knn

June 29, 2023

[50]:
```
# 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, "cs231n/assignments/assignment1/"

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

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

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

[51]:
```
# 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
```

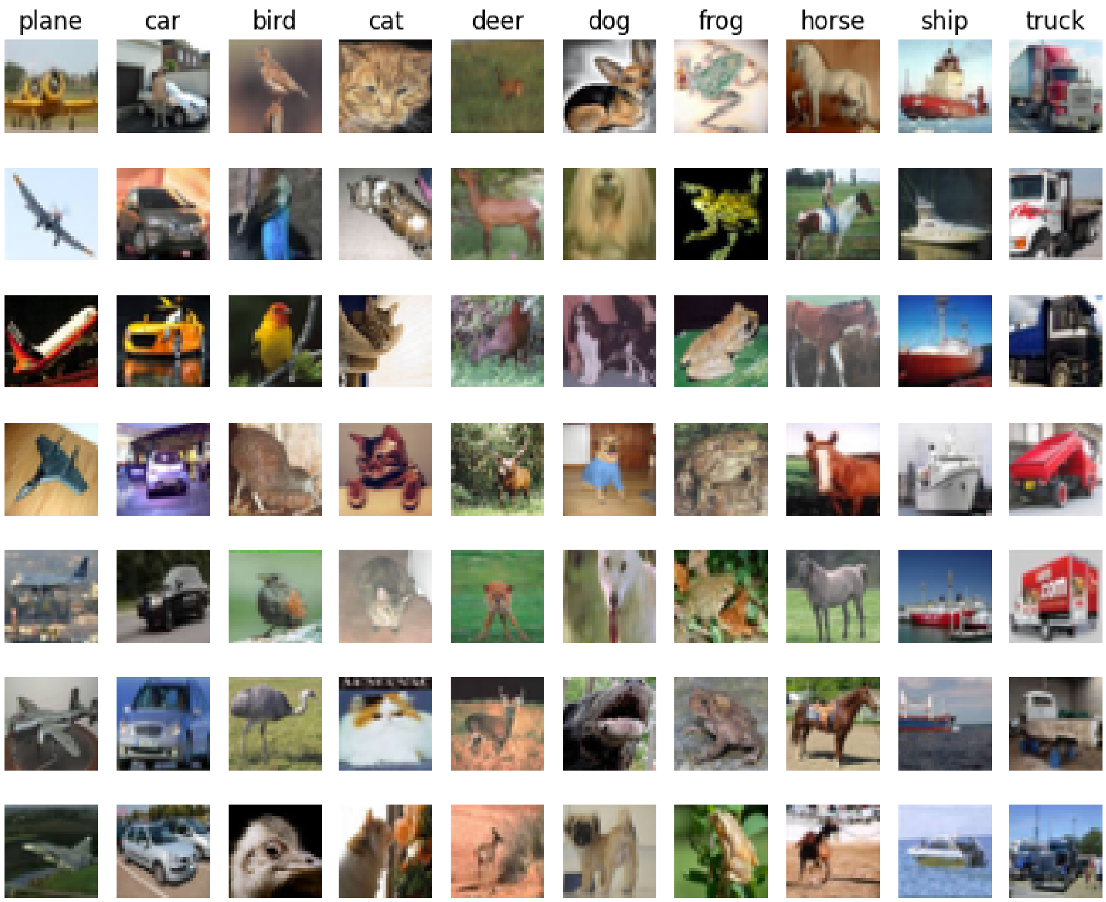
[52]:
```
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

# 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,)
```

[53]:

```
# 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()
```



[54]:

```
# 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)
```

[55]:
```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.

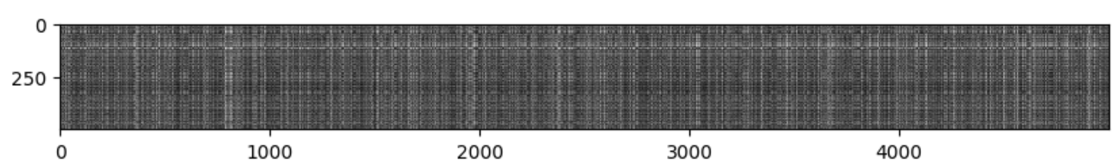
Note: For the three distance computations that we require you to implement in this notebook, you may not use the np.linalg.norm() function that numpy provides.

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.

[56]:
```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)
```

[57]:
```python
# We can visualize the distance matrix: each row is a single test example and
# its distances to training examples
plt.imshow(dists, interpolation='none')
plt.show()
```



Inline Question 1

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

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

YourAnswer : As the brightness of $p_{i,j}$ indicates the distance between i^{th} test image and j^{th} training image, a particularly different image from the mean, say a very dark or a very bright one, will create a bright output line. Such test samples would create bright rows, whereas training samples would create bright columns.

