc: 0.166000
(Iteration 1 / 4900) loss: 2.343628
```

```
(Epoch 0 / 10) train acc: 0.155000; val_acc: 0.160000
(Epoch 1 / 10) train acc: 0.095000; val_acc: 0.078000
(Epoch 2 / 10) train acc: 0.107000; val_acc: 0.113000
(Iteration 1001 / 4900) loss: 2.304015
(Epoch 3 / 10) train acc: 0.091000; val_acc: 0.078000
(Epoch 4 / 10) train acc: 0.089000; val_acc: 0.102000
(Iteration 2001 / 4900) loss: 2.305045
(Epoch 5 / 10) train acc: 0.092000; val_acc: 0.107000
(Epoch 6 / 10) train acc: 0.105000; val_acc: 0.078000
(Iteration 3001 / 4900) loss: 2.302542
(Epoch 7 / 10) train acc: 0.104000; val_acc: 0.087000
(Epoch 8 / 10) train acc: 0.102000; val_acc: 0.105000
(Iteration 4001 / 4900) loss: 2.296368
(Epoch 9 / 10) train acc: 0.117000; val_acc: 0.098000
(Epoch 10 / 10) train acc: 0.092000; val_acc: 0.078000
(Iteration 1 / 4900) loss: 2.302587
(Epoch 0 / 10) train acc: 0.094000; val_acc: 0.105000
(Epoch 1 / 10) train acc: 0.539000; val_acc: 0.496000
(Epoch 2 / 10) train acc: 0.603000; val_acc: 0.529000
(Iteration 1001 / 4900) loss: 1.265599
(Epoch 3 / 10) train acc: 0.602000; val_acc: 0.540000
(Epoch 4 / 10) train acc: 0.610000; val_acc: 0.536000
(Iteration 2001 / 4900) loss: 1.436698
(Epoch 5 / 10) train acc: 0.624000; val_acc: 0.563000
(Epoch 6 / 10) train acc: 0.662000; val_acc: 0.560000
(Iteration 3001 / 4900) loss: 1.286507
(Epoch 7 / 10) train acc: 0.648000; val_acc: 0.551000
(Epoch 8 / 10) train acc: 0.640000; val_acc: 0.549000
(Iteration 4001 / 4900) loss: 1.232344
(Epoch 9 / 10) train acc: 0.664000; val_acc: 0.532000
(Epoch 10 / 10) train acc: 0.644000; val_acc: 0.522000
(Iteration 1 / 4900) loss: 2.302775
(Epoch 0 / 10) train acc: 0.090000; val_acc: 0.119000
(Epoch 1 / 10) train acc: 0.501000; val_acc: 0.482000
(Epoch 2 / 10) train acc: 0.531000; val_acc: 0.504000
(Iteration 1001 / 4900) loss: 1.738569
(Epoch 3 / 10) train acc: 0.527000; val_acc: 0.520000
(Epoch 4 / 10) train acc: 0.535000; val_acc: 0.518000
(Iteration 2001 / 4900) loss: 1.436861
(Epoch 5 / 10) train acc: 0.542000; val_acc: 0.500000
(Epoch 6 / 10) train acc: 0.562000; val_acc: 0.513000
(Iteration 3001 / 4900) loss: 1.660758
(Epoch 7 / 10) train acc: 0.540000; val_acc: 0.529000
(Epoch 8 / 10) train acc: 0.554000; val_acc: 0.516000
(Iteration 4001 / 4900) loss: 1.447080
(Epoch 9 / 10) train acc: 0.527000; val_acc: 0.523000
(Epoch 10 / 10) train acc: 0.515000; val_acc: 0.504000
(Iteration 1 / 4900) loss: 2.303030
```

```
(Epoch 0 / 10) train acc: 0.108000; val_acc: 0.112000
(Epoch 1 / 10) train acc: 0.462000; val_acc: 0.473000
(Epoch 2 / 10) train acc: 0.498000; val_acc: 0.492000
