knn

November 10, 2022

[24]:
```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive')

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

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

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

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

1 k-Nearest Neighbor (kNN) exercise

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

The kNN classifier consists of two stages:

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

- The value of k is cross-validated

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

[25]:
```
# 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
```

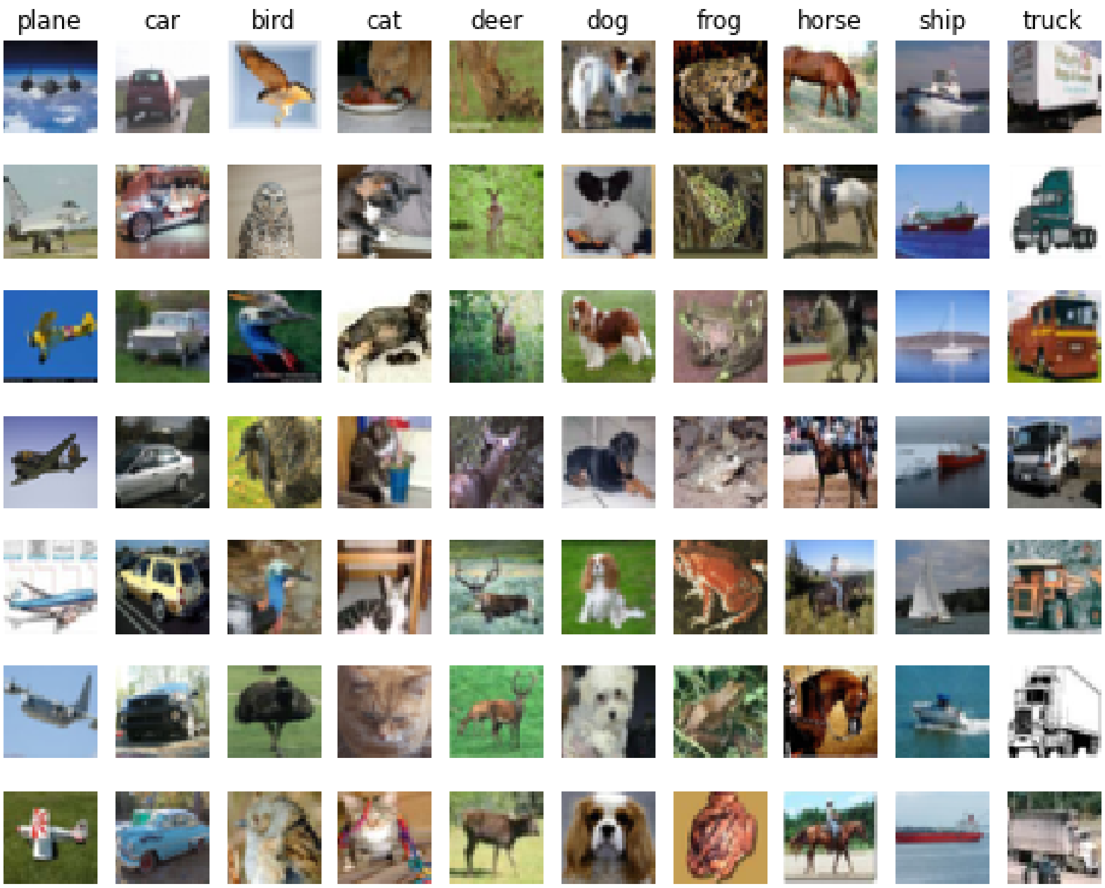
```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

[26]:
```
# 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,)
```

[27]:

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



[28]:

```
# 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)
```

[29]:
```python
from cs231n.classifiers import KNearestNeighbor

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

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

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

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

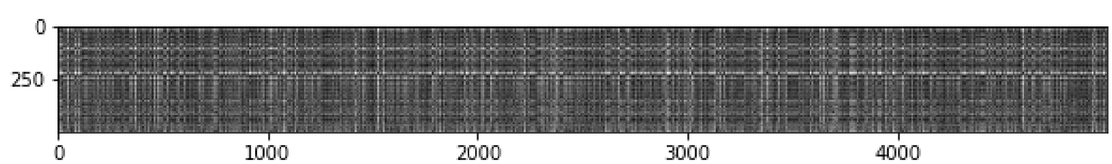
Note: For the three distance computations that we require you to implement in this notebook, you may not use the np.linalg.norm() function that numpy provides.

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

[22]:
```python
# Open cs231n/classifiers/k_nearest_neighbor.py and implement
# compute_distances_two_loops.

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

```
(500, 5000)
```

[9]:
```python
# We can visualize the distance matrix: each row is a single test example and
# its distances to training examples
plt.imshow(dists, interpolation='none')
plt.show()
```



Inline Question 1

Notice the structured patterns in the distance matrix, where some rows or columns are visibly brighter. (Note that with the default color scheme black indicates low distances while white indicates high distances.)

- What in the data is the cause behind the distinctly bright rows?
- What causes the columns?

$\color{blue}{\textit Your Answer:}$ *fill this in.*

[30]:
```
# 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`:

[31]:
```
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. 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.

$Your Answer:$

$Your Explanation:$

[12]:
```
# 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
```

[13]:
```
# 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
```

[14]:
```
# 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 42.183372 seconds
One loop version took 110.040659 seconds
No loop version took 1.062496 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.

[17]:
```
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)*****

range_split = np.array_split(range(X_train.shape[0]), num_folds)
```

```
y_train_folds = [ y_train[range_split[i]] for i in range(num_folds)]
X_train_folds = [ X_train[range_split[i]] for i in range(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:
    for fold in range(num_folds): #This fold will be omitted.
        #Creating validation data and temp training data
        validation_X_test = X_train_folds[fold]
        validation_y_test = y_train_folds[fold]
        temp_X_train = np.concatenate(X_train_folds[:fold] + X_train_folds[fold␣
 ↪+ 1:])
        temp_y_train = np.concatenate(y_train_folds[:fold] + y_train_folds[fold␣
 ↪+ 1:])

        #Initializing a class
        test_classifier = KNearestNeighbor()
        test_classifier.train( temp_X_train, temp_y_train )

        #Computing the distance
        temp_dists = test_classifier.
 ↪compute_distances_two_loops(validation_X_test)
        temp_y_test_pred = test_classifier.predict_labels(temp_dists, k=k)

        #Checking accuracies
        num_correct = np.sum(temp_y_test_pred == validation_y_test)
        num_test = validation_X_test.shape[0]
        accuracy = float(num_correct) / num_test
        print("k=",k,"Fold=",fold,"Accuracy=",accuracy)
        k_to_accuracies[k] = k_to_accuracies.get(k,[]) + [accuracy]
```

```python
# *****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))
```

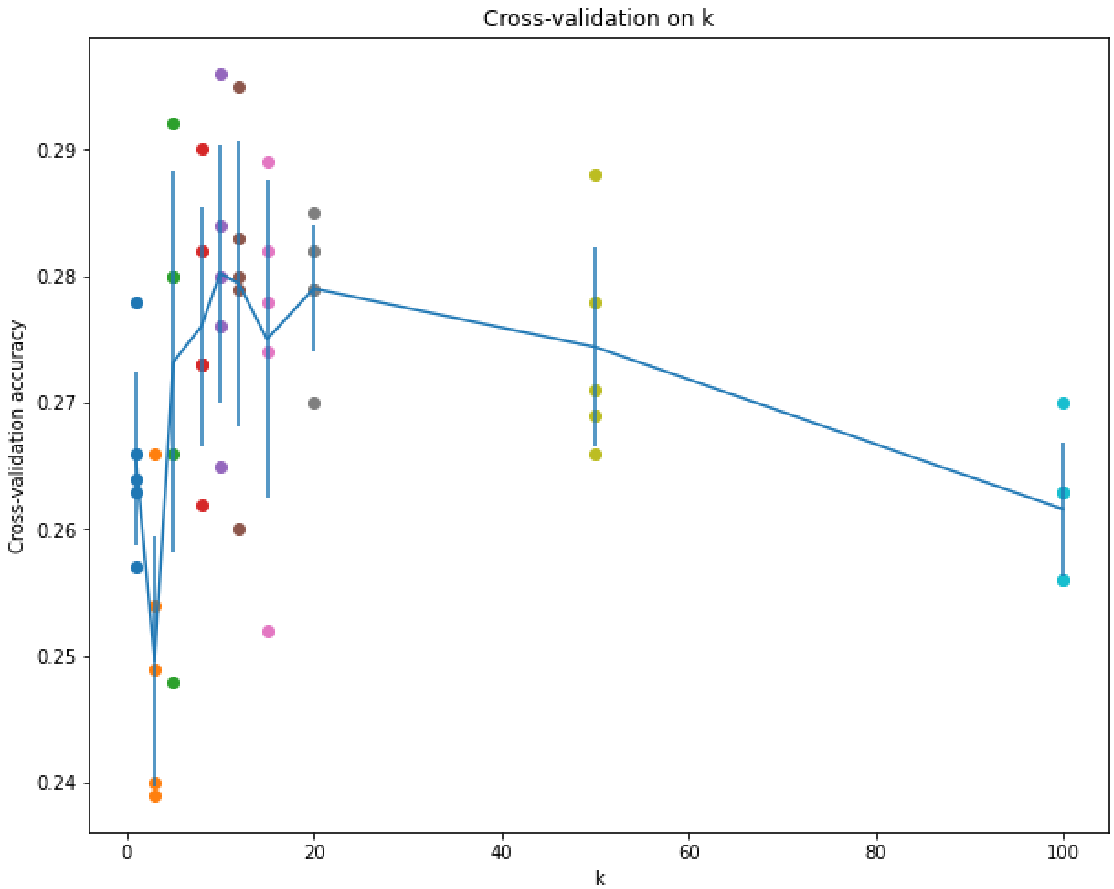
```
k= 1 Fold= 0 Accuracy= 0.263
k= 1 Fold= 1 Accuracy= 0.257
k= 1 Fold= 2 Accuracy= 0.264
k= 1 Fold= 3 Accuracy= 0.278
k= 1 Fold= 4 Accuracy= 0.266
k= 3 Fold= 0 Accuracy= 0.239
k= 3 Fold= 1 Accuracy= 0.249
k= 3 Fold= 2 Accuracy= 0.24
k= 3 Fold= 3 Accuracy= 0.266
k= 3 Fold= 4 Accuracy= 0.254
k= 5 Fold= 0 Accuracy= 0.248
k= 5 Fold= 1 Accuracy= 0.266
k= 5 Fold= 2 Accuracy= 0.28
k= 5 Fold= 3 Accuracy= 0.292
k= 5 Fold= 4 Accuracy= 0.28
k= 8 Fold= 0 Accuracy= 0.262
k= 8 Fold= 1 Accuracy= 0.282
k= 8 Fold= 2 Accuracy= 0.273
k= 8 Fold= 3 Accuracy= 0.29
k= 8 Fold= 4 Accuracy= 0.273
k= 10 Fold= 0 Accuracy= 0.265
k= 10 Fold= 1 Accuracy= 0.296
k= 10 Fold= 2 Accuracy= 0.276
k= 10 Fold= 3 Accuracy= 0.284
k= 10 Fold= 4 Accuracy= 0.28
k= 12 Fold= 0 Accuracy= 0.26
k= 12 Fold= 1 Accuracy= 0.295
k= 12 Fold= 2 Accuracy= 0.279
k= 12 Fold= 3 Accuracy= 0.283
k= 12 Fold= 4 Accuracy= 0.28
k= 15 Fold= 0 Accuracy= 0.252
k= 15 Fold= 1 Accuracy= 0.289
k= 15 Fold= 2 Accuracy= 0.278
k= 15 Fold= 3 Accuracy= 0.282
k= 15 Fold= 4 Accuracy= 0.274
k= 20 Fold= 0 Accuracy= 0.27
k= 20 Fold= 1 Accuracy= 0.279
k= 20 Fold= 2 Accuracy= 0.279
k= 20 Fold= 3 Accuracy= 0.282
```

```
k= 20 Fold= 4 Accuracy= 0.285
k= 50 Fold= 0 Accuracy= 0.271
k= 50 Fold= 1 Accuracy= 0.288
k= 50 Fold= 2 Accuracy= 0.278
k= 50 Fold= 3 Accuracy= 0.269
k= 50 Fold= 4 Accuracy= 0.266
k= 100 Fold= 0 Accuracy= 0.256
k= 100 Fold= 1 Accuracy= 0.27
k= 100 Fold= 2 Accuracy= 0.263
k= 100 Fold= 3 Accuracy= 0.256
k= 100 Fold= 4 Accuracy= 0.263
k = 1, accuracy = 0.263000
k = 1, accuracy = 0.257000
k = 1, accuracy = 0.264000
k = 1, accuracy = 0.278000
k = 1, accuracy = 0.266000
k = 3, accuracy = 0.239000
k = 3, accuracy = 0.249000
k = 3, accuracy = 0.240000
k = 3, accuracy = 0.266000
k = 3, accuracy = 0.254000
k = 5, accuracy = 0.248000
k = 5, accuracy = 0.266000
k = 5, accuracy = 0.280000
k = 5, accuracy = 0.292000
k = 5, accuracy = 0.280000
k = 8, accuracy = 0.262000
k = 8, accuracy = 0.282000
k = 8, accuracy = 0.273000
k = 8, accuracy = 0.290000
k = 8, accuracy = 0.273000
k = 10, accuracy = 0.265000
k = 10, accuracy = 0.296000
k = 10, accuracy = 0.276000
k = 10, accuracy = 0.284000
k = 10, accuracy = 0.280000
k = 12, accuracy = 0.260000
k = 12, accuracy = 0.295000
k = 12, accuracy = 0.279000
k = 12, accuracy = 0.283000
k = 12, accuracy = 0.280000
k = 15, accuracy = 0.252000
k = 15, accuracy = 0.289000
k = 15, accuracy = 0.278000
k = 15, accuracy = 0.282000
k = 15, accuracy = 0.274000
k = 20, accuracy = 0.270000
k = 20, accuracy = 0.279000
```

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

[18]:

```
# 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()
```



[38]:
```python
# Based on the cross-validation results above, choose the best value for k,
# retrain the classifier using all the training data, and test it on the test
# data. You should be able to get above 28% accuracy on the test data.
best_k = 10

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

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

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

Inline Question 3

Which of the following statements about k-Nearest Neighbor (k-NN) are true in a classification setting, and for all k? Select all that apply. 1. The decision boundary of the k-NN classifier is linear. 2. The training error of a 1-NN will always be lower than or equal to that of 5-NN. 3. The test error of a 1-NN will always be lower than that of a 5-NN. 4. The time needed to classify a test example with the k-NN classifier grows with the size of the training set. 5. None of the above.