[58]:
```
# 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`:

[59]:
```
y_test_pred = classifier.predict_labels(dists, k=5)
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
```

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

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

Inline Question 2

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

the mean μ across all pixels over all images is

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

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

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

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

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

1. Subtracting the mean μ ($\tilde{p}_{ij}^{(k)} = p_{ij}^{(k)} - \mu$.)
2. Subtracting the per pixel mean μ_{ij} ($\tilde{p}_{ij}^{(k)} = p_{ij}^{(k)} - \mu_{ij}$.)
3. Subtracting the mean μ and dividing by the standard deviation σ.
4. Subtracting the pixel-wise mean μ_{ij} and dividing by the pixel-wise standard deviation σ_{ij}.
5. Rotating the coordinate axes of the data, which means rotating all the images by the same angle. Empty regions in the image caused by rotation are padded with a same pixel value and no interpolation is performed.

$\color{blue}{\textit{Your Answer :}}$ 1,2,3,5

$\color{blue}{\textit{Your Explanation :}}$ 1) Subtracting the mean won't effect the L1 distance between two images, as the same value is subtracted from the both sides of the equation, thus keeping the relative distances unchanged. 2) Subtracting the per pixel mean doesn't effect the L1 distance as well, because only pixels which have the same coordinates are compared. The mean pixel value subtracted from both of which would be the same, as they have identical coordinates i and j, thus keeping the distance unchanged. 3) Scaling with σ would keep the same KNN performance even though it may change the L1 distances. This is due to the fact that the scaling wouldn't change the order of distances between images and only change the magnitudes, as long as all dimensions are scaled with the same constant. Therefore preventing any dimension to become more or less pronounced.

4) Scaling with σ's that depend on the dimension would change the performance as it would cause some dimensions to be more pronounced, increasing the effect they have on the final L1 distance by disproportionately scaling down the other dimensions.
5) Rotation would keep the L1 distances unchanged, as any pair of pixels to be compared stays the same. Meaning $p_{ij}^{(k_1)}$ becomes $p_{R(i),R(j)}^{(k_1)}$ while its counterpart $p_{ij}^{(k_2)}$ becomes $p_{R(i),R(j)}^{(k_2)}$ and the L1 distance stays the same.

$$p_{ij}^{(k_1)} - p_{ij}^{(k_2)} = p_{R(i)R(j)}^{(k_1)} - p_{R(i)R(j)}^{(k_2)}$$

[60]:
```
# 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
```

[61]:

```python
# 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
```

[62]:

```python
# 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 41.253127 seconds
One loop version took 57.323945 seconds
No loop version took 0.534587 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.

[65]:
```
num_folds = 5
k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50, 100]

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

pass

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

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


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

    try:
      X_train_set = np.concatenate((X_train_folds[:i],X_train_folds[i+1:]))
      y_train_set = np.concatenate((y_train_folds[:i],y_train_folds[i+1:]))
```