(Iteration 1001 / 4900) loss: 1.646771
(Epoch 3 / 10) train acc: 0.504000; val_acc: 0.483000
(Epoch 4 / 10) train acc: 0.481000; val_acc: 0.447000
(Iteration 2001 / 4900) loss: 1.480930
(Epoch 5 / 10) train acc: 0.465000; val_acc: 0.473000
(Epoch 6 / 10) train acc: 0.502000; val_acc: 0.493000
(Iteration 3001 / 4900) loss: 1.818297
(Epoch 7 / 10) train acc: 0.481000; val_acc: 0.495000
(Epoch 8 / 10) train acc: 0.475000; val_acc: 0.490000
(Iteration 4001 / 4900) loss: 1.838157
(Epoch 9 / 10) train acc: 0.474000; val_acc: 0.465000
(Epoch 10 / 10) train acc: 0.498000; val_acc: 0.444000
(Iteration 1 / 4900) loss: 2.306681
(Epoch 0 / 10) train acc: 0.108000; val_acc: 0.087000
(Epoch 1 / 10) train acc: 0.360000; val_acc: 0.329000
(Epoch 2 / 10) train acc: 0.285000; val_acc: 0.315000
(Iteration 1001 / 4900) loss: 2.264788
(Epoch 3 / 10) train acc: 0.318000; val_acc: 0.316000
(Epoch 4 / 10) train acc: 0.328000; val_acc: 0.301000
(Iteration 2001 / 4900) loss: 2.003153
(Epoch 5 / 10) train acc: 0.375000; val_acc: 0.339000
(Epoch 6 / 10) train acc: 0.366000; val_acc: 0.329000
(Iteration 3001 / 4900) loss: 2.125555
(Epoch 7 / 10) train acc: 0.327000; val_acc: 0.332000
(Epoch 8 / 10) train acc: 0.346000; val_acc: 0.334000
(Iteration 4001 / 4900) loss: 2.182108
(Epoch 9 / 10) train acc: 0.320000; val_acc: 0.364000
(Epoch 10 / 10) train acc: 0.355000; val_acc: 0.330000
(Iteration 1 / 4900) loss: 2.322929
(Epoch 0 / 10) train acc: 0.169000; val_acc: 0.157000
(Epoch 1 / 10) train acc: 0.098000; val_acc: 0.119000
(Epoch 2 / 10) train acc: 0.139000; val_acc: 0.119000
(Iteration 1001 / 4900) loss: 2.301511
(Epoch 3 / 10) train acc: 0.159000; val_acc: 0.166000
(Epoch 4 / 10) train acc: 0.171000; val_acc: 0.166000
(Iteration 2001 / 4900) loss: 2.298300
(Epoch 5 / 10) train acc: 0.102000; val_acc: 0.096000
(Epoch 6 / 10) train acc: 0.180000; val_acc: 0.171000
(Iteration 3001 / 4900) loss: 2.311852
(Epoch 7 / 10) train acc: 0.153000; val_acc: 0.160000
(Epoch 8 / 10) train acc: 0.143000; val_acc: 0.148000
(Iteration 4001 / 4900) loss: 2.318630
(Epoch 9 / 10) train acc: 0.174000; val_acc: 0.170000
(Epoch 10 / 10) train acc: 0.154000; val_acc: 0.147000
(Iteration 1 / 4900) loss: 2.343469
```

```
(Epoch 0 / 10) train acc: 0.084000; val_acc: 0.113000
(Epoch 1 / 10) train acc: 0.102000; val_acc: 0.079000
(Epoch 2 / 10) train acc: 0.108000; val_acc: 0.105000
(Iteration 1001 / 4900) loss: 2.301339
(Epoch 3 / 10) train acc: 0.117000; val_acc: 0.102000
(Epoch 4 / 10) train acc: 0.089000; val_acc: 0.119000
(Iteration 2001 / 4900) loss: 2.306301