Your Answer :

Your Explanation :

svm

November 10, 2022

[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
```

1 Multiclass Support Vector Machine exercise

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

In this exercise you will:

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

- **optimize** the loss function with **SGD**
- **visualize** the final learned weights

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

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

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

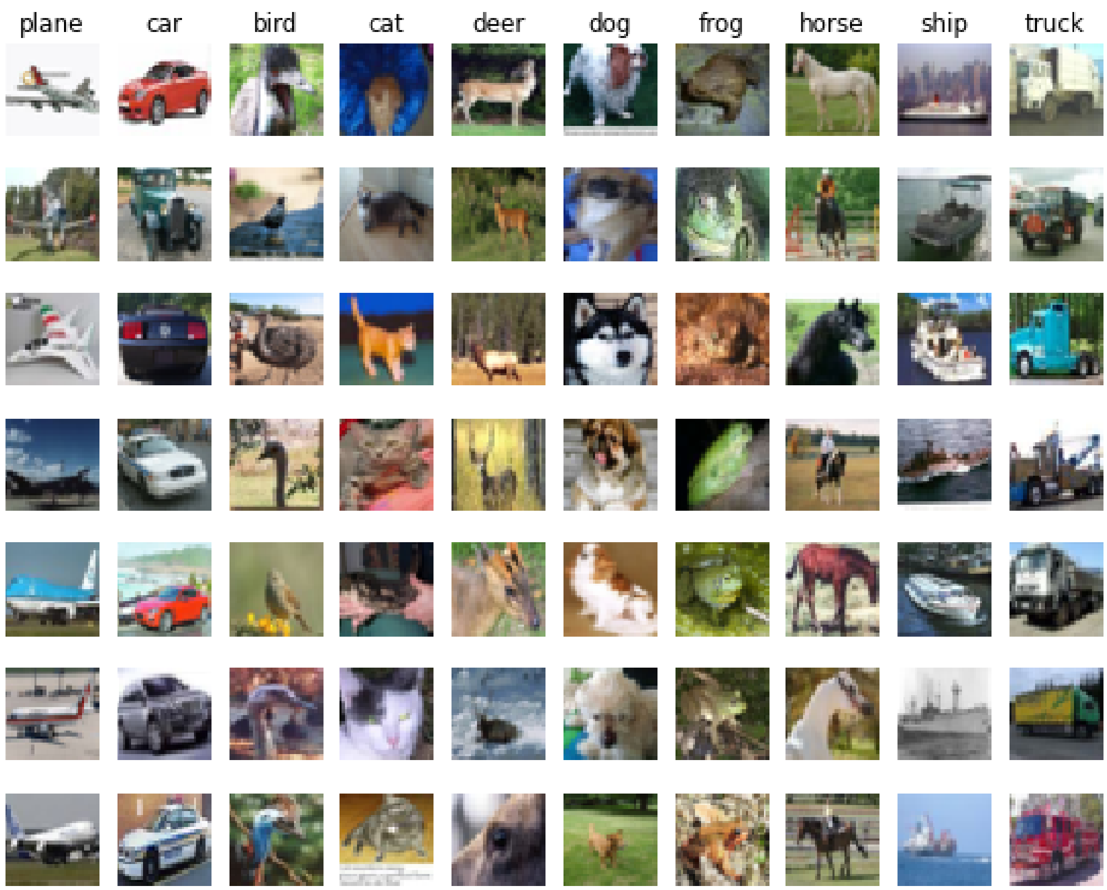
1.1 CIFAR-10 Data Loading and Preprocessing

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

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

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

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

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

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

[4]:
```python
# Visualize some examples from the dataset.
# We show a few examples of training images from each class.
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
 ↪'ship', 'truck']
num_classes = len(classes)
samples_per_class = 7
for y, cls in enumerate(classes):
    idxs = np.flatnonzero(y_train == y)
    idxs = np.random.choice(idxs, samples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt_idx = i * num_classes + y + 1
        plt.subplot(samples_per_class, num_classes, plt_idx)
        plt.imshow(X_train[idx].astype('uint8'))
        plt.axis('off')
        if i == 0:
            plt.title(cls)
plt.show()
```



[5]:

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

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

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

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

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

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

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

[6]:

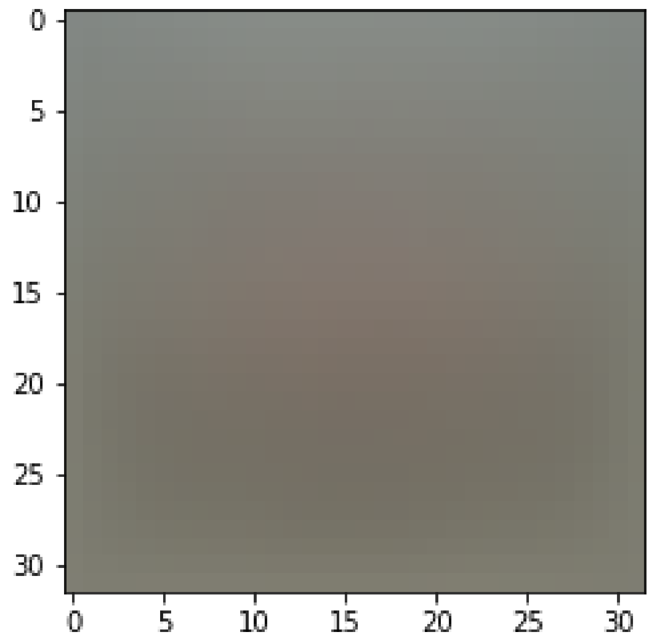
```python
# Preprocessing: reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_val = np.reshape(X_val, (X_val.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

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

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

[7]:

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

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

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

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

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



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

1.2 SVM Classifier

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

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

[13]:
```
# Evaluate the naive implementation of the loss we provided for you:
from cs231n.classifiers.linear_svm import svm_loss_naive
import time

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

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

```
loss: 9.172780
```

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:

[14]:
```
# Once you've implemented the gradient, recompute it with the code below
# and gradient check it with the function we provided for you

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

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

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

```
numerical: 10.979371 analytic: 10.979371, relative error: 3.189609e-11
numerical: -3.821736 analytic: -3.821736, relative error: 4.747824e-11
numerical: -3.867832 analytic: -3.867832, relative error: 7.553267e-11
numerical: -3.096097 analytic: -3.090560, relative error: 8.948578e-04
numerical: 1.807379 analytic: 1.807379, relative error: 1.120134e-10
numerical: -8.604955 analytic: -8.604955, relative error: 4.040640e-12
numerical: 8.756213 analytic: 8.756213, relative error: 1.282025e-11
numerical: 3.226249 analytic: 3.226249, relative error: 1.132287e-10
numerical: 7.207171 analytic: 7.207171, relative error: 1.640214e-11
numerical: -13.910436 analytic: -13.910436, relative error: 9.296529e-13
numerical: 8.581986 analytic: 8.581986, relative error: 1.823273e-11
numerical: 2.203453 analytic: 2.231025, relative error: 6.217572e-03
numerical: -16.491508 analytic: -16.491508, relative error: 1.754878e-11
numerical: 21.307451 analytic: 21.307451, relative error: 1.948891e-12
numerical: 16.642869 analytic: 16.642869, relative error: 7.325675e-12
numerical: -6.803547 analytic: -6.825709, relative error: 1.626117e-03
numerical: -3.734418 analytic: -3.734418, relative error: 6.896622e-11
numerical: -3.385117 analytic: -3.385117, relative error: 4.402176e-11
numerical: 12.913836 analytic: 12.913836, relative error: 9.009822e-12
numerical: -37.549233 analytic: -37.549233, relative error: 7.688400e-13
```

Inline Question 1

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

Your Answer : fill this in.

[17]:
```python
# Next implement the function svm_loss_vectorized; for now only compute the␣
 ↪loss;
# we will implement the gradient in a moment.
tic = time.time()
loss_naive, grad_naive = svm_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Naive loss: %e computed in %fs' % (loss_naive, toc - tic))

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

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

```
Naive loss: 9.172780e+00 computed in 0.125294s
Vectorized loss: 9.172780e+00 computed in 0.023001s
difference: -0.000000
```

[18]:
```python
# 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 0.125427s
Vectorized loss and gradient: computed in 0.016052s
difference: 0.000000
```

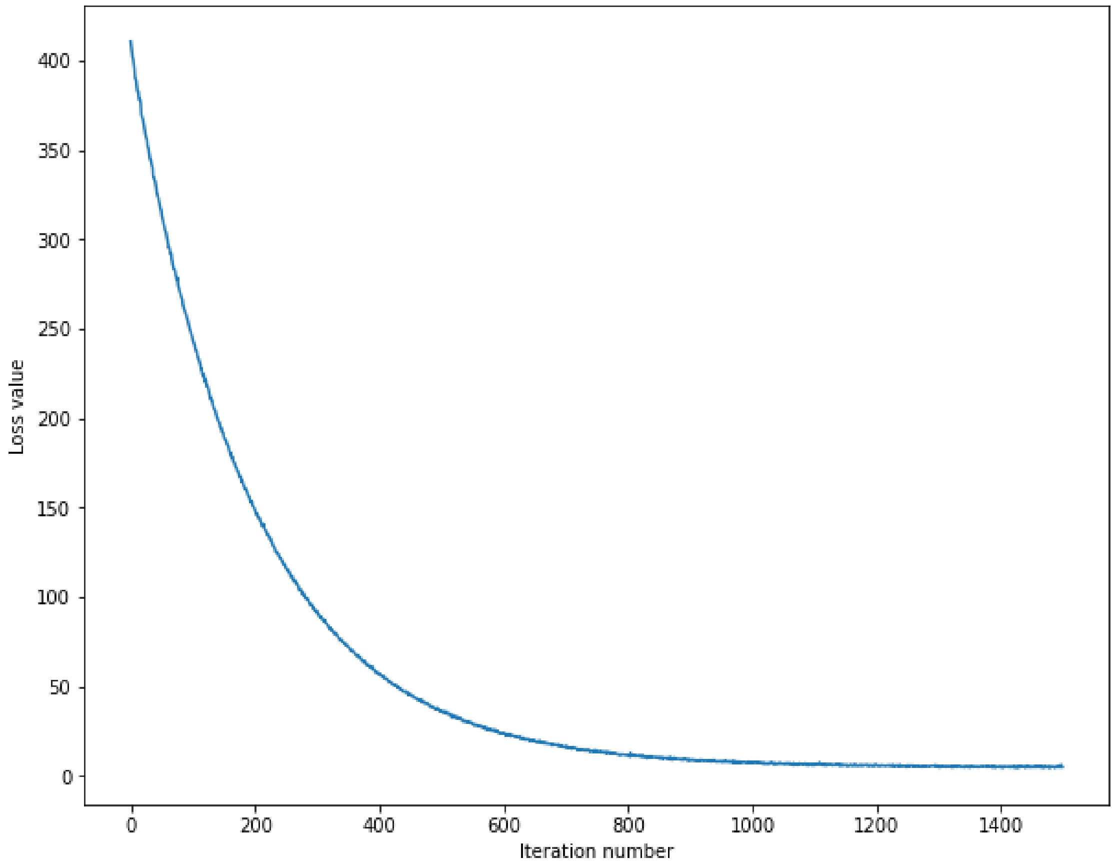
1.2.1 Stochastic Gradient Descent

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`.

[19]:
```
# 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 410.477200
iteration 100 / 1500: loss 243.358417
iteration 200 / 1500: loss 148.528906
iteration 300 / 1500: loss 92.254450
iteration 400 / 1500: loss 57.417909
iteration 500 / 1500: loss 36.021097
iteration 600 / 1500: loss 24.413140
iteration 700 / 1500: loss 16.686311
iteration 800 / 1500: loss 11.351255
iteration 900 / 1500: loss 9.095025
iteration 1000 / 1500: loss 7.253724
iteration 1100 / 1500: loss 6.030606
iteration 1200 / 1500: loss 6.296263
iteration 1300 / 1500: loss 4.962865
iteration 1400 / 1500: loss 5.010737
That took 13.651752s
```

[20]:
```
# 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()
```



[21]:
```
# 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: 0.375388
validation accuracy: 0.379000
```

[22]:
```
# 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 learning_rate in learning_rates:
    for reg in regularization_strengths:
        print("LR",learning_rate,"reg",reg)
        svm = LinearSVM()
        loss_hist = svm.train(X_train, y_train, learning_rate=learning_rate,␣
↪reg=reg,
                      num_iters=1500, verbose=True)
        y_train_pred = svm.predict(X_train)
        y_val_pred = svm.predict(X_val)
        results[(learning_rate,reg)] = (np.mean(y_train == y_train_pred),np.
↪mean(y_val == y_val_pred))

        if best_val < np.mean(y_val == y_val_pred):
            best_val = np.mean(y_val == y_val_pred)
            best_parameters = { 'LR':learning_rate, 'reg': reg}

print("Best val accuracy", best_val)
```

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

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

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