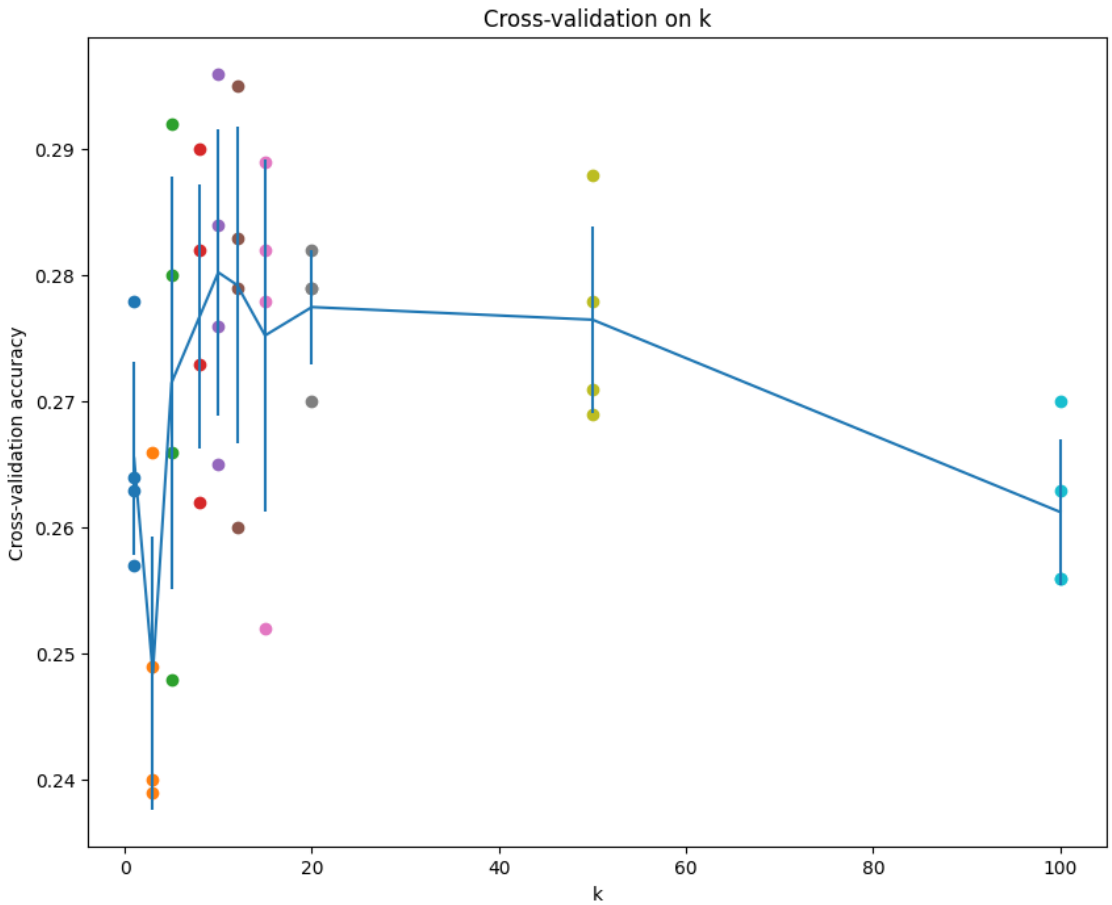
```
    except:
      if i == 0:
        X_train_set = X_train_folds[i+1:]
        y_train_set = y_train_folds[i+1:]
      elif i == num_folds:
        X_train_set = X_train_folds[:i]
        y_train_set = y_train_folds[:i]
      else:
        continue

    classifier.train(np.concatenate(X_train_set), np.concatenate(y_train_set))
    dists = classifier.compute_distances_no_loops(validation_set)
    y_test_pred = classifier.predict_labels(dists, k)
    num_correct = np.sum(y_test_pred == y_train_folds[i])
    accuracy = float(num_correct) / len(y_test_pred)
    k_to_accuracies[k].append(accuracy)



pass

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

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

```
k = 1, accuracy = 0.263000
k = 1, accuracy = 0.257000
k = 1, accuracy = 0.264000
k = 1, accuracy = 0.278000
k = 3, accuracy = 0.239000
k = 3, accuracy = 0.249000
k = 3, accuracy = 0.240000
k = 3, accuracy = 0.266000
k = 5, accuracy = 0.248000
k = 5, accuracy = 0.266000
k = 5, accuracy = 0.280000
k = 5, accuracy = 0.292000
k = 8, accuracy = 0.262000
k = 8, accuracy = 0.282000
k = 8, accuracy = 0.273000
k = 8, accuracy = 0.290000
k = 10, accuracy = 0.265000
k = 10, accuracy = 0.296000
k = 10, accuracy = 0.276000
```

```
k = 10, accuracy = 0.284000
k = 12, accuracy = 0.260000
k = 12, accuracy = 0.295000
k = 12, accuracy = 0.279000
k = 12, accuracy = 0.283000
k = 15, accuracy = 0.252000
k = 15, accuracy = 0.289000
k = 15, accuracy = 0.278000
k = 15, accuracy = 0.282000
k = 20, accuracy = 0.270000
k = 20, accuracy = 0.279000
k = 20, accuracy = 0.279000
k = 20, accuracy = 0.282000
k = 50, accuracy = 0.271000
k = 50, accuracy = 0.288000
k = 50, accuracy = 0.278000
k = 50, accuracy = 0.269000
k = 100, accuracy = 0.256000
k = 100, accuracy = 0.270000
k = 100, accuracy = 0.263000
k = 100, accuracy = 0.256000
```

[66]:

```python
# plot the raw observations
for k in k_choices:
    accuracies = k_to_accuracies[k]
    plt.scatter([k] * len(accuracies), accuracies)

# plot the trend line with error bars that correspond to standard deviation
accuracies_mean = np.array([np.mean(v) for k,v in sorted(k_to_accuracies.
 ↪items())])
accuracies_std = np.array([np.std(v) for k,v in sorted(k_to_accuracies.
 ↪items())])
plt.errorbar(k_choices, accuracies_mean, yerr=accuracies_std)
plt.title('Cross-validation on k')
plt.xlabel('k')
plt.ylabel('Cross-validation accuracy')
plt.show()
```



[64]:
```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.