(Epoch 5 / 10) train acc: 0.104000; val_acc: 0.113000
(Epoch 6 / 10) train acc: 0.094000; val_acc: 0.087000
(Iteration 3001 / 4900) loss: 2.311676
(Epoch 7 / 10) train acc: 0.100000; val_acc: 0.112000
(Epoch 8 / 10) train acc: 0.110000; val_acc: 0.102000
(Iteration 4001 / 4900) loss: 2.296219
(Epoch 9 / 10) train acc: 0.094000; val_acc: 0.078000
(Epoch 10 / 10) train acc: 0.093000; val_acc: 0.098000
(Iteration 1 / 4900) loss: 2.302623
(Epoch 0 / 10) train acc: 0.093000; val_acc: 0.107000
(Epoch 1 / 10) train acc: 0.094000; val_acc: 0.087000
(Epoch 2 / 10) train acc: 0.096000; val_acc: 0.087000
(Iteration 1001 / 4900) loss: nan
(Epoch 3 / 10) train acc: 0.091000; val_acc: 0.087000
(Epoch 4 / 10) train acc: 0.097000; val_acc: 0.087000
(Iteration 2001 / 4900) loss: nan
(Epoch 5 / 10) train acc: 0.096000; val_acc: 0.087000
(Epoch 6 / 10) train acc: 0.114000; val_acc: 0.087000
(Iteration 3001 / 4900) loss: nan
(Epoch 7 / 10) train acc: 0.117000; val_acc: 0.087000
(Epoch 8 / 10) train acc: 0.108000; val_acc: 0.087000
(Iteration 4001 / 4900) loss: nan
(Epoch 9 / 10) train acc: 0.112000; val_acc: 0.087000
(Epoch 10 / 10) train acc: 0.104000; val_acc: 0.087000
(Iteration 1 / 4900) loss: 2.302723
(Epoch 0 / 10) train acc: 0.099000; val_acc: 0.119000
(Epoch 1 / 10) train acc: 0.096000; val_acc: 0.087000
(Epoch 2 / 10) train acc: 0.101000; val_acc: 0.087000
(Iteration 1001 / 4900) loss: nan
(Epoch 3 / 10) train acc: 0.102000; val_acc: 0.087000
(Epoch 4 / 10) train acc: 0.105000; val_acc: 0.087000
(Iteration 2001 / 4900) loss: nan
(Epoch 5 / 10) train acc: 0.101000; val_acc: 0.087000
(Epoch 6 / 10) train acc: 0.089000; val_acc: 0.087000
(Iteration 3001 / 4900) loss: nan
(Epoch 7 / 10) train acc: 0.094000; val_acc: 0.087000
(Epoch 8 / 10) train acc: 0.079000; val_acc: 0.087000
(Iteration 4001 / 4900) loss: nan
(Epoch 9 / 10) train acc: 0.084000; val_acc: 0.087000
(Epoch 10 / 10) train acc: 0.106000; val_acc: 0.087000
(Iteration 1 / 4900) loss: 2.303014
```

```
(Epoch 0 / 10) train acc: 0.093000; val_acc: 0.113000
(Epoch 1 / 10) train acc: 0.102000; val_acc: 0.087000
(Epoch 2 / 10) train acc: 0.103000; val_acc: 0.087000
(Iteration 1001 / 4900) loss: nan
(Epoch 3 / 10) train acc: 0.086000; val_acc: 0.087000
(Epoch 4 / 10) train acc: 0.106000; val_acc: 0.087000
(Iteration 2001 / 4900) loss: nan
(Epoch 5 / 10) train acc: 0.088000; val_acc: 0.087000
(Epoch 6 / 10) train acc: 0.100000; val_acc: 0.087000
(Iteration 3001 / 4900) loss: nan
(Epoch 7 / 10) train acc: 0.114000; val_acc: 0.087000
(Epoch 8 / 10) train acc: 0.082000; val_acc: 0.087000
(Iteration 4001 / 4900) loss: nan