```
LR 1e-07 reg 25000.0
iteration 0 / 1500: loss 406.901259
iteration 100 / 1500: loss 239.898303
iteration 200 / 1500: loss 147.280727
iteration 300 / 1500: loss 90.169770
iteration 400 / 1500: loss 56.957639
iteration 500 / 1500: loss 36.162667
iteration 600 / 1500: loss 23.891907
iteration 700 / 1500: loss 16.252905
iteration 800 / 1500: loss 11.898191
iteration 900 / 1500: loss 9.221702
iteration 1000 / 1500: loss 7.380308
iteration 1100 / 1500: loss 6.741648
iteration 1200 / 1500: loss 6.566321
iteration 1300 / 1500: loss 5.064031
iteration 1400 / 1500: loss 5.073536
LR 1e-07 reg 50000.0
iteration 0 / 1500: loss 791.253062
iteration 100 / 1500: loss 288.762093
iteration 200 / 1500: loss 108.453681
iteration 300 / 1500: loss 43.038430
iteration 400 / 1500: loss 19.054303
iteration 500 / 1500: loss 10.762531
iteration 600 / 1500: loss 6.909195
iteration 700 / 1500: loss 5.506306
iteration 800 / 1500: loss 5.858848
iteration 900 / 1500: loss 5.008105
iteration 1000 / 1500: loss 5.169315
iteration 1100 / 1500: loss 5.482904
iteration 1200 / 1500: loss 4.817158
iteration 1300 / 1500: loss 5.283412
iteration 1400 / 1500: loss 5.860334
LR 5e-05 reg 25000.0
iteration 0 / 1500: loss 401.690977
iteration 100 / 1500: loss 1099.459579
iteration 200 / 1500: loss 998.378629
```

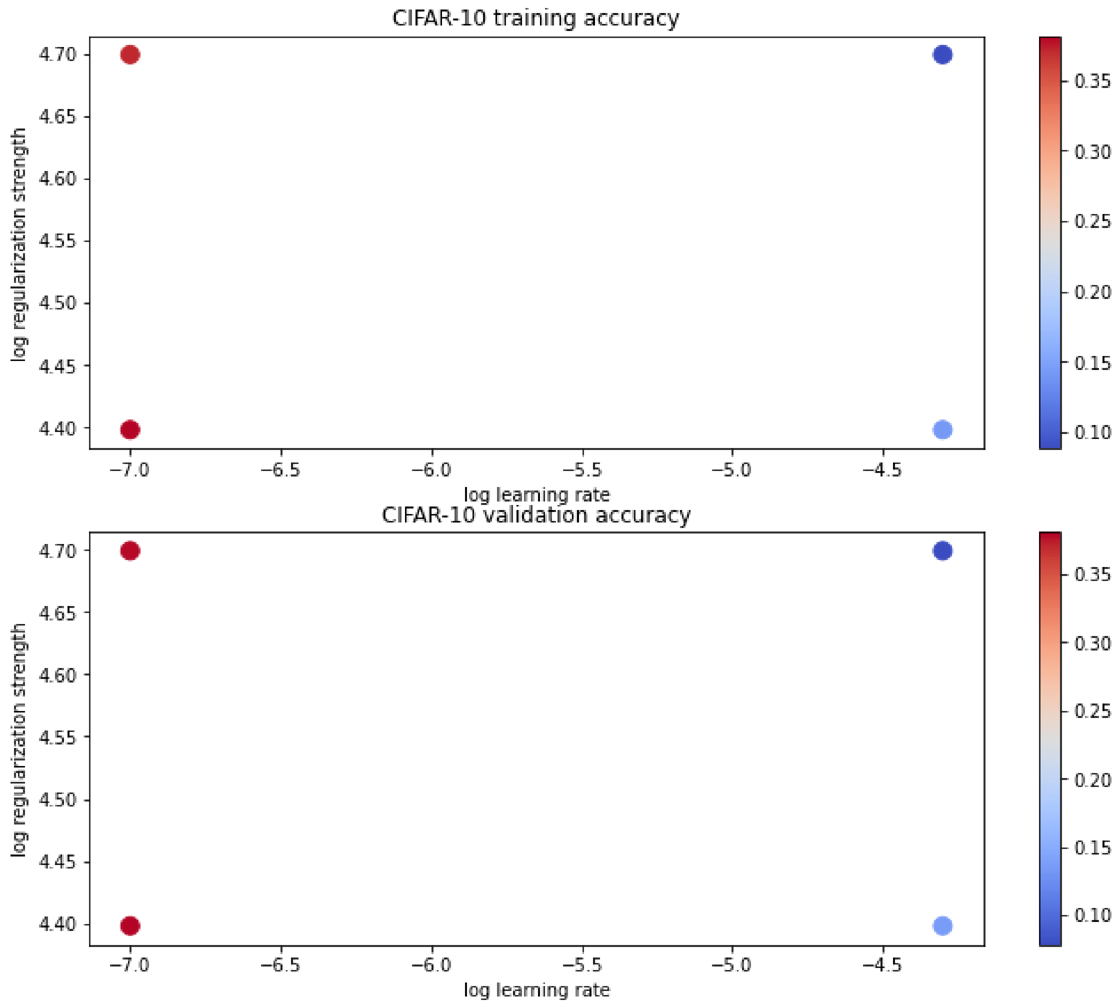
```
iteration 300 / 1500: loss 1156.957409
iteration 400 / 1500: loss 809.958447
iteration 500 / 1500: loss 860.527290
iteration 600 / 1500: loss 1028.967834
iteration 700 / 1500: loss 751.238270
iteration 800 / 1500: loss 1022.165514
iteration 900 / 1500: loss 963.141308
iteration 1000 / 1500: loss 1142.661873
iteration 1100 / 1500: loss 1044.673998
iteration 1200 / 1500: loss 722.942613
iteration 1300 / 1500: loss 911.233781
iteration 1400 / 1500: loss 1012.000636
LR 5e-05 reg 50000.0
iteration 0 / 1500: loss 790.307436
iteration 100 / 1500: loss 41708696978631227925820175333519995 6992.000000
iteration 200 / 1500: loss 6894113568901262822182094808778981826058661520903389 7
5747997788426354 03264.000000
```

```
  margin[margin>0]=1
```

```
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
Best val accuracy 0.381
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.381204 val accuracy: 0.381000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.371204 val accuracy: 0.379000
lr 5.000000e-05 reg 2.500000e+04 train accuracy: 0.142082 val accuracy: 0.137000
lr 5.000000e-05 reg 5.000000e+04 train accuracy: 0.088143 val accuracy: 0.078000
best validation accuracy achieved during cross-validation: 0.381000
```

[23]:

```
# 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()
```



[24]:
```
# Evaluate the best svm on test set
best_svm = LinearSVM()
best_svm.train(X_train, y_train, learning_rate=best_parameters['LR'],␣
 ↪reg=best_parameters['reg'],
                      num_iters=2000, verbose=True)
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)
```

```
iteration 0 / 2000: loss 402.336009
iteration 100 / 2000: loss 238.568838
iteration 200 / 2000: loss 145.871873
iteration 300 / 2000: loss 90.205634
iteration 400 / 2000: loss 56.496478
iteration 500 / 2000: loss 35.537156
iteration 600 / 2000: loss 24.208079
```

```
iteration 700 / 2000: loss 16.191145
iteration 800 / 2000: loss 11.425611
iteration 900 / 2000: loss 9.051570
iteration 1000 / 2000: loss 7.778282
iteration 1100 / 2000: loss 6.460719
iteration 1200 / 2000: loss 5.850448
iteration 1300 / 2000: loss 5.589773
iteration 1400 / 2000: loss 5.455875
iteration 1500 / 2000: loss 5.023243
iteration 1600 / 2000: loss 5.108862
iteration 1700 / 2000: loss 4.582140
iteration 1800 / 2000: loss 4.664795
iteration 1900 / 2000: loss 4.672989
linear SVM on raw pixels final test set accuracy: 0.380000
```

[25]:

```
# Visualize the learned weights for each class.
# Depending on your choice of learning rate and regularization strength, these␣
↪may
# or may not be nice to look at.
w = best_svm.W[:-1,:] # strip out the bias
w = w.reshape(32, 32, 3, 10)
w_min, w_max = np.min(w), np.max(w)
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
↪'ship', 'truck']
for i in range(10):
    plt.subplot(2, 5, i + 1)

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

Inline question 2

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

Your Answer : *fill this in*

softmax

November 10, 2022

[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
```

1 Softmax exercise

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

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

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

- **optimize** the loss function with **SGD**
- **visualize** the final learned weights

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

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

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

[3]:
```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000,␣
↪num_dev=500):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    it for the linear classifier. These are the same steps as we used for the
    SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

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

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

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

```python
    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`.

[4]:
```python
# First implement the naive softmax loss function with nested loops.
# Open the file cs231n/classifiers/softmax.py and implement the
# softmax_loss_naive function.

from cs231n.classifiers.softmax import softmax_loss_naive
import time

# Generate a random softmax weight matrix and use it to compute the loss.
W = np.random.randn(3073, 10) * 0.0001
loss, grad = softmax_loss_naive(W, X_dev, y_dev, 0.0)

# As a rough sanity check, our loss should be something close to -log(0.1).
print('loss: %f' % loss)
print('sanity check: %f' % (-np.log(0.1)))
```

```
loss: 2.327009
sanity check: 2.302585
```

Inline Question 1

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

$\color{blue}{\textit{Your Answer:}}$ *Fill this in*

[5]:
```python
# Complete the implementation of softmax_loss_naive and implement a (naive)
# version of the gradient that uses nested loops.
loss, grad = softmax_loss_naive(W, X_dev, y_dev, 0.0)

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

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

```
numerical: 1.101466 analytic: 1.101466, relative error: 1.883183e-08
numerical: 1.790404 analytic: 1.790404, relative error: 2.715073e-08
numerical: 2.671292 analytic: 2.671292, relative error: 3.496274e-09
```

```
numerical: 2.792409 analytic: 2.792409, relative error: 6.310009e-09
numerical: -2.682267 analytic: -2.682267, relative error: 3.167293e-08
numerical: -1.679300 analytic: -1.679300, relative error: 3.076206e-08
numerical: 0.997521 analytic: 0.997521, relative error: 5.719311e-08
numerical: 1.085600 analytic: 1.085600, relative error: 1.670923e-08
numerical: -3.271831 analytic: -3.271831, relative error: 2.099695e-08
numerical: 2.006267 analytic: 2.006267, relative error: 2.933419e-08
numerical: -2.365365 analytic: -2.365365, relative error: 2.138399e-08
numerical: 0.131685 analytic: 0.131685, relative error: 3.606496e-09
numerical: 0.737901 analytic: 0.737901, relative error: 8.478608e-08
numerical: 1.053614 analytic: 1.053614, relative error: 2.477162e-08
numerical: 5.081056 analytic: 5.081056, relative error: 2.172598e-08
numerical: 0.783212 analytic: 0.783212, relative error: 5.470968e-08
numerical: 1.913388 analytic: 1.913388, relative error: 9.472867e-09
numerical: -2.908479 analytic: -2.908480, relative error: 2.308367e-08
numerical: 1.879369 analytic: 1.879369, relative error: 2.034962e-08
numerical: 1.693029 analytic: 1.693029, relative error: 2.978806e-10
```

[6]:

```python
# Now that we have a naive implementation of the softmax loss function and its␣
↪gradient,
# implement a vectorized version in softmax_loss_vectorized.
# The two versions should compute the same results, but the vectorized version␣
↪should be
# much faster.
tic = time.time()
loss_naive, grad_naive = softmax_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('naive loss: %e computed in %fs' % (loss_naive, toc - tic))

from cs231n.classifiers.softmax import softmax_loss_vectorized
tic = time.time()
loss_vectorized, grad_vectorized = softmax_loss_vectorized(W, X_dev, y_dev, 0.
↪000005)
toc = time.time()
print('vectorized loss: %e computed in %fs' % (loss_vectorized, toc - tic))

# As we did for the SVM, we use the Frobenius norm to compare the two versions
# of the gradient.
grad_difference = np.linalg.norm(grad_naive - grad_vectorized, ord='fro')
print('Loss difference: %f' % np.abs(loss_naive - loss_vectorized))
print('Gradient difference: %f' % grad_difference)
```

```
naive loss: 2.327009e+00 computed in 0.173111s
vectorized loss: 2.327009e+00 computed in 0.023303s
Loss difference: 0.000000
Gradient difference: 0.000000
```

[7]:

```python
# 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 learning_rate in learning_rates:
    for reg in regularization_strengths:
        sftmax = Softmax()
        loss_hist = sftmax.train(X_train, y_train, learning_rate=learning_rate,␣
↪reg=reg,
                      num_iters=1500, verbose=True)
        y_train_pred = sftmax.predict(X_train)
        y_val_pred = sftmax.predict(X_val)
        results[(learning_rate,reg)] = (np.mean(y_train == y_train_pred),np.
↪mean(y_val == y_val_pred))
        if best_val < np.mean(y_val == y_val_pred):
            best_val = np.mean(y_val == y_val_pred)
            best_parameters = { 'LR':learning_rate, 'reg': reg}


# *****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))
```

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

```
iteration 0 / 1500: loss 387.793519
iteration 100 / 1500: loss 234.088450
iteration 200 / 1500: loss 142.254388
iteration 300 / 1500: loss 86.724548
iteration 400 / 1500: loss 53.273377
iteration 500 / 1500: loss 32.975647
iteration 600 / 1500: loss 20.749008
iteration 700 / 1500: loss 13.480063
iteration 800 / 1500: loss 8.869713
iteration 900 / 1500: loss 6.193296
iteration 1000 / 1500: loss 4.432800
iteration 1100 / 1500: loss 3.579351
iteration 1200 / 1500: loss 3.010070
iteration 1300 / 1500: loss 2.555300
iteration 1400 / 1500: loss 2.359951
iteration 0 / 1500: loss 769.787863
iteration 100 / 1500: loss 282.413454
iteration 200 / 1500: loss 104.552185
iteration 300 / 1500: loss 39.606911
iteration 400 / 1500: loss 15.814729
iteration 500 / 1500: loss 7.129013
iteration 600 / 1500: loss 3.887625
iteration 700 / 1500: loss 2.832489
iteration 800 / 1500: loss 2.273453
iteration 900 / 1500: loss 2.174075
iteration 1000 / 1500: loss 2.104635
iteration 1100 / 1500: loss 2.091009
iteration 1200 / 1500: loss 2.142139
iteration 1300 / 1500: loss 2.089678
iteration 1400 / 1500: loss 2.056556
iteration 0 / 1500: loss 394.629411
iteration 100 / 1500: loss 33.141627
iteration 200 / 1500: loss 4.601970
iteration 300 / 1500: loss 2.212404
iteration 400 / 1500: loss 2.100111
iteration 500 / 1500: loss 1.966080
iteration 600 / 1500: loss 1.943473
iteration 700 / 1500: loss 2.064857
iteration 800 / 1500: loss 2.003126
iteration 900 / 1500: loss 2.041093
iteration 1000 / 1500: loss 1.993464
iteration 1100 / 1500: loss 2.099648
iteration 1200 / 1500: loss 1.981934
iteration 1300 / 1500: loss 2.104337
```

```
iteration 1400 / 1500: loss 1.984901
iteration 0 / 1500: loss 762.611268
iteration 100 / 1500: loss 6.765057
iteration 200 / 1500: loss 2.126556
iteration 300 / 1500: loss 2.080210
iteration 400 / 1500: loss 2.087085
iteration 500 / 1500: loss 2.045454
iteration 600 / 1500: loss 2.100839
iteration 700 / 1500: loss 2.127662
iteration 800 / 1500: loss 2.130292
iteration 900 / 1500: loss 2.095461
iteration 1000 / 1500: loss 2.095422
iteration 1100 / 1500: loss 2.068251
iteration 1200 / 1500: loss 2.119629
iteration 1300 / 1500: loss 2.101740
iteration 1400 / 1500: loss 2.102824
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.350020 val accuracy: 0.354000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.326551 val accuracy: 0.340000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.345449 val accuracy: 0.341000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.326020 val accuracy: 0.336000
best validation accuracy achieved during cross-validation: 0.354000
```