$Your Answer: 2, 4$

$Your Explanation:$ 2) Training error of a 1-NN is always 0 as the nearest datapoint in the training set is the datapoint itself. 4) The classification in a kNN is a linear search at its heart, which scales with O(n). (n being the size of the training set)

svm

June 29, 2023

[39]:
```
# 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, "cs231n/assignments/assignment1/"

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

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

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

[40]:
```
# Run some setup code for this notebook.
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

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

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

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

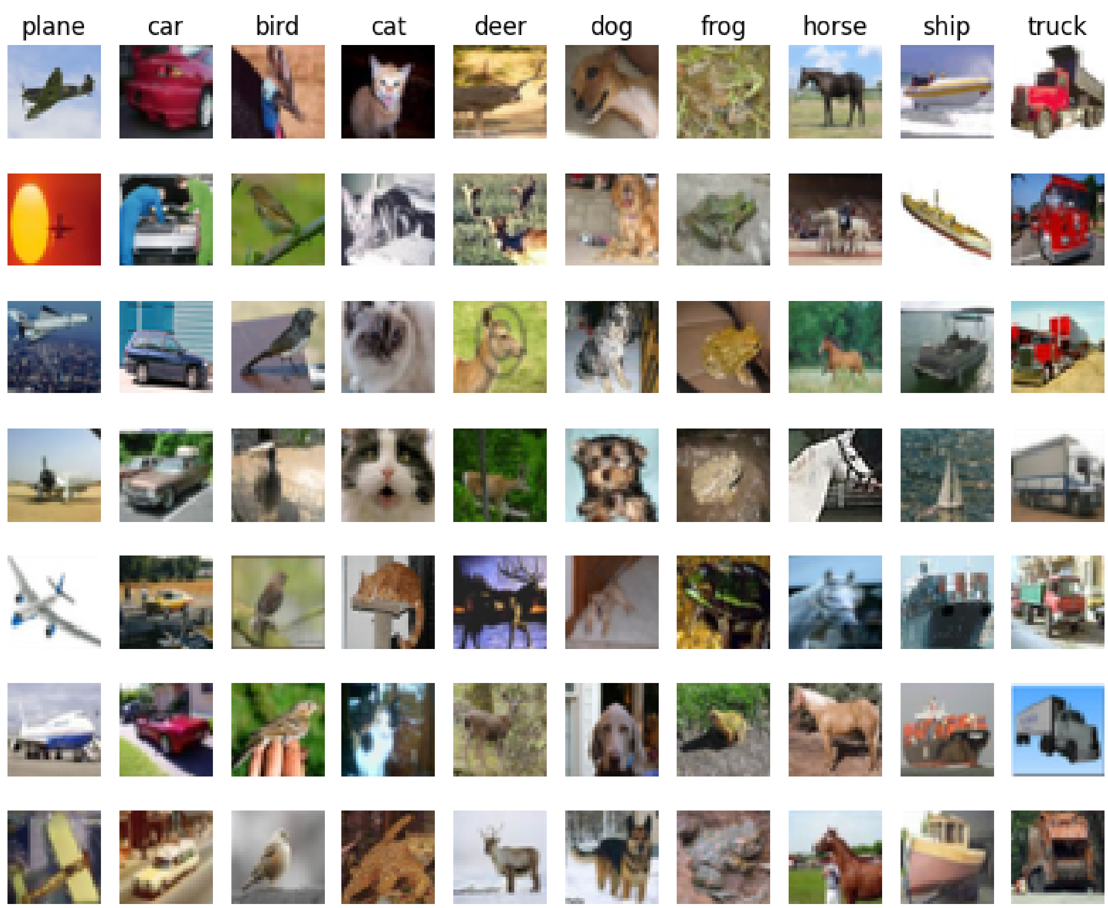
1.1 CIFAR-10 Data Loading and Preprocessing

[41]:
```
# 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,)
```

[42]:
```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()
```



[43]:

```python
# 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)
```

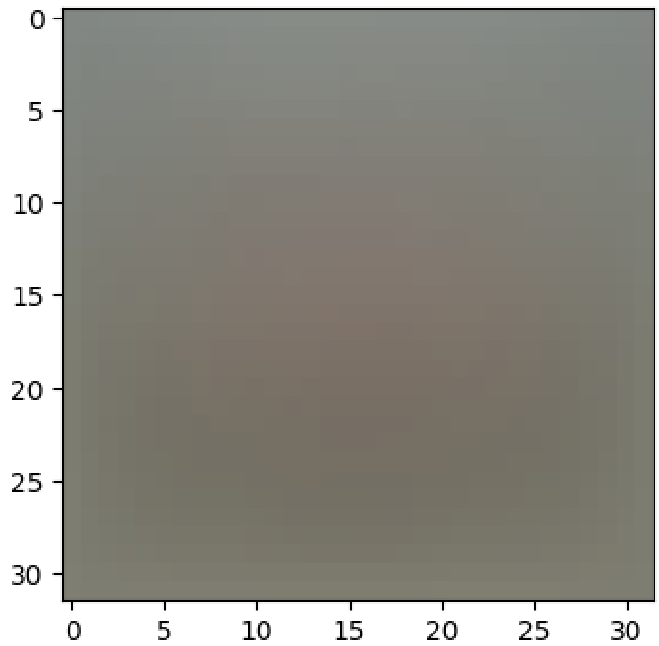
```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

[44]:
```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)
```

[45]:
```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.