(Epoch 9 / 10) train acc: 0.110000; val_acc: 0.087000
(Epoch 10 / 10) train acc: 0.121000; val_acc: 0.087000
(Iteration 1 / 4900) loss: 2.306701
(Epoch 0 / 10) train acc: 0.083000; val_acc: 0.098000
(Epoch 1 / 10) train acc: 0.106000; val_acc: 0.087000
(Epoch 2 / 10) train acc: 0.092000; val_acc: 0.087000
(Iteration 1001 / 4900) loss: nan
(Epoch 3 / 10) train acc: 0.093000; val_acc: 0.087000
(Epoch 4 / 10) train acc: 0.103000; val_acc: 0.087000
(Iteration 2001 / 4900) loss: nan
(Epoch 5 / 10) train acc: 0.090000; val_acc: 0.087000
(Epoch 6 / 10) train acc: 0.093000; val_acc: 0.087000
(Iteration 3001 / 4900) loss: nan
(Epoch 7 / 10) train acc: 0.105000; val_acc: 0.087000
(Epoch 8 / 10) train acc: 0.097000; val_acc: 0.087000
(Iteration 4001 / 4900) loss: nan
(Epoch 9 / 10) train acc: 0.118000; val_acc: 0.087000
(Epoch 10 / 10) train acc: 0.098000; val_acc: 0.087000
(Iteration 1 / 4900) loss: 2.322997
(Epoch 0 / 10) train acc: 0.094000; val_acc: 0.119000
(Epoch 1 / 10) train acc: 0.114000; val_acc: 0.087000
(Epoch 2 / 10) train acc: 0.085000; val_acc: 0.087000
(Iteration 1001 / 4900) loss: nan
(Epoch 3 / 10) train acc: 0.087000; val_acc: 0.087000
(Epoch 4 / 10) train acc: 0.096000; val_acc: 0.087000
(Iteration 2001 / 4900) loss: nan
(Epoch 5 / 10) train acc: 0.100000; val_acc: 0.087000
(Epoch 6 / 10) train acc: 0.105000; val_acc: 0.087000
(Iteration 3001 / 4900) loss: nan
(Epoch 7 / 10) train acc: 0.081000; val_acc: 0.087000
(Epoch 8 / 10) train acc: 0.087000; val_acc: 0.087000
(Iteration 4001 / 4900) loss: nan
(Epoch 9 / 10) train acc: 0.098000; val_acc: 0.087000
(Epoch 10 / 10) train acc: 0.101000; val_acc: 0.087000
(Iteration 1 / 4900) loss: 2.344109
```

```
(Epoch 0 / 10) train acc: 0.097000; val_acc: 0.098000
(Epoch 1 / 10) train acc: 0.111000; val_acc: 0.087000
(Epoch 2 / 10) train acc: 0.115000; val_acc: 0.087000
(Iteration 1001 / 4900) loss: nan
(Epoch 3 / 10) train acc: 0.099000; val_acc: 0.087000
(Epoch 4 / 10) train acc: 0.100000; val_acc: 0.087000
(Iteration 2001 / 4900) loss: nan
(Epoch 5 / 10) train acc: 0.090000; val_acc: 0.087000
(Epoch 6 / 10) train acc: 0.115000; val_acc: 0.087000
(Iteration 3001 / 4900) loss: nan
(Epoch 7 / 10) train acc: 0.101000; val_acc: 0.087000
(Epoch 8 / 10) train acc: 0.114000; val_acc: 0.087000
(Iteration 4001 / 4900) loss: nan
(Epoch 9 / 10) train acc: 0.107000; val_acc: 0.087000
(Epoch 10 / 10) train acc: 0.083000; val_acc: 0.087000
best validation accuracy achieved during cross-validation: 0.601000
[0.5, 500, 0.001]
```

[71]:
```
# 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)
```

```
0.567
```