[11]:
```
# evaluate on test set
# Evaluate the best softmax on test set
best_softmax = Softmax()
best_softmax.train(X_train, y_train, learning_rate=best_parameters['LR'],␣
↪reg=best_parameters['reg'],
                      num_iters=3000, verbose=True)
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, ))
```

```
iteration 0 / 3000: loss 392.241092
iteration 100 / 3000: loss 237.342817
iteration 200 / 3000: loss 144.106301
iteration 300 / 3000: loss 88.120557
iteration 400 / 3000: loss 53.990622
iteration 500 / 3000: loss 33.391728
iteration 600 / 3000: loss 20.975196
iteration 700 / 3000: loss 13.494400
iteration 800 / 3000: loss 8.944619
iteration 900 / 3000: loss 6.226601
iteration 1000 / 3000: loss 4.513601
iteration 1100 / 3000: loss 3.568951
iteration 1200 / 3000: loss 3.051907
iteration 1300 / 3000: loss 2.575779
iteration 1400 / 3000: loss 2.346855
iteration 1500 / 3000: loss 2.253385
```

```
iteration 1600 / 3000: loss 2.158476
iteration 1700 / 3000: loss 2.074573
iteration 1800 / 3000: loss 2.038975
iteration 1900 / 3000: loss 2.031261
iteration 2000 / 3000: loss 2.003653
iteration 2100 / 3000: loss 2.049747
iteration 2200 / 3000: loss 2.024377
iteration 2300 / 3000: loss 2.084902
iteration 2400 / 3000: loss 2.000918
iteration 2500 / 3000: loss 1.983169
iteration 2600 / 3000: loss 2.051688
iteration 2700 / 3000: loss 1.993804
iteration 2800 / 3000: loss 1.967397
iteration 2900 / 3000: loss 1.980580
softmax on raw pixels final test set accuracy: 0.359000
```

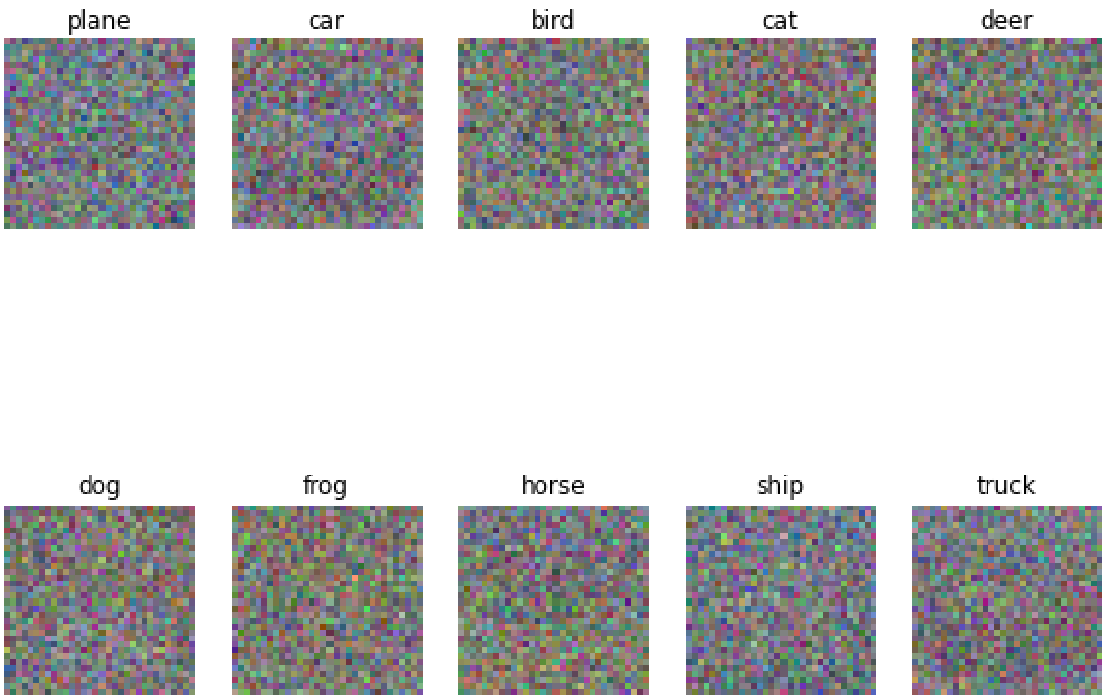
Inline Question 2 - *True or False*

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.

Your Answer :

Your Explanation :

[9]:
```
# Visualize the learned weights for each class
w = best_softmax.W[:-1,:] # strip out the bias
w = w.reshape(32, 32, 3, 10)

w_min, w_max = np.min(w), np.max(w)

classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
↪'ship', 'truck']
for i in range(10):
    plt.subplot(2, 5, i + 1)

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


plane
car
bird
cat
deer
dog
frog
horse
ship
truck

```
[ ]:
```

two_layer_net

November 10, 2022

[4]:
```
# 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.

[5]:
```
# 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
```

[6]:
```
# 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:

[7]:
```
# 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.

[8]:
```python
# 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:

[9]:
```python
# 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:

[10]:
```
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:

[FILL THIS IN]

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:

[11]:
```
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:

[12]:
```
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:  9.384673161989355e-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.

[13]:
```
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: 3.12e-10
b1 relative error: 9.83e-09
b2 relative error: 4.33e-10
Running numeric gradient check with reg =  0.7
W1 relative error: 2.53e-07
W2 relative error: 2.85e-08
b1 relative error: 1.56e-08
b2 relative error: 7.76e-10
```

9 Solver

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

[14]:
```
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
model = TwoLayerNet(input_size, hidden_size, num_classes)
solver = Solver(model, data, optim_config={"learning_rate":0.001},␣
 ↪print_every=500)


##############################################################################
# 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.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
(Iteration 501 / 4900) loss: 1.712717
(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
(Iteration 1501 / 4900) loss: 1.319916
(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
(Iteration 2501 / 4900) loss: 1.229392
(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
(Iteration 3501 / 4900) loss: 1.492338
(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
(Iteration 4501 / 4900) loss: 1.394010
(Epoch 10 / 10) train acc: 0.524000; val_acc: 0.456000
```

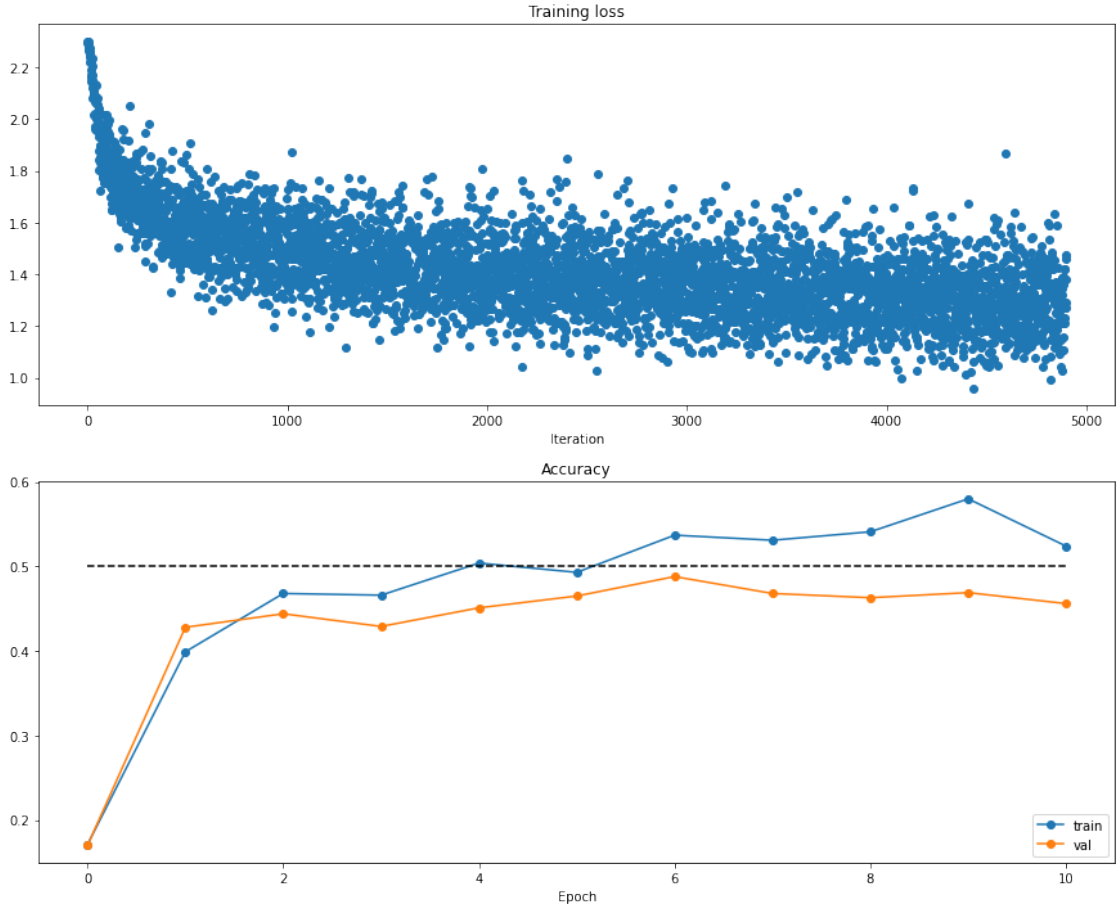
10 Debug the training

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.

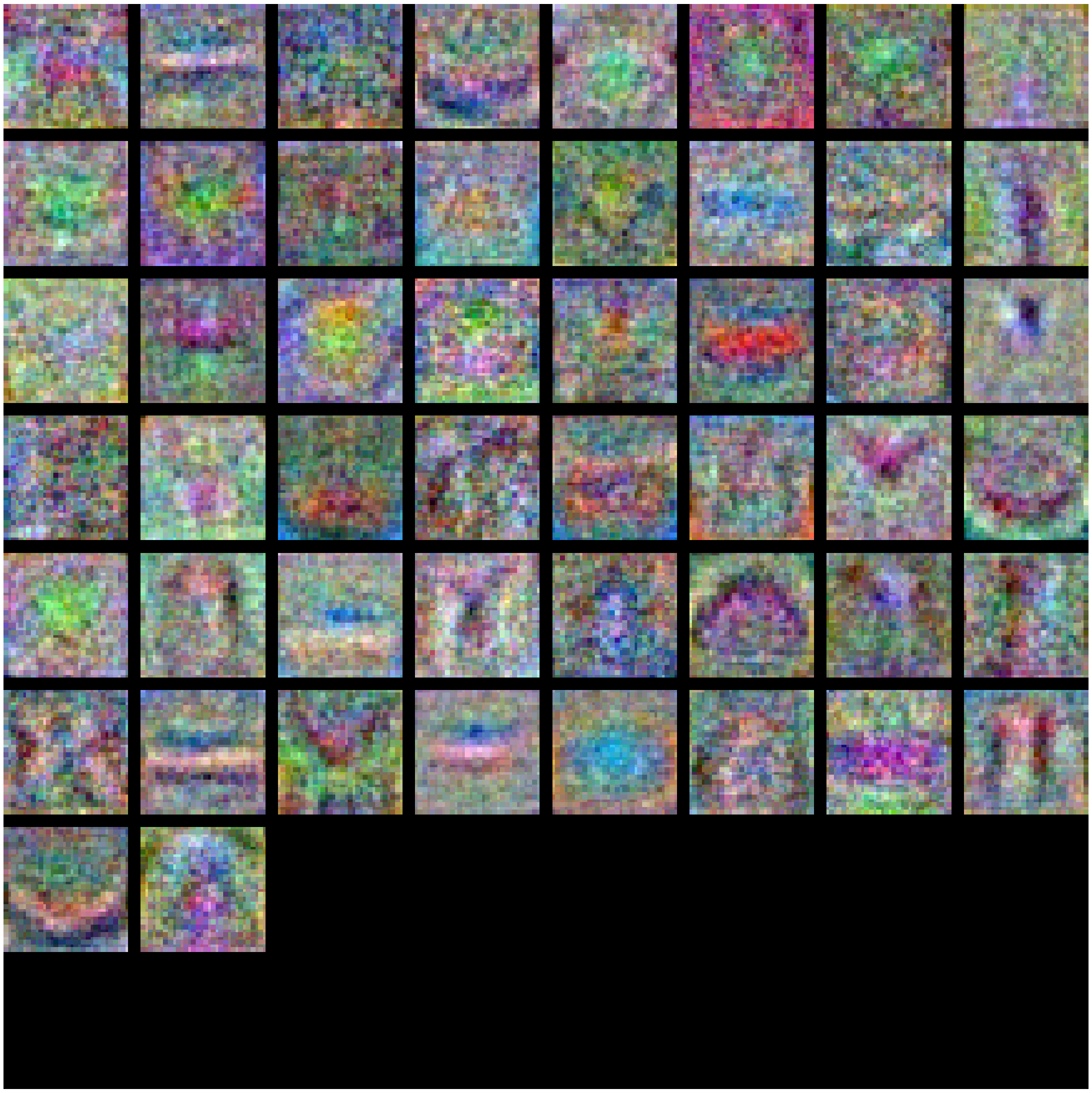
One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

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.

[15]:
```python
# Run this cell to visualize training loss and train / val accuracy
from scipy.stats import norm

norm.cdf(5)
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()
```



[16]:
```python
from cs231n.vis_utils import visualize_grid

# Visualize the weights of the network

def show_net_weights(net):
    W1 = net.params['W1']
    W1 = W1.reshape(3, 32, 32, -1).transpose(3, 1, 2, 0)
    plt.imshow(visualize_grid(W1, padding=3).astype('uint8'))
    plt.gca().axis('off')
    plt.show()

show_net_weights(model)
```



11 Tune your hyperparameters

What's wrong?. Looking at the visualizations above, we see that the loss is decreasing more or less linearly, which seems to suggest that the learning rate may be too low. Moreover, there is no gap between the training and validation accuracy, suggesting that the model we used has low capacity, and that we should increase its size. On the other hand, with a very large model we would expect to see more overfitting, which would manifest itself as a very large gap between the training and validation accuracy.

Tuning. Tuning the hyperparameters and developing intuition for how they affect the final performance is a large part of using Neural Networks, so we want you to get a lot of practice. Below, you should experiment with different values of the various hyperparameters, including hidden layer size, learning rate, numer of training epochs, and regularization strength. You might also consider tuning the learning rate decay, but you should be able to get good performance using the default value.

Approximate results. You should be aim to achieve a classification accuracy of greater than 48% on the validation set. Our best network gets over 52% on the validation set.

Experiment: You goal in this exercise is to get as good of a result on CIFAR-10 as you can (52% could serve as a reference), with a fully-connected Neural Network. Feel free implement your own techniques (e.g. PCA to reduce dimensionality, or adding dropout, or adding features to the solver, etc.).

[17]:
```
best_model = None


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

input_size = 32 * 32 * 3
hidden_size = 165
num_classes = 10
model = TwoLayerNet(input_size, hidden_size, num_classes)
solver = Solver(model, data,num_epochs=50,print_every=100,
    batch_size=1000,update_rule="adam",
    optim_config={'learning_rate': 0.005},
    verbose=True)
```