[46]:
```python
# 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: 9.179041
```

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:

[47]:

```python
# 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: -1.014776 analytic: -1.014776, relative error: 2.157884e-10
numerical: -6.315420 analytic: -6.315420, relative error: 1.358886e-11
numerical: -1.483994 analytic: -1.483994, relative error: 4.910354e-10
numerical: 28.852215 analytic: 28.852215, relative error: 1.139158e-11
numerical: -9.482695 analytic: -9.482695, relative error: 3.473870e-12
numerical: 16.236101 analytic: 16.236101, relative error: 7.080985e-12
numerical: -35.915460 analytic: -35.915460, relative error: 6.558118e-12
numerical: 24.645735 analytic: 24.645735, relative error: 7.453052e-12
numerical: -10.948812 analytic: -10.948812, relative error: 1.446038e-11
numerical: 15.543184 analytic: 15.543184, relative error: 1.606336e-11
numerical: 7.412488 analytic: 7.412488, relative error: 3.188689e-11
numerical: 10.854518 analytic: 10.854518, relative error: 2.803119e-11
numerical: 4.690335 analytic: 4.690335, relative error: 7.129698e-11
numerical: 28.874445 analytic: 28.874445, relative error: 9.356951e-12
numerical: 5.734724 analytic: 5.734724, relative error: 2.079948e-11
numerical: 24.140537 analytic: 24.140537, relative error: 4.327989e-12
numerical: 9.092725 analytic: 9.092725, relative error: 1.757187e-11
numerical: 12.122950 analytic: 12.122950, relative error: 2.667755e-11
numerical: -5.826501 analytic: -5.826501, relative error: 1.022834e-11
numerical: 5.594174 analytic: 5.594174, relative error: 3.195676e-11
```

Inline Question 1

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

$Your Answer:$ *fill this in.*

[61]:
```
# 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: 9.179041e+00 computed in 0.072195s
(500, 10) (3073, 500)
Vectorized loss: 9.179041e+00 computed in 0.007785s
difference: 0.000000
```

[82]:
```
# 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 0.074029s
Vectorized loss and gradient: computed in 0.007139s
difference: 0.000000
```