```python
solver.train()
best_model = model

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

```
(Iteration 1 / 2450) loss: 2.302503
(Epoch 0 / 50) train acc: 0.157000; val_acc: 0.147000
(Epoch 1 / 50) train acc: 0.291000; val_acc: 0.286000
(Epoch 2 / 50) train acc: 0.366000; val_acc: 0.361000
(Iteration 101 / 2450) loss: 2.215852
(Epoch 3 / 50) train acc: 0.448000; val_acc: 0.432000
(Epoch 4 / 50) train acc: 0.523000; val_acc: 0.483000
(Iteration 201 / 2450) loss: 1.375990
(Epoch 5 / 50) train acc: 0.548000; val_acc: 0.494000
(Epoch 6 / 50) train acc: 0.548000; val_acc: 0.492000
(Iteration 301 / 2450) loss: 1.298325
(Epoch 7 / 50) train acc: 0.565000; val_acc: 0.471000
(Epoch 8 / 50) train acc: 0.577000; val_acc: 0.490000
(Iteration 401 / 2450) loss: 1.272602
(Epoch 9 / 50) train acc: 0.554000; val_acc: 0.495000
(Epoch 10 / 50) train acc: 0.571000; val_acc: 0.482000
(Iteration 501 / 2450) loss: 1.161620
(Epoch 11 / 50) train acc: 0.528000; val_acc: 0.460000
(Epoch 12 / 50) train acc: 0.567000; val_acc: 0.474000
(Iteration 601 / 2450) loss: 1.276201
(Epoch 13 / 50) train acc: 0.540000; val_acc: 0.454000
(Epoch 14 / 50) train acc: 0.454000; val_acc: 0.416000
(Iteration 701 / 2450) loss: 3.570605
(Epoch 15 / 50) train acc: 0.165000; val_acc: 0.142000
(Epoch 16 / 50) train acc: 0.277000; val_acc: 0.308000
(Iteration 801 / 2450) loss: 38.132283
(Epoch 17 / 50) train acc: 0.385000; val_acc: 0.362000
(Epoch 18 / 50) train acc: 0.345000; val_acc: 0.374000
(Iteration 901 / 2450) loss: 3.589730
(Epoch 19 / 50) train acc: 0.421000; val_acc: 0.393000
(Epoch 20 / 50) train acc: 0.428000; val_acc: 0.451000
(Iteration 1001 / 2450) loss: 1.641311
(Epoch 21 / 50) train acc: 0.508000; val_acc: 0.446000
(Epoch 22 / 50) train acc: 0.451000; val_acc: 0.405000
(Iteration 1101 / 2450) loss: 1.559979
(Epoch 23 / 50) train acc: 0.504000; val_acc: 0.457000
(Epoch 24 / 50) train acc: 0.527000; val_acc: 0.484000
(Iteration 1201 / 2450) loss: 1.329939
(Epoch 25 / 50) train acc: 0.577000; val_acc: 0.481000
```

```
(Epoch 26 / 50) train acc: 0.546000; val_acc: 0.463000
(Iteration 1301 / 2450) loss: 1.442567
(Epoch 27 / 50) train acc: 0.585000; val_acc: 0.478000
(Epoch 28 / 50) train acc: 0.605000; val_acc: 0.475000
(Iteration 1401 / 2450) loss: 1.174232
(Epoch 29 / 50) train acc: 0.600000; val_acc: 0.484000
(Epoch 30 / 50) train acc: 0.609000; val_acc: 0.510000
(Iteration 1501 / 2450) loss: 1.059486
(Epoch 31 / 50) train acc: 0.641000; val_acc: 0.488000
(Epoch 32 / 50) train acc: 0.652000; val_acc: 0.475000
(Iteration 1601 / 2450) loss: 1.101880
(Epoch 33 / 50) train acc: 0.610000; val_acc: 0.477000
(Epoch 34 / 50) train acc: 0.648000; val_acc: 0.493000
(Iteration 1701 / 2450) loss: 1.095887
(Epoch 35 / 50) train acc: 0.661000; val_acc: 0.513000
(Epoch 36 / 50) train acc: 0.646000; val_acc: 0.488000
(Iteration 1801 / 2450) loss: 0.959368
(Epoch 37 / 50) train acc: 0.614000; val_acc: 0.483000
(Epoch 38 / 50) train acc: 0.656000; val_acc: 0.517000
(Iteration 1901 / 2450) loss: 1.089022
(Epoch 39 / 50) train acc: 0.649000; val_acc: 0.450000
(Epoch 40 / 50) train acc: 0.665000; val_acc: 0.483000
(Iteration 2001 / 2450) loss: 0.973512
(Epoch 41 / 50) train acc: 0.659000; val_acc: 0.494000
(Epoch 42 / 50) train acc: 0.627000; val_acc: 0.490000
(Iteration 2101 / 2450) loss: 1.128203
(Epoch 43 / 50) train acc: 0.654000; val_acc: 0.488000
(Epoch 44 / 50) train acc: 0.662000; val_acc: 0.487000
(Iteration 2201 / 2450) loss: 0.924563
(Epoch 45 / 50) train acc: 0.673000; val_acc: 0.486000
(Epoch 46 / 50) train acc: 0.613000; val_acc: 0.452000
(Iteration 2301 / 2450) loss: 1.122891
(Epoch 47 / 50) train acc: 0.635000; val_acc: 0.457000
(Epoch 48 / 50) train acc: 0.613000; val_acc: 0.493000
(Iteration 2401 / 2450) loss: 1.209126
(Epoch 49 / 50) train acc: 0.646000; val_acc: 0.498000
(Epoch 50 / 50) train acc: 0.543000; val_acc: 0.404000
```

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.

[18]:
```python
best_model=model
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:  0.517
```

[19]:
```
y_test_pred = np.argmax(best_model.loss(data['X_test']), axis=1)
print('Test set accuracy: ', (y_test_pred == data['y_test']).mean())
```

```
Test set accuracy:  0.493
```

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.

$\textit{Your Answer :}$

$\textit{Your Explanation :}$

[19]:
```

```

features

November 10, 2022

[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
```

1 Image features exercise

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

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

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

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


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

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

1.1 Load data

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

[3]:
```
from cs231n.features import color_histogram_hsv, hog_feature

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

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

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

    # Subsample the data
    mask = list(range(num_training, num_training + num_validation))
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = list(range(num_training))
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = list(range(num_test))
```

```
    X_test = X_test[mask]
    y_test = y_test[mask]

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

X_train, y_train, X_val, y_val, X_test, y_test = get_CIFAR10_data()
```

1.2 Extract Features

For each image we will compute a Histogram of Oriented Gradients (HOG) as well as a color histogram using the hue channel in HSV color space. We form our final feature vector for each image by concatenating the HOG and color histogram feature vectors.

Roughly speaking, HOG should capture the texture of the image while ignoring color information, and the color histogram represents the color of the input image while ignoring texture. As a result, we expect that using both together ought to work better than using either alone. Verifying this assumption would be a good thing to try for your own interest.

The `hog_feature` and `color_histogram_hsv` functions both operate on a single image and return a feature vector for that image. The extract_features function takes a set of images and a list of feature functions and evaluates each feature function on each image, storing the results in a matrix where each column is the concatenation of all feature vectors for a single image.

[4]:
```
from cs231n.features import *

num_color_bins = 10 # Number of bins in the color histogram
feature_fns = [hog_feature, lambda img: color_histogram_hsv(img,␣
 ↪nbin=num_color_bins)]
X_train_feats = extract_features(X_train, feature_fns, verbose=True)
X_val_feats = extract_features(X_val, feature_fns)
X_test_feats = extract_features(X_test, feature_fns)

# Preprocessing: Subtract the mean feature
mean_feat = np.mean(X_train_feats, axis=0, keepdims=True)
X_train_feats -= mean_feat
X_val_feats -= mean_feat
X_test_feats -= mean_feat

# Preprocessing: Divide by standard deviation. This ensures that each feature
# has roughly the same scale.
std_feat = np.std(X_train_feats, axis=0, keepdims=True)
X_train_feats /= std_feat
X_val_feats /= std_feat
X_test_feats /= std_feat

# Preprocessing: Add a bias dimension
X_train_feats = np.hstack([X_train_feats, np.ones((X_train_feats.shape[0], 1))])
```

```python
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.

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

from cs231n.classifiers.linear_classifier import LinearSVM

learning_rates = [1e-9, 1e-8, 1e-7]
regularization_strengths = [1e5, 1e6, 1e7]

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 learning_rate, reg_strength in zip(learning_rates,␣
↪regularization_strengths):

    svm = LinearSVM()
    svm.train(X_train_feats, y_train, learning_rate=learning_rate,
        reg=reg_strength, num_iters=2000, verbose=False)
    y_val_pred = svm.predict(X_val_feats)
    y_train_pred = svm.predict(X_train_feats)
    valid_accuracy = np.mean(y_val == y_val_pred)
    train_accuracy = np.mean(y_train == y_train_pred)
    results[learning_rate, reg_strength] = (train_accuracy, valid_accuracy)
    if valid_accuracy > best_val:
        best_svm = svm
        best_val = valid_accuracy
    print(learning_rate, reg_strength, valid_accuracy)
```

```
print(results)

# *****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)
```

```
1e-09 100000.0 0.111
1e-08 1000000.0 0.416
1e-07 10000000.0 0.288
{(1e-09, 100000.0): (0.10706122448979592, 0.111), (1e-08, 1000000.0):
(0.414734693877551, 0.416), (1e-07, 10000000.0): (0.28444897959183674, 0.288)}
lr 1.000000e-09 reg 1.000000e+05 train accuracy: 0.107061 val accuracy: 0.111000
lr 1.000000e-08 reg 1.000000e+06 train accuracy: 0.414735 val accuracy: 0.416000
lr 1.000000e-07 reg 1.000000e+07 train accuracy: 0.284449 val accuracy: 0.288000
best validation accuracy achieved: 0.416000
```

[9]:

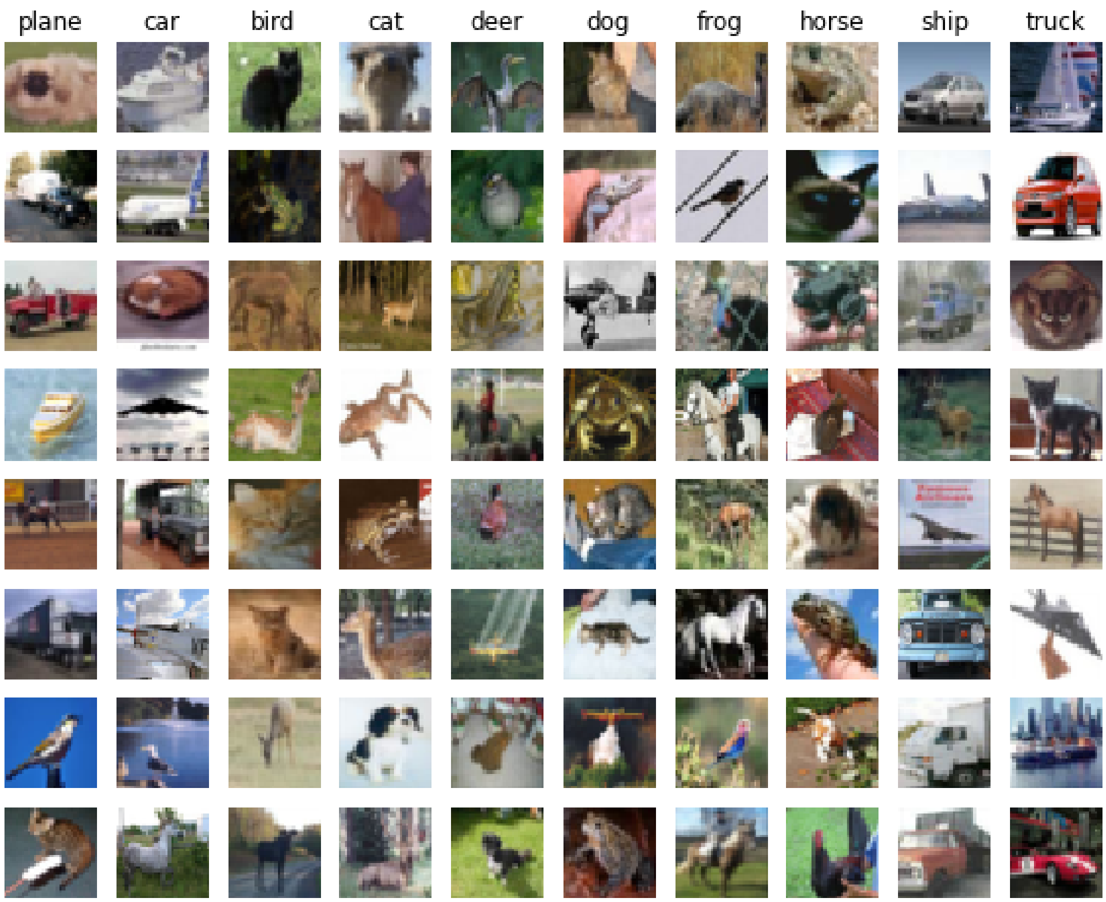
```
# Evaluate your trained SVM on the test set: you should be able to get at least␣
↪0.40
y_test_pred = best_svm.predict(X_test_feats)
test_accuracy = np.mean(y_test == y_test_pred)
print(test_accuracy)
```

```
0.423
```

[10]:

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

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

```
        plt.axis('off')
        if i == 0:
            plt.title(cls_name)
plt.show()
```



1.3.1 Inline question 1:

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

Your Answer :

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.

[11]:
```
# Preprocessing: Remove the bias dimension
# Make sure to run this cell only ONCE
print(X_train_feats.shape)
X_train_feats = X_train_feats[:, :-1]
X_val_feats = X_val_feats[:, :-1]
X_test_feats = X_test_feats[:, :-1]

print(X_train_feats.shape)
```

```
(49000, 155)
(49000, 154)
```

[20]:
```
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,
}

model = TwoLayerNet(input_dim, hidden_dim, num_classes)
best_model = None
learning_rates = 10.0**np.random.uniform(-4,-2,10)
regularization_strengths = 10.0**np.random.uniform(-8,-6,10)

results = {}
best_val = -1

################################################################################
# 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)*****

for learning_rate, reg in zip(learning_rates, regularization_strengths):
```

```python
    model=TwoLayerNet(input_dim, hidden_dim, num_classes,reg=reg)
    solver = Solver(
    model,
    data,
    num_epochs=50,
    print_every=1000,
    batch_size=1000,
    update_rule="adam",
    optim_config={'learning_rate': learning_rate},
    verbose=True
)
    solver.train()
    best_model = model
    y_val_pred = best_model.loss(X_val_feats)
    y_train_pred = best_model.loss(X_train_feats)
    valid_accuracy = np.mean(y_val == y_val_pred)
    train_accuracy = np.mean(y_train == y_train_pred)
    results[learning_rate, reg_strength] = (train_accuracy, valid_accuracy)
    if valid_accuracy > best_val:
        best_model = model
        best_val = valid_accuracy
    print(learning_rate, reg, valid_accuracy)

print(results)

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

```
(Iteration 1 / 2450) loss: 2.302576
(Epoch 0 / 50) train acc: 0.256000; val_acc: 0.259000
(Epoch 1 / 50) train acc: 0.502000; val_acc: 0.510000
(Epoch 2 / 50) train acc: 0.526000; val_acc: 0.519000
(Epoch 3 / 50) train acc: 0.589000; val_acc: 0.547000
(Epoch 4 / 50) train acc: 0.603000; val_acc: 0.556000
(Epoch 5 / 50) train acc: 0.614000; val_acc: 0.567000
(Epoch 6 / 50) train acc: 0.647000; val_acc: 0.587000
(Epoch 7 / 50) train acc: 0.674000; val_acc: 0.594000
(Epoch 8 / 50) train acc: 0.675000; val_acc: 0.584000
(Epoch 9 / 50) train acc: 0.663000; val_acc: 0.579000
(Epoch 10 / 50) train acc: 0.675000; val_acc: 0.601000
(Epoch 11 / 50) train acc: 0.709000; val_acc: 0.608000
(Epoch 12 / 50) train acc: 0.708000; val_acc: 0.603000
(Epoch 13 / 50) train acc: 0.749000; val_acc: 0.599000
(Epoch 14 / 50) train acc: 0.750000; val_acc: 0.593000
(Epoch 15 / 50) train acc: 0.776000; val_acc: 0.595000
(Epoch 16 / 50) train acc: 0.798000; val_acc: 0.602000
(Epoch 17 / 50) train acc: 0.785000; val_acc: 0.606000
(Epoch 18 / 50) train acc: 0.779000; val_acc: 0.581000
```

```
(Epoch 19 / 50) train acc: 0.797000; val_acc: 0.584000
(Epoch 20 / 50) train acc: 0.809000; val_acc: 0.585000
(Iteration 1001 / 2450) loss: 0.544893
(Epoch 21 / 50) train acc: 0.803000; val_acc: 0.594000
(Epoch 22 / 50) train acc: 0.834000; val_acc: 0.591000
(Epoch 23 / 50) train acc: 0.842000; val_acc: 0.592000
(Epoch 24 / 50) train acc: 0.812000; val_acc: 0.579000
(Epoch 25 / 50) train acc: 0.858000; val_acc: 0.576000
(Epoch 26 / 50) train acc: 0.865000; val_acc: 0.590000
(Epoch 27 / 50) train acc: 0.848000; val_acc: 0.584000
(Epoch 28 / 50) train acc: 0.840000; val_acc: 0.585000
(Epoch 29 / 50) train acc: 0.868000; val_acc: 0.591000
(Epoch 30 / 50) train acc: 0.875000; val_acc: 0.586000
(Epoch 31 / 50) train acc: 0.878000; val_acc: 0.580000
(Epoch 32 / 50) train acc: 0.876000; val_acc: 0.584000
(Epoch 33 / 50) train acc: 0.891000; val_acc: 0.569000
(Epoch 34 / 50) train acc: 0.880000; val_acc: 0.585000
(Epoch 35 / 50) train acc: 0.931000; val_acc: 0.572000
(Epoch 36 / 50) train acc: 0.907000; val_acc: 0.574000
(Epoch 37 / 50) train acc: 0.890000; val_acc: 0.572000
(Epoch 38 / 50) train acc: 0.900000; val_acc: 0.568000
(Epoch 39 / 50) train acc: 0.931000; val_acc: 0.585000
(Epoch 40 / 50) train acc: 0.910000; val_acc: 0.576000
(Iteration 2001 / 2450) loss: 0.254766
(Epoch 41 / 50) train acc: 0.928000; val_acc: 0.563000
(Epoch 42 / 50) train acc: 0.919000; val_acc: 0.572000
(Epoch 43 / 50) train acc: 0.924000; val_acc: 0.583000
(Epoch 44 / 50) train acc: 0.934000; val_acc: 0.574000
(Epoch 45 / 50) train acc: 0.942000; val_acc: 0.562000
(Epoch 46 / 50) train acc: 0.932000; val_acc: 0.567000
(Epoch 47 / 50) train acc: 0.950000; val_acc: 0.577000
(Epoch 48 / 50) train acc: 0.945000; val_acc: 0.574000
(Epoch 49 / 50) train acc: 0.949000; val_acc: 0.581000
(Epoch 50 / 50) train acc: 0.961000; val_acc: 0.584000
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:48:
DeprecationWarning: elementwise comparison failed; this will raise an error in
the future.
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:49:
DeprecationWarning: elementwise comparison failed; this will raise an error in
the future.
```

```
0.0014720021561806858 3.432878212427517e-07 0.0
(Iteration 1 / 2450) loss: 2.302577
(Epoch 0 / 50) train acc: 0.201000; val_acc: 0.197000
(Epoch 1 / 50) train acc: 0.495000; val_acc: 0.520000
(Epoch 2 / 50) train acc: 0.535000; val_acc: 0.516000
(Epoch 3 / 50) train acc: 0.558000; val_acc: 0.559000
(Epoch 4 / 50) train acc: 0.575000; val_acc: 0.572000
```

```
(Epoch 5 / 50) train acc: 0.630000; val_acc: 0.566000
(Epoch 6 / 50) train acc: 0.618000; val_acc: 0.585000
(Epoch 7 / 50) train acc: 0.658000; val_acc: 0.593000
(Epoch 8 / 50) train acc: 0.686000; val_acc: 0.601000
(Epoch 9 / 50) train acc: 0.675000; val_acc: 0.598000
(Epoch 10 / 50) train acc: 0.693000; val_acc: 0.595000
(Epoch 11 / 50) train acc: 0.745000; val_acc: 0.598000
(Epoch 12 / 50) train acc: 0.736000; val_acc: 0.610000
(Epoch 13 / 50) train acc: 0.740000; val_acc: 0.595000
(Epoch 14 / 50) train acc: 0.753000; val_acc: 0.603000
(Epoch 15 / 50) train acc: 0.767000; val_acc: 0.609000
(Epoch 16 / 50) train acc: 0.782000; val_acc: 0.606000
(Epoch 17 / 50) train acc: 0.809000; val_acc: 0.605000
(Epoch 18 / 50) train acc: 0.805000; val_acc: 0.610000
(Epoch 19 / 50) train acc: 0.820000; val_acc: 0.597000
(Epoch 20 / 50) train acc: 0.809000; val_acc: 0.576000
(Iteration 1001 / 2450) loss: 0.611028
(Epoch 21 / 50) train acc: 0.833000; val_acc: 0.597000
(Epoch 22 / 50) train acc: 0.845000; val_acc: 0.586000
(Epoch 23 / 50) train acc: 0.827000; val_acc: 0.596000
(Epoch 24 / 50) train acc: 0.856000; val_acc: 0.588000
(Epoch 25 / 50) train acc: 0.858000; val_acc: 0.592000
(Epoch 26 / 50) train acc: 0.852000; val_acc: 0.587000
(Epoch 27 / 50) train acc: 0.883000; val_acc: 0.580000
(Epoch 28 / 50) train acc: 0.883000; val_acc: 0.586000
(Epoch 29 / 50) train acc: 0.878000; val_acc: 0.576000
(Epoch 30 / 50) train acc: 0.908000; val_acc: 0.570000
(Epoch 31 / 50) train acc: 0.888000; val_acc: 0.578000
(Epoch 32 / 50) train acc: 0.882000; val_acc: 0.580000
(Epoch 33 / 50) train acc: 0.903000; val_acc: 0.579000
(Epoch 34 / 50) train acc: 0.904000; val_acc: 0.572000
(Epoch 35 / 50) train acc: 0.920000; val_acc: 0.588000
(Epoch 36 / 50) train acc: 0.906000; val_acc: 0.579000
(Epoch 37 / 50) train acc: 0.918000; val_acc: 0.577000
(Epoch 38 / 50) train acc: 0.904000; val_acc: 0.577000
(Epoch 39 / 50) train acc: 0.916000; val_acc: 0.576000
(Epoch 40 / 50) train acc: 0.932000; val_acc: 0.579000
(Iteration 2001 / 2450) loss: 0.245886
(Epoch 41 / 50) train acc: 0.923000; val_acc: 0.569000
(Epoch 42 / 50) train acc: 0.944000; val_acc: 0.586000
(Epoch 43 / 50) train acc: 0.941000; val_acc: 0.569000
(Epoch 44 / 50) train acc: 0.936000; val_acc: 0.575000
(Epoch 45 / 50) train acc: 0.960000; val_acc: 0.569000
(Epoch 46 / 50) train acc: 0.962000; val_acc: 0.576000
(Epoch 47 / 50) train acc: 0.959000; val_acc: 0.579000
(Epoch 48 / 50) train acc: 0.953000; val_acc: 0.561000
(Epoch 49 / 50) train acc: 0.959000; val_acc: 0.570000
(Epoch 50 / 50) train acc: 0.963000; val_acc: 0.566000
```

```
0.0015503204093147735 4.389546658761831e-08 0.0
(Iteration 1 / 2450) loss: 2.302579
(Epoch 0 / 50) train acc: 0.188000; val_acc: 0.220000
(Epoch 1 / 50) train acc: 0.522000; val_acc: 0.514000
(Epoch 2 / 50) train acc: 0.539000; val_acc: 0.544000
(Epoch 3 / 50) train acc: 0.563000; val_acc: 0.557000
(Epoch 4 / 50) train acc: 0.631000; val_acc: 0.566000
(Epoch 5 / 50) train acc: 0.624000; val_acc: 0.591000
(Epoch 6 / 50) train acc: 0.657000; val_acc: 0.601000
(Epoch 7 / 50) train acc: 0.696000; val_acc: 0.606000
(Epoch 8 / 50) train acc: 0.695000; val_acc: 0.600000
(Epoch 9 / 50) train acc: 0.739000; val_acc: 0.607000
(Epoch 10 / 50) train acc: 0.749000; val_acc: 0.590000
(Epoch 11 / 50) train acc: 0.770000; val_acc: 0.589000
(Epoch 12 / 50) train acc: 0.785000; val_acc: 0.592000
(Epoch 13 / 50) train acc: 0.794000; val_acc: 0.594000
(Epoch 14 / 50) train acc: 0.766000; val_acc: 0.591000
(Epoch 15 / 50) train acc: 0.803000; val_acc: 0.601000
(Epoch 16 / 50) train acc: 0.792000; val_acc: 0.574000
(Epoch 17 / 50) train acc: 0.820000; val_acc: 0.582000
(Epoch 18 / 50) train acc: 0.851000; val_acc: 0.582000
(Epoch 19 / 50) train acc: 0.824000; val_acc: 0.593000
(Epoch 20 / 50) train acc: 0.866000; val_acc: 0.581000
(Iteration 1001 / 2450) loss: 0.428913
(Epoch 21 / 50) train acc: 0.857000; val_acc: 0.586000
(Epoch 22 / 50) train acc: 0.863000; val_acc: 0.580000
(Epoch 23 / 50) train acc: 0.881000; val_acc: 0.576000
(Epoch 24 / 50) train acc: 0.898000; val_acc: 0.577000
(Epoch 25 / 50) train acc: 0.865000; val_acc: 0.571000
(Epoch 26 / 50) train acc: 0.867000; val_acc: 0.564000
(Epoch 27 / 50) train acc: 0.884000; val_acc: 0.583000
(Epoch 28 / 50) train acc: 0.880000; val_acc: 0.567000
(Epoch 29 / 50) train acc: 0.904000; val_acc: 0.576000
(Epoch 30 / 50) train acc: 0.907000; val_acc: 0.577000
(Epoch 31 / 50) train acc: 0.904000; val_acc: 0.568000
(Epoch 32 / 50) train acc: 0.926000; val_acc: 0.568000
(Epoch 33 / 50) train acc: 0.916000; val_acc: 0.568000
(Epoch 34 / 50) train acc: 0.929000; val_acc: 0.576000
(Epoch 35 / 50) train acc: 0.917000; val_acc: 0.571000
(Epoch 36 / 50) train acc: 0.933000; val_acc: 0.570000
(Epoch 37 / 50) train acc: 0.939000; val_acc: 0.565000
(Epoch 38 / 50) train acc: 0.964000; val_acc: 0.572000
(Epoch 39 / 50) train acc: 0.944000; val_acc: 0.561000
(Epoch 40 / 50) train acc: 0.944000; val_acc: 0.564000
(Iteration 2001 / 2450) loss: 0.199601
(Epoch 41 / 50) train acc: 0.950000; val_acc: 0.554000
(Epoch 42 / 50) train acc: 0.943000; val_acc: 0.568000
(Epoch 43 / 50) train acc: 0.952000; val_acc: 0.564000
```

```
(Epoch 44 / 50) train acc: 0.963000; val_acc: 0.559000
(Epoch 45 / 50) train acc: 0.954000; val_acc: 0.565000
(Epoch 46 / 50) train acc: 0.964000; val_acc: 0.554000
(Epoch 47 / 50) train acc: 0.967000; val_acc: 0.563000
(Epoch 48 / 50) train acc: 0.961000; val_acc: 0.558000
(Epoch 49 / 50) train acc: 0.974000; val_acc: 0.563000
(Epoch 50 / 50) train acc: 0.967000; val_acc: 0.546000
0.001996794834469079 5.9451806205728586e-08 0.0
(Iteration 1 / 2450) loss: 2.302595
(Epoch 0 / 50) train acc: 0.109000; val_acc: 0.119000
(Epoch 1 / 50) train acc: 0.574000; val_acc: 0.520000
(Epoch 2 / 50) train acc: 0.581000; val_acc: 0.561000
(Epoch 3 / 50) train acc: 0.644000; val_acc: 0.568000
(Epoch 4 / 50) train acc: 0.687000; val_acc: 0.575000
(Epoch 5 / 50) train acc: 0.703000; val_acc: 0.580000
(Epoch 6 / 50) train acc: 0.739000; val_acc: 0.585000
(Epoch 7 / 50) train acc: 0.744000; val_acc: 0.579000
(Epoch 8 / 50) train acc: 0.759000; val_acc: 0.562000
(Epoch 9 / 50) train acc: 0.802000; val_acc: 0.580000
(Epoch 10 / 50) train acc: 0.797000; val_acc: 0.571000
(Epoch 11 / 50) train acc: 0.811000; val_acc: 0.573000
(Epoch 12 / 50) train acc: 0.816000; val_acc: 0.578000
(Epoch 13 / 50) train acc: 0.826000; val_acc: 0.571000
(Epoch 14 / 50) train acc: 0.850000; val_acc: 0.555000
(Epoch 15 / 50) train acc: 0.845000; val_acc: 0.557000
(Epoch 16 / 50) train acc: 0.867000; val_acc: 0.557000