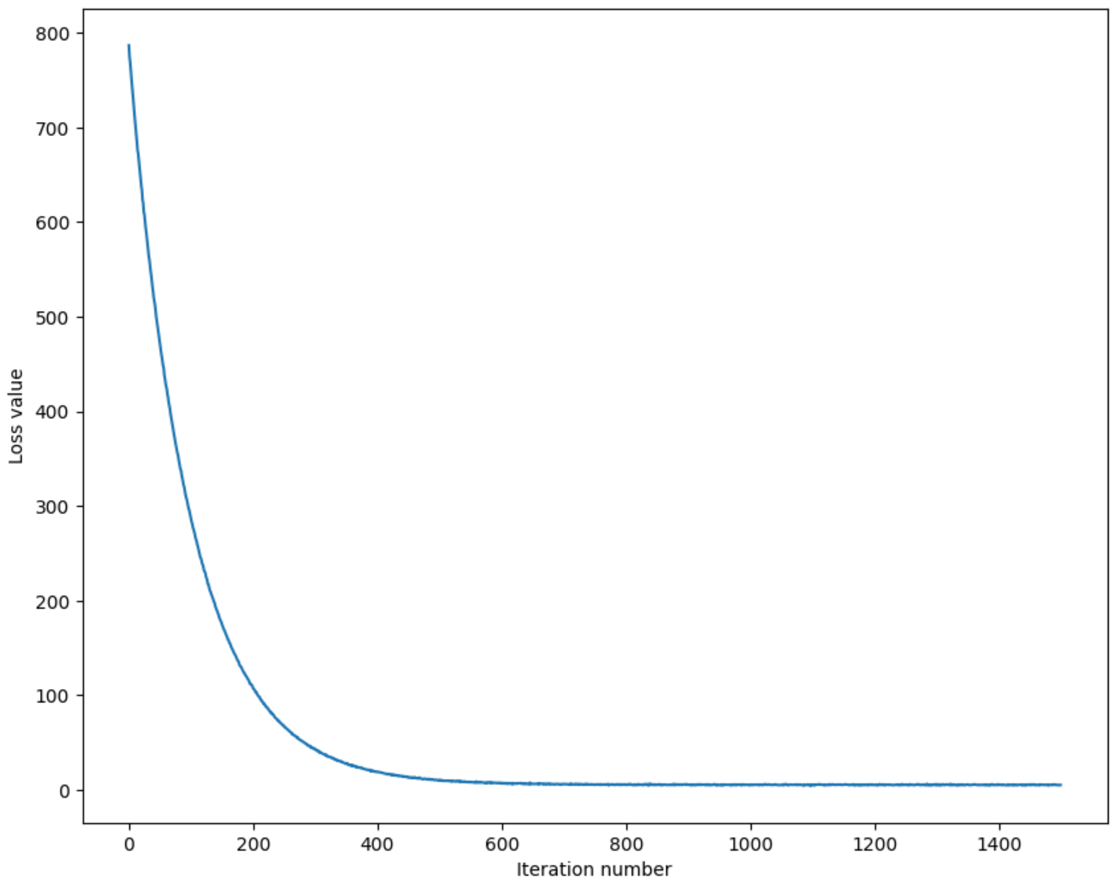
1.2.1 Stochastic Gradient Descent

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`.

[91]:
```
# 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 786.778885
iteration 100 / 1500: loss 286.912724
iteration 200 / 1500: loss 107.241274
iteration 300 / 1500: loss 42.717384
iteration 400 / 1500: loss 19.015385
iteration 500 / 1500: loss 9.965770
iteration 600 / 1500: loss 6.799083
iteration 700 / 1500: loss 6.133234
iteration 800 / 1500: loss 5.202870
iteration 900 / 1500: loss 5.643680
iteration 1000 / 1500: loss 5.231457
iteration 1100 / 1500: loss 5.358856
iteration 1200 / 1500: loss 5.077203
iteration 1300 / 1500: loss 5.818218
iteration 1400 / 1500: loss 5.504388
That took 14.241492s
```

[92]:
```
# 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()
```



[93]:

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

```
training accuracy: 0.364959
validation accuracy: 0.368000
```

[112]:

```
# 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
```

```python
# (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, 1e-8, 1e-6, 5e-8]
regularization_strengths = [5e3, 1e4, 2.5e4, 3e4]

# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
for lr in learning_rates:
  for reg_str in regularization_strengths:
    svm = LinearSVM()
    svm.train(X_train, y_train, learning_rate=lr, reg=reg_str,
                      num_iters=1500, verbose=False)
    y_pred = svm.predict(X_val)
    validation_accuracy = np.sum(y_pred == y_val) / len(y_val)
    y_pred = svm.predict(X_train)
    training_accuracy = np.sum(y_pred == y_train) / len(y_train)
    results[(lr, reg_str)] = (training_accuracy, validation_accuracy)
    if validation_accuracy > best_val:
      best_val = validation_accuracy
      best_svm = svm

pass

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

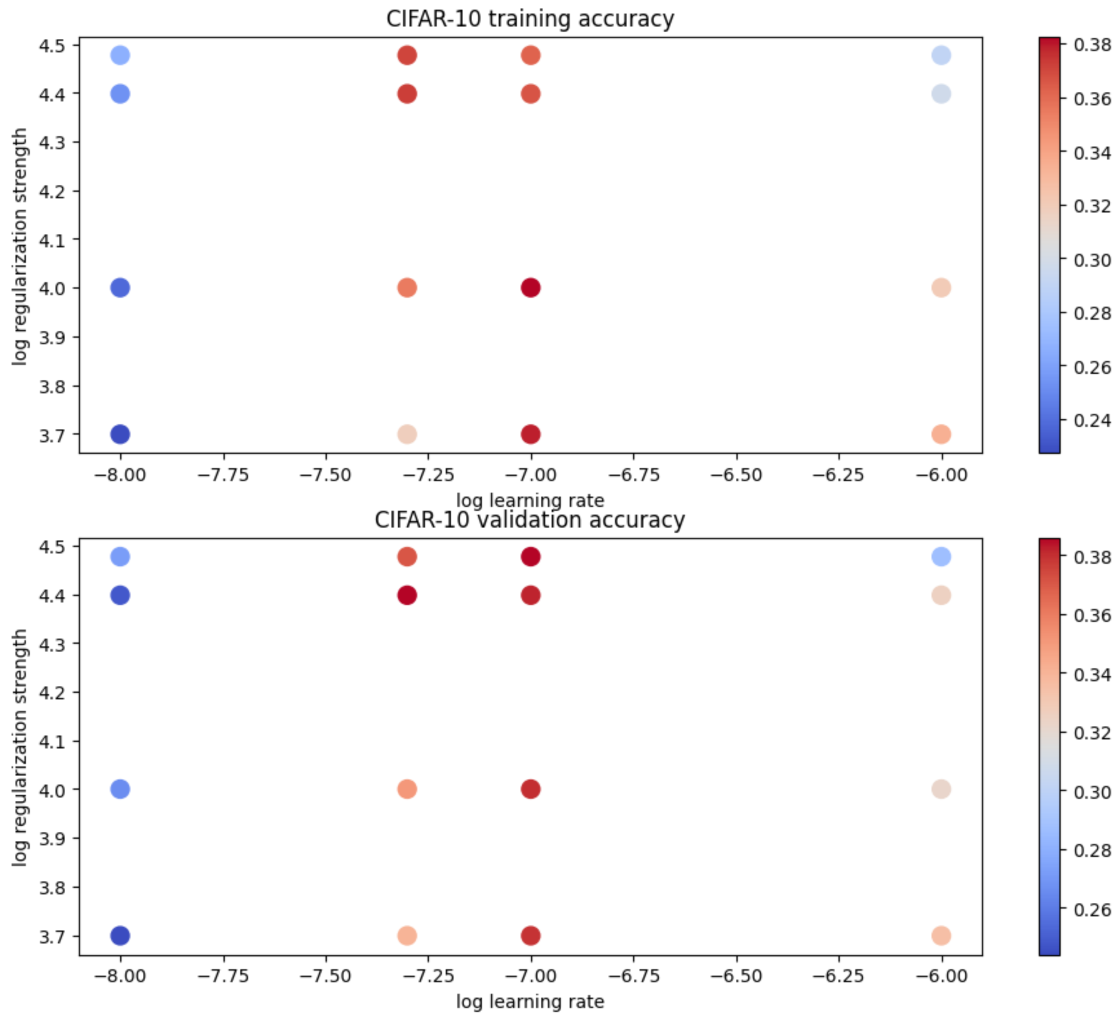
```
# Print out results.
for lr, reg in sorted(results):
    train_accuracy, val_accuracy = results[(lr, reg)]
    print('lr %e reg %e train accuracy: %f val accuracy: %f' % (
                lr, reg, train_accuracy, val_accuracy))