(Epoch 17 / 50) train acc: 0.881000; val_acc: 0.571000
(Epoch 18 / 50) train acc: 0.893000; val_acc: 0.546000
(Epoch 19 / 50) train acc: 0.902000; val_acc: 0.555000
(Epoch 20 / 50) train acc: 0.879000; val_acc: 0.542000
(Iteration 1001 / 2450) loss: 0.296358
(Epoch 21 / 50) train acc: 0.894000; val_acc: 0.548000
(Epoch 22 / 50) train acc: 0.895000; val_acc: 0.541000
(Epoch 23 / 50) train acc: 0.909000; val_acc: 0.561000
(Epoch 24 / 50) train acc: 0.932000; val_acc: 0.543000
(Epoch 25 / 50) train acc: 0.936000; val_acc: 0.543000
(Epoch 26 / 50) train acc: 0.920000; val_acc: 0.547000
(Epoch 27 / 50) train acc: 0.933000; val_acc: 0.546000
(Epoch 28 / 50) train acc: 0.938000; val_acc: 0.543000
(Epoch 29 / 50) train acc: 0.945000; val_acc: 0.542000
(Epoch 30 / 50) train acc: 0.948000; val_acc: 0.534000
(Epoch 31 / 50) train acc: 0.946000; val_acc: 0.554000
(Epoch 32 / 50) train acc: 0.947000; val_acc: 0.543000
(Epoch 33 / 50) train acc: 0.949000; val_acc: 0.545000
(Epoch 34 / 50) train acc: 0.964000; val_acc: 0.542000
(Epoch 35 / 50) train acc: 0.960000; val_acc: 0.543000
(Epoch 36 / 50) train acc: 0.964000; val_acc: 0.547000
(Epoch 37 / 50) train acc: 0.960000; val_acc: 0.539000
```

```
(Epoch 38 / 50) train acc: 0.965000; val_acc: 0.543000
(Epoch 39 / 50) train acc: 0.964000; val_acc: 0.551000
(Epoch 40 / 50) train acc: 0.955000; val_acc: 0.545000
(Iteration 2001 / 2450) loss: 0.101200
(Epoch 41 / 50) train acc: 0.967000; val_acc: 0.536000
(Epoch 42 / 50) train acc: 0.974000; val_acc: 0.537000
(Epoch 43 / 50) train acc: 0.975000; val_acc: 0.539000
(Epoch 44 / 50) train acc: 0.983000; val_acc: 0.542000
(Epoch 45 / 50) train acc: 0.982000; val_acc: 0.547000
(Epoch 46 / 50) train acc: 0.980000; val_acc: 0.542000
(Epoch 47 / 50) train acc: 0.972000; val_acc: 0.532000
(Epoch 48 / 50) train acc: 0.967000; val_acc: 0.522000
(Epoch 49 / 50) train acc: 0.983000; val_acc: 0.537000
(Epoch 50 / 50) train acc: 0.974000; val_acc: 0.541000
0.00486067968787039 5.4708193713753016e-08 0.0
(Iteration 1 / 2450) loss: 2.302562
(Epoch 0 / 50) train acc: 0.226000; val_acc: 0.237000
(Epoch 1 / 50) train acc: 0.337000; val_acc: 0.331000
(Epoch 2 / 50) train acc: 0.370000; val_acc: 0.369000
(Epoch 3 / 50) train acc: 0.451000; val_acc: 0.431000
(Epoch 4 / 50) train acc: 0.497000; val_acc: 0.466000
(Epoch 5 / 50) train acc: 0.486000; val_acc: 0.493000
(Epoch 6 / 50) train acc: 0.500000; val_acc: 0.500000
(Epoch 7 / 50) train acc: 0.505000; val_acc: 0.509000
(Epoch 8 / 50) train acc: 0.537000; val_acc: 0.512000
(Epoch 9 / 50) train acc: 0.504000; val_acc: 0.515000
(Epoch 10 / 50) train acc: 0.525000; val_acc: 0.514000
(Epoch 11 / 50) train acc: 0.562000; val_acc: 0.522000
(Epoch 12 / 50) train acc: 0.552000; val_acc: 0.521000
(Epoch 13 / 50) train acc: 0.555000; val_acc: 0.532000
(Epoch 14 / 50) train acc: 0.549000; val_acc: 0.539000
(Epoch 15 / 50) train acc: 0.527000; val_acc: 0.539000
(Epoch 16 / 50) train acc: 0.576000; val_acc: 0.539000
(Epoch 17 / 50) train acc: 0.564000; val_acc: 0.555000
(Epoch 18 / 50) train acc: 0.576000; val_acc: 0.556000
(Epoch 19 / 50) train acc: 0.563000; val_acc: 0.558000
(Epoch 20 / 50) train acc: 0.601000; val_acc: 0.555000
(Iteration 1001 / 2450) loss: 1.200875
(Epoch 21 / 50) train acc: 0.617000; val_acc: 0.551000
(Epoch 22 / 50) train acc: 0.578000; val_acc: 0.551000
(Epoch 23 / 50) train acc: 0.592000; val_acc: 0.556000
(Epoch 24 / 50) train acc: 0.590000; val_acc: 0.565000
(Epoch 25 / 50) train acc: 0.585000; val_acc: 0.572000
(Epoch 26 / 50) train acc: 0.602000; val_acc: 0.568000
(Epoch 27 / 50) train acc: 0.600000; val_acc: 0.566000
(Epoch 28 / 50) train acc: 0.592000; val_acc: 0.574000
(Epoch 29 / 50) train acc: 0.606000; val_acc: 0.569000
(Epoch 30 / 50) train acc: 0.628000; val_acc: 0.574000
```

```
(Epoch 31 / 50) train acc: 0.617000; val_acc: 0.583000
(Epoch 32 / 50) train acc: 0.611000; val_acc: 0.588000
(Epoch 33 / 50) train acc: 0.614000; val_acc: 0.583000
(Epoch 34 / 50) train acc: 0.603000; val_acc: 0.592000
(Epoch 35 / 50) train acc: 0.617000; val_acc: 0.586000
(Epoch 36 / 50) train acc: 0.648000; val_acc: 0.589000
(Epoch 37 / 50) train acc: 0.622000; val_acc: 0.589000
(Epoch 38 / 50) train acc: 0.636000; val_acc: 0.593000
(Epoch 39 / 50) train acc: 0.654000; val_acc: 0.595000
(Epoch 40 / 50) train acc: 0.663000; val_acc: 0.595000
(Iteration 2001 / 2450) loss: 0.970920
(Epoch 41 / 50) train acc: 0.663000; val_acc: 0.595000
(Epoch 42 / 50) train acc: 0.647000; val_acc: 0.595000
(Epoch 43 / 50) train acc: 0.672000; val_acc: 0.600000
(Epoch 44 / 50) train acc: 0.635000; val_acc: 0.604000
(Epoch 45 / 50) train acc: 0.648000; val_acc: 0.593000
(Epoch 46 / 50) train acc: 0.650000; val_acc: 0.594000
(Epoch 47 / 50) train acc: 0.648000; val_acc: 0.599000
(Epoch 48 / 50) train acc: 0.666000; val_acc: 0.601000
(Epoch 49 / 50) train acc: 0.701000; val_acc: 0.599000
(Epoch 50 / 50) train acc: 0.668000; val_acc: 0.600000
0.00019220676705850566 1.6636217298001517e-07 0.0
(Iteration 1 / 2450) loss: 2.302562
(Epoch 0 / 50) train acc: 0.223000; val_acc: 0.214000
(Epoch 1 / 50) train acc: 0.503000; val_acc: 0.514000
(Epoch 2 / 50) train acc: 0.540000; val_acc: 0.539000
(Epoch 3 / 50) train acc: 0.590000; val_acc: 0.561000
(Epoch 4 / 50) train acc: 0.635000; val_acc: 0.577000
(Epoch 5 / 50) train acc: 0.647000; val_acc: 0.595000
(Epoch 6 / 50) train acc: 0.677000; val_acc: 0.587000
(Epoch 7 / 50) train acc: 0.696000; val_acc: 0.604000
(Epoch 8 / 50) train acc: 0.719000; val_acc: 0.589000
(Epoch 9 / 50) train acc: 0.731000; val_acc: 0.591000
(Epoch 10 / 50) train acc: 0.765000; val_acc: 0.587000
(Epoch 11 / 50) train acc: 0.766000; val_acc: 0.578000
(Epoch 12 / 50) train acc: 0.786000; val_acc: 0.587000
(Epoch 13 / 50) train acc: 0.776000; val_acc: 0.573000
(Epoch 14 / 50) train acc: 0.775000; val_acc: 0.584000
(Epoch 15 / 50) train acc: 0.798000; val_acc: 0.588000
(Epoch 16 / 50) train acc: 0.822000; val_acc: 0.581000
(Epoch 17 / 50) train acc: 0.825000; val_acc: 0.578000
(Epoch 18 / 50) train acc: 0.827000; val_acc: 0.574000
(Epoch 19 / 50) train acc: 0.865000; val_acc: 0.580000
(Epoch 20 / 50) train acc: 0.868000; val_acc: 0.568000
(Iteration 1001 / 2450) loss: 0.434368
(Epoch 21 / 50) train acc: 0.879000; val_acc: 0.574000
(Epoch 22 / 50) train acc: 0.857000; val_acc: 0.581000
(Epoch 23 / 50) train acc: 0.874000; val_acc: 0.576000
```

```
(Epoch 24 / 50) train acc: 0.889000; val_acc: 0.564000
(Epoch 25 / 50) train acc: 0.903000; val_acc: 0.566000
(Epoch 26 / 50) train acc: 0.882000; val_acc: 0.553000
(Epoch 27 / 50) train acc: 0.924000; val_acc: 0.563000
(Epoch 28 / 50) train acc: 0.900000; val_acc: 0.563000
(Epoch 29 / 50) train acc: 0.913000; val_acc: 0.559000
(Epoch 30 / 50) train acc: 0.916000; val_acc: 0.565000
(Epoch 31 / 50) train acc: 0.913000; val_acc: 0.556000
(Epoch 32 / 50) train acc: 0.934000; val_acc: 0.557000
(Epoch 33 / 50) train acc: 0.935000; val_acc: 0.556000
(Epoch 34 / 50) train acc: 0.933000; val_acc: 0.562000
(Epoch 35 / 50) train acc: 0.948000; val_acc: 0.559000
(Epoch 36 / 50) train acc: 0.951000; val_acc: 0.545000
(Epoch 37 / 50) train acc: 0.945000; val_acc: 0.547000
(Epoch 38 / 50) train acc: 0.964000; val_acc: 0.555000
(Epoch 39 / 50) train acc: 0.964000; val_acc: 0.566000
(Epoch 40 / 50) train acc: 0.971000; val_acc: 0.548000
(Iteration 2001 / 2450) loss: 0.153704
(Epoch 41 / 50) train acc: 0.971000; val_acc: 0.556000
(Epoch 42 / 50) train acc: 0.978000; val_acc: 0.552000
(Epoch 43 / 50) train acc: 0.968000; val_acc: 0.543000
(Epoch 44 / 50) train acc: 0.971000; val_acc: 0.554000
(Epoch 45 / 50) train acc: 0.966000; val_acc: 0.543000
(Epoch 46 / 50) train acc: 0.976000; val_acc: 0.549000
(Epoch 47 / 50) train acc: 0.977000; val_acc: 0.548000
(Epoch 48 / 50) train acc: 0.981000; val_acc: 0.551000
(Epoch 49 / 50) train acc: 0.972000; val_acc: 0.551000
(Epoch 50 / 50) train acc: 0.989000; val_acc: 0.551000
0.0024615122506708004 2.1007983611506767e-08 0.0
(Iteration 1 / 2450) loss: 2.302570
(Epoch 0 / 50) train acc: 0.170000; val_acc: 0.182000
(Epoch 1 / 50) train acc: 0.539000; val_acc: 0.529000
(Epoch 2 / 50) train acc: 0.614000; val_acc: 0.559000
(Epoch 3 / 50) train acc: 0.625000; val_acc: 0.580000
(Epoch 4 / 50) train acc: 0.684000; val_acc: 0.586000
(Epoch 5 / 50) train acc: 0.705000; val_acc: 0.588000
(Epoch 6 / 50) train acc: 0.702000; val_acc: 0.588000
(Epoch 7 / 50) train acc: 0.742000; val_acc: 0.580000
(Epoch 8 / 50) train acc: 0.759000; val_acc: 0.583000
(Epoch 9 / 50) train acc: 0.787000; val_acc: 0.586000
(Epoch 10 / 50) train acc: 0.799000; val_acc: 0.582000
(Epoch 11 / 50) train acc: 0.787000; val_acc: 0.574000
(Epoch 12 / 50) train acc: 0.793000; val_acc: 0.582000
(Epoch 13 / 50) train acc: 0.822000; val_acc: 0.579000
(Epoch 14 / 50) train acc: 0.853000; val_acc: 0.581000
(Epoch 15 / 50) train acc: 0.860000; val_acc: 0.557000
(Epoch 16 / 50) train acc: 0.854000; val_acc: 0.561000
(Epoch 17 / 50) train acc: 0.871000; val_acc: 0.564000
```

```
(Epoch 18 / 50) train acc: 0.879000; val_acc: 0.559000
(Epoch 19 / 50) train acc: 0.883000; val_acc: 0.569000
(Epoch 20 / 50) train acc: 0.881000; val_acc: 0.570000
(Iteration 1001 / 2450) loss: 0.344897
(Epoch 21 / 50) train acc: 0.912000; val_acc: 0.562000
(Epoch 22 / 50) train acc: 0.906000; val_acc: 0.560000
(Epoch 23 / 50) train acc: 0.899000; val_acc: 0.555000
(Epoch 24 / 50) train acc: 0.930000; val_acc: 0.550000
(Epoch 25 / 50) train acc: 0.922000; val_acc: 0.564000
(Epoch 26 / 50) train acc: 0.939000; val_acc: 0.562000
(Epoch 27 / 50) train acc: 0.938000; val_acc: 0.546000
(Epoch 28 / 50) train acc: 0.943000; val_acc: 0.556000
(Epoch 29 / 50) train acc: 0.951000; val_acc: 0.558000
(Epoch 30 / 50) train acc: 0.942000; val_acc: 0.551000
(Epoch 31 / 50) train acc: 0.939000; val_acc: 0.550000
(Epoch 32 / 50) train acc: 0.947000; val_acc: 0.558000
(Epoch 33 / 50) train acc: 0.960000; val_acc: 0.565000
(Epoch 34 / 50) train acc: 0.964000; val_acc: 0.551000
(Epoch 35 / 50) train acc: 0.960000; val_acc: 0.555000
(Epoch 36 / 50) train acc: 0.961000; val_acc: 0.548000
(Epoch 37 / 50) train acc: 0.969000; val_acc: 0.560000
(Epoch 38 / 50) train acc: 0.976000; val_acc: 0.547000
(Epoch 39 / 50) train acc: 0.971000; val_acc: 0.562000
(Epoch 40 / 50) train acc: 0.980000; val_acc: 0.561000