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

```
lr 1.000000e-08 reg 5.000000e+03 train accuracy: 0.227286 val accuracy: 0.244000
lr 1.000000e-08 reg 1.000000e+04 train accuracy: 0.238184 val accuracy: 0.266000
lr 1.000000e-08 reg 2.500000e+04 train accuracy: 0.253286 val accuracy: 0.248000
lr 1.000000e-08 reg 3.000000e+04 train accuracy: 0.266388 val accuracy: 0.272000
lr 5.000000e-08 reg 5.000000e+03 train accuracy: 0.318265 val accuracy: 0.340000
lr 5.000000e-08 reg 1.000000e+04 train accuracy: 0.354735 val accuracy: 0.351000
lr 5.000000e-08 reg 2.500000e+04 train accuracy: 0.372837 val accuracy: 0.386000
lr 5.000000e-08 reg 3.000000e+04 train accuracy: 0.370939 val accuracy: 0.371000
lr 1.000000e-07 reg 5.000000e+03 train accuracy: 0.378837 val accuracy: 0.379000
lr 1.000000e-07 reg 1.000000e+04 train accuracy: 0.382694 val accuracy: 0.380000
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.366429 val accuracy: 0.382000
lr 1.000000e-07 reg 3.000000e+04 train accuracy: 0.362367 val accuracy: 0.386000
lr 1.000000e-06 reg 5.000000e+03 train accuracy: 0.334592 val accuracy: 0.335000
lr 1.000000e-06 reg 1.000000e+04 train accuracy: 0.320673 val accuracy: 0.322000
lr 1.000000e-06 reg 2.500000e+04 train accuracy: 0.297245 val accuracy: 0.325000
lr 1.000000e-06 reg 3.000000e+04 train accuracy: 0.289837 val accuracy: 0.287000
best validation accuracy achieved during cross-validation: 0.386000
```

[113]:

```
# 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()
```



[114]:
```
# Evaluate the best svm on test set
y_test_pred = best_svm.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
```

```
linear SVM on raw pixels final test set accuracy: 0.372000
```

[115]:

```python
# 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

June 29, 2023

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

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

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

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

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

1 Softmax exercise

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

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

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

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

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

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

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

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

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

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

```
    mask = np.random.choice(num_training, num_dev, replace=False)
    X_dev = X_train[mask]
    y_dev = y_train[mask]

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

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

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

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


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

```
Train data shape:  (49000, 3073)
Train labels shape:  (49000,)
Validation data shape:  (1000, 3073)
Validation labels shape:  (1000,)
Test data shape:  (1000, 3073)
Test labels shape:  (1000,)
dev data shape:  (500, 3073)
dev labels shape:  (500,)
```

1.1 Softmax Classifier

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

[19]:
```
# 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: 2.397620
sanity check: 2.302585
```