(Iteration 2001 / 2450) loss: 0.117508
(Epoch 41 / 50) train acc: 0.981000; val_acc: 0.548000
(Epoch 42 / 50) train acc: 0.977000; val_acc: 0.558000
(Epoch 43 / 50) train acc: 0.972000; val_acc: 0.545000
(Epoch 44 / 50) train acc: 0.978000; val_acc: 0.546000
(Epoch 45 / 50) train acc: 0.980000; val_acc: 0.542000
(Epoch 46 / 50) train acc: 0.988000; val_acc: 0.542000
(Epoch 47 / 50) train acc: 0.986000; val_acc: 0.549000
(Epoch 48 / 50) train acc: 0.986000; val_acc: 0.543000
(Epoch 49 / 50) train acc: 0.986000; val_acc: 0.537000
(Epoch 50 / 50) train acc: 0.990000; val_acc: 0.554000
0.0044870137329137885 1.4520914137030708e-08 0.0
(Iteration 1 / 2450) loss: 2.302586
(Epoch 0 / 50) train acc: 0.276000; val_acc: 0.278000
(Epoch 1 / 50) train acc: 0.325000; val_acc: 0.314000
(Epoch 2 / 50) train acc: 0.356000; val_acc: 0.346000
(Epoch 3 / 50) train acc: 0.427000; val_acc: 0.392000
(Epoch 4 / 50) train acc: 0.422000; val_acc: 0.436000
(Epoch 5 / 50) train acc: 0.439000; val_acc: 0.465000
(Epoch 6 / 50) train acc: 0.516000; val_acc: 0.498000
(Epoch 7 / 50) train acc: 0.513000; val_acc: 0.498000
(Epoch 8 / 50) train acc: 0.507000; val_acc: 0.500000
(Epoch 9 / 50) train acc: 0.496000; val_acc: 0.519000
(Epoch 10 / 50) train acc: 0.536000; val_acc: 0.520000
```

```
(Epoch 11 / 50) train acc: 0.524000; val_acc: 0.521000
(Epoch 12 / 50) train acc: 0.551000; val_acc: 0.517000
(Epoch 13 / 50) train acc: 0.540000; val_acc: 0.520000
(Epoch 14 / 50) train acc: 0.534000; val_acc: 0.523000
(Epoch 15 / 50) train acc: 0.550000; val_acc: 0.525000
(Epoch 16 / 50) train acc: 0.558000; val_acc: 0.523000
(Epoch 17 / 50) train acc: 0.565000; val_acc: 0.537000
(Epoch 18 / 50) train acc: 0.550000; val_acc: 0.531000
(Epoch 19 / 50) train acc: 0.569000; val_acc: 0.531000
(Epoch 20 / 50) train acc: 0.572000; val_acc: 0.535000
(Iteration 1001 / 2450) loss: 1.248566
(Epoch 21 / 50) train acc: 0.568000; val_acc: 0.543000
(Epoch 22 / 50) train acc: 0.594000; val_acc: 0.549000
(Epoch 23 / 50) train acc: 0.566000; val_acc: 0.557000
(Epoch 24 / 50) train acc: 0.593000; val_acc: 0.551000
(Epoch 25 / 50) train acc: 0.586000; val_acc: 0.557000
(Epoch 26 / 50) train acc: 0.539000; val_acc: 0.564000
(Epoch 27 / 50) train acc: 0.591000; val_acc: 0.563000
(Epoch 28 / 50) train acc: 0.599000; val_acc: 0.564000
(Epoch 29 / 50) train acc: 0.589000; val_acc: 0.559000
(Epoch 30 / 50) train acc: 0.606000; val_acc: 0.566000
(Epoch 31 / 50) train acc: 0.607000; val_acc: 0.558000
(Epoch 32 / 50) train acc: 0.586000; val_acc: 0.568000
(Epoch 33 / 50) train acc: 0.613000; val_acc: 0.570000
(Epoch 34 / 50) train acc: 0.592000; val_acc: 0.580000
(Epoch 35 / 50) train acc: 0.613000; val_acc: 0.580000
(Epoch 36 / 50) train acc: 0.591000; val_acc: 0.576000
(Epoch 37 / 50) train acc: 0.582000; val_acc: 0.576000
(Epoch 38 / 50) train acc: 0.610000; val_acc: 0.566000
(Epoch 39 / 50) train acc: 0.625000; val_acc: 0.581000
(Epoch 40 / 50) train acc: 0.629000; val_acc: 0.584000
(Iteration 2001 / 2450) loss: 1.151488
(Epoch 41 / 50) train acc: 0.638000; val_acc: 0.576000
(Epoch 42 / 50) train acc: 0.644000; val_acc: 0.577000
(Epoch 43 / 50) train acc: 0.610000; val_acc: 0.578000
(Epoch 44 / 50) train acc: 0.628000; val_acc: 0.582000
(Epoch 45 / 50) train acc: 0.621000; val_acc: 0.576000
(Epoch 46 / 50) train acc: 0.657000; val_acc: 0.584000
(Epoch 47 / 50) train acc: 0.638000; val_acc: 0.587000
(Epoch 48 / 50) train acc: 0.654000; val_acc: 0.596000
(Epoch 49 / 50) train acc: 0.640000; val_acc: 0.592000
(Epoch 50 / 50) train acc: 0.643000; val_acc: 0.592000
0.00014827646636610917 1.456627748875489e-08 0.0
(Iteration 1 / 2450) loss: 2.302589
(Epoch 0 / 50) train acc: 0.289000; val_acc: 0.300000
(Epoch 1 / 50) train acc: 0.505000; val_acc: 0.513000
(Epoch 2 / 50) train acc: 0.556000; val_acc: 0.535000
(Epoch 3 / 50) train acc: 0.607000; val_acc: 0.543000
```

```
(Epoch 4 / 50) train acc: 0.623000; val_acc: 0.560000
(Epoch 5 / 50) train acc: 0.633000; val_acc: 0.570000
(Epoch 6 / 50) train acc: 0.663000; val_acc: 0.590000
(Epoch 7 / 50) train acc: 0.680000; val_acc: 0.598000
(Epoch 8 / 50) train acc: 0.715000; val_acc: 0.581000
(Epoch 9 / 50) train acc: 0.669000; val_acc: 0.581000
(Epoch 10 / 50) train acc: 0.723000; val_acc: 0.608000
(Epoch 11 / 50) train acc: 0.736000; val_acc: 0.604000
(Epoch 12 / 50) train acc: 0.725000; val_acc: 0.612000
(Epoch 13 / 50) train acc: 0.761000; val_acc: 0.596000
(Epoch 14 / 50) train acc: 0.792000; val_acc: 0.594000
(Epoch 15 / 50) train acc: 0.766000; val_acc: 0.594000
(Epoch 16 / 50) train acc: 0.795000; val_acc: 0.589000
(Epoch 17 / 50) train acc: 0.783000; val_acc: 0.601000
(Epoch 18 / 50) train acc: 0.788000; val_acc: 0.581000
(Epoch 19 / 50) train acc: 0.837000; val_acc: 0.598000
(Epoch 20 / 50) train acc: 0.835000; val_acc: 0.579000
(Iteration 1001 / 2450) loss: 0.544925
(Epoch 21 / 50) train acc: 0.823000; val_acc: 0.589000
(Epoch 22 / 50) train acc: 0.861000; val_acc: 0.596000
(Epoch 23 / 50) train acc: 0.864000; val_acc: 0.572000
(Epoch 24 / 50) train acc: 0.848000; val_acc: 0.595000
(Epoch 25 / 50) train acc: 0.855000; val_acc: 0.595000
(Epoch 26 / 50) train acc: 0.866000; val_acc: 0.583000
(Epoch 27 / 50) train acc: 0.882000; val_acc: 0.588000
(Epoch 28 / 50) train acc: 0.881000; val_acc: 0.590000
(Epoch 29 / 50) train acc: 0.889000; val_acc: 0.591000
(Epoch 30 / 50) train acc: 0.894000; val_acc: 0.593000
(Epoch 31 / 50) train acc: 0.889000; val_acc: 0.585000
(Epoch 32 / 50) train acc: 0.892000; val_acc: 0.591000
(Epoch 33 / 50) train acc: 0.925000; val_acc: 0.583000
(Epoch 34 / 50) train acc: 0.920000; val_acc: 0.590000
(Epoch 35 / 50) train acc: 0.927000; val_acc: 0.584000
(Epoch 36 / 50) train acc: 0.913000; val_acc: 0.575000
(Epoch 37 / 50) train acc: 0.923000; val_acc: 0.576000
(Epoch 38 / 50) train acc: 0.934000; val_acc: 0.579000
(Epoch 39 / 50) train acc: 0.944000; val_acc: 0.579000
(Epoch 40 / 50) train acc: 0.928000; val_acc: 0.577000
(Iteration 2001 / 2450) loss: 0.233693
(Epoch 41 / 50) train acc: 0.935000; val_acc: 0.580000
(Epoch 42 / 50) train acc: 0.944000; val_acc: 0.572000
(Epoch 43 / 50) train acc: 0.957000; val_acc: 0.575000
(Epoch 44 / 50) train acc: 0.947000; val_acc: 0.572000
(Epoch 45 / 50) train acc: 0.963000; val_acc: 0.564000
(Epoch 46 / 50) train acc: 0.952000; val_acc: 0.571000
(Epoch 47 / 50) train acc: 0.976000; val_acc: 0.584000
(Epoch 48 / 50) train acc: 0.957000; val_acc: 0.565000
(Epoch 49 / 50) train acc: 0.950000; val_acc: 0.571000
```

```
(Epoch 50 / 50) train acc: 0.960000; val_acc: 0.571000
0.0017524050723277805 4.8397326736490864e-08 0.0
(Iteration 1 / 2450) loss: 2.302571
(Epoch 0 / 50) train acc: 0.152000; val_acc: 0.176000
(Epoch 1 / 50) train acc: 0.554000; val_acc: 0.534000
(Epoch 2 / 50) train acc: 0.624000; val_acc: 0.558000
(Epoch 3 / 50) train acc: 0.640000; val_acc: 0.573000
(Epoch 4 / 50) train acc: 0.694000; val_acc: 0.589000
(Epoch 5 / 50) train acc: 0.705000; val_acc: 0.575000
(Epoch 6 / 50) train acc: 0.739000; val_acc: 0.592000
(Epoch 7 / 50) train acc: 0.775000; val_acc: 0.600000
(Epoch 8 / 50) train acc: 0.795000; val_acc: 0.590000
(Epoch 9 / 50) train acc: 0.786000; val_acc: 0.570000
(Epoch 10 / 50) train acc: 0.808000; val_acc: 0.578000
(Epoch 11 / 50) train acc: 0.807000; val_acc: 0.570000
(Epoch 12 / 50) train acc: 0.823000; val_acc: 0.577000
(Epoch 13 / 50) train acc: 0.830000; val_acc: 0.568000
(Epoch 14 / 50) train acc: 0.852000; val_acc: 0.567000
(Epoch 15 / 50) train acc: 0.868000; val_acc: 0.569000
(Epoch 16 / 50) train acc: 0.871000; val_acc: 0.578000
(Epoch 17 / 50) train acc: 0.875000; val_acc: 0.557000
(Epoch 18 / 50) train acc: 0.869000; val_acc: 0.557000
(Epoch 19 / 50) train acc: 0.883000; val_acc: 0.566000
(Epoch 20 / 50) train acc: 0.907000; val_acc: 0.562000
(Iteration 1001 / 2450) loss: 0.263547
(Epoch 21 / 50) train acc: 0.905000; val_acc: 0.564000
(Epoch 22 / 50) train acc: 0.906000; val_acc: 0.571000
(Epoch 23 / 50) train acc: 0.928000; val_acc: 0.554000
(Epoch 24 / 50) train acc: 0.918000; val_acc: 0.540000
(Epoch 25 / 50) train acc: 0.922000; val_acc: 0.550000
(Epoch 26 / 50) train acc: 0.924000; val_acc: 0.570000
(Epoch 27 / 50) train acc: 0.924000; val_acc: 0.566000
(Epoch 28 / 50) train acc: 0.946000; val_acc: 0.553000
(Epoch 29 / 50) train acc: 0.941000; val_acc: 0.537000
(Epoch 30 / 50) train acc: 0.936000; val_acc: 0.539000
(Epoch 31 / 50) train acc: 0.936000; val_acc: 0.554000
(Epoch 32 / 50) train acc: 0.950000; val_acc: 0.551000
(Epoch 33 / 50) train acc: 0.960000; val_acc: 0.553000
(Epoch 34 / 50) train acc: 0.962000; val_acc: 0.547000
(Epoch 35 / 50) train acc: 0.953000; val_acc: 0.567000
(Epoch 36 / 50) train acc: 0.965000; val_acc: 0.562000
(Epoch 37 / 50) train acc: 0.963000; val_acc: 0.546000
(Epoch 38 / 50) train acc: 0.957000; val_acc: 0.555000
(Epoch 39 / 50) train acc: 0.953000; val_acc: 0.559000
(Epoch 40 / 50) train acc: 0.960000; val_acc: 0.552000
(Iteration 2001 / 2450) loss: 0.126702
(Epoch 41 / 50) train acc: 0.976000; val_acc: 0.545000
(Epoch 42 / 50) train acc: 0.962000; val_acc: 0.561000
```

```
(Epoch 43 / 50) train acc: 0.966000; val_acc: 0.542000
(Epoch 44 / 50) train acc: 0.970000; val_acc: 0.554000
(Epoch 45 / 50) train acc: 0.975000; val_acc: 0.548000
(Epoch 46 / 50) train acc: 0.953000; val_acc: 0.545000
(Epoch 47 / 50) train acc: 0.958000; val_acc: 0.554000
(Epoch 48 / 50) train acc: 0.958000; val_acc: 0.563000
(Epoch 49 / 50) train acc: 0.963000; val_acc: 0.553000
(Epoch 50 / 50) train acc: 0.964000; val_acc: 0.560000
0.005993473246598649 1.544336996264558e-07 0.0
{(0.0014720021561806858, 10000000.0): (0.0, 0.0), (0.0015503204093147735,
10000000.0): (0.0, 0.0), (0.001996794834469079, 10000000.0): (0.0, 0.0),
(0.00486067968787039, 10000000.0): (0.0, 0.0), (0.00019220676705850566,
10000000.0): (0.0, 0.0), (0.0024615122506708004, 10000000.0): (0.0, 0.0),
(0.0044870137329137885, 10000000.0): (0.0, 0.0), (0.00014827646636610917,
10000000.0): (0.0, 0.0), (0.0017524050723277805, 10000000.0): (0.0, 0.0),
(0.005993473246598649, 10000000.0): (0.0, 0.0)}
```

[21]:

```
# 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_model.loss(data['X_test']), axis=1)
test_acc = (y_test_pred == data['y_test']).mean()
print(test_acc)
```

```
0.58
```