Inline Question 1

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

$Your Answer:$ *Since there are 10 classes and the initial weights are randomly distributed between classes, the system guessing the correct class has a probability of 0.1*

[18]:
```
# 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: 1.917720 analytic: 1.917720, relative error: 3.001296e-08
numerical: -0.475834 analytic: -0.475834, relative error: 1.015283e-07
numerical: 0.954850 analytic: 0.954850, relative error: 4.479967e-08
numerical: -3.243446 analytic: -3.243446, relative error: 1.984364e-08
numerical: 0.944306 analytic: 0.944306, relative error: 2.826871e-08
numerical: 2.374205 analytic: 2.374204, relative error: 2.258077e-08
```

```
numerical: -2.084182 analytic: -2.084182, relative error: 1.801168e-08
numerical: -0.514204 analytic: -0.514204, relative error: 1.019706e-07
numerical: -1.538442 analytic: -1.538442, relative error: 3.313266e-11
numerical: -0.166803 analytic: -0.166804, relative error: 6.203692e-07
numerical: -1.945028 analytic: -1.945028, relative error: 1.184581e-08
numerical: 0.963225 analytic: 0.963225, relative error: 2.306702e-08
numerical: -1.208382 analytic: -1.208382, relative error: 6.506099e-09
numerical: -0.040918 analytic: -0.040918, relative error: 6.576977e-07
numerical: -1.640869 analytic: -1.640869, relative error: 4.714041e-09
numerical: -1.905214 analytic: -1.905214, relative error: 1.272606e-08
numerical: 0.518419 analytic: 0.518419, relative error: 7.912633e-08
numerical: 0.389615 analytic: 0.389614, relative error: 1.225383e-07
numerical: -0.002104 analytic: -0.002104, relative error: 1.801524e-05
numerical: 0.109785 analytic: 0.109785, relative error: 1.122969e-07
```

[39]:

```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: 2.397620e+00 computed in 0.054942s
vectorized loss: 2.397620e+00 computed in 0.011222s
Loss difference: 0.000000
Gradient difference: 0.000000
```

[46]:

```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
```

```python
# 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-8, 5e-7]
regularization_strengths = [5e3, 1e4]

# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
for lr in learning_rates:
  for reg_str in regularization_strengths:
    softmax = Softmax()
    softmax.train(X_train, y_train, learning_rate=lr, reg=reg_str,
                  num_iters=1500, verbose=False)
    y_pred = softmax.predict(X_val)
    validation_accuracy = np.sum(y_pred == y_val) / len(y_val)
    y_pred = softmax.predict(X_train)
    training_accuracy = np.sum(y_pred == y_train) / len(y_train)
    results[(lr, reg_str)] = (training_accuracy, validation_accuracy)
    if validation_accuracy > best_val:
      best_val = validation_accuracy
      best_softmax = softmax

pass

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

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

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

```
lr 5.000000e-08 reg 5.000000e+03 train accuracy: 0.259551 val accuracy: 0.270000
lr 5.000000e-08 reg 1.000000e+04 train accuracy: 0.301592 val accuracy: 0.314000
lr 1.000000e-07 reg 5.000000e+03 train accuracy: 0.336612 val accuracy: 0.343000
lr 1.000000e-07 reg 1.000000e+04 train accuracy: 0.355224 val accuracy: 0.366000
lr 5.000000e-07 reg 5.000000e+03 train accuracy: 0.370143 val accuracy: 0.377000
lr 5.000000e-07 reg 1.000000e+04 train accuracy: 0.355204 val accuracy: 0.363000
best validation accuracy achieved during cross-validation: 0.377000
```

[47]:
```
# evaluate on test set
# Evaluate the best softmax on test set
y_test_pred = best_softmax.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('softmax on raw pixels final test set accuracy: %f' % (test_accuracy, ))
```

```
softmax on raw pixels final test set accuracy: 0.376000
```

Inline Question 2 - *True or False*

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

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

$\color{blue}{\textit{Your Explanation :}}$ SVM loss doesn't care about the values attributed to the classes as long as the correct one is above all the others by some margin δ. Thus a datapoint that *would be* correctly labeled by the initial system would have a per-datapoint loss of 0. And the addition of such a point to the training set wouldn't effect the total loss.

But the softmax needs all the incorrect classes to have -inf score due to the exponential summation in its definition. Unless this is the case (and it never is), total loss will increase.

[48]:
```
# 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

June 29, 2023

[1]:
```python
# 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, 'cs231n/assignments/assignment1/'

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

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

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

1 Fully-Connected Neural Nets

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

```python
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.

[2]:
```
# 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))))
```

[3]:
```
# 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:

[4]:
```
# 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.

[5]:
```
# 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)
print(db, db_num)
# 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))
```

```
[-5.78588657 -2.14288946 -3.93648137 -4.10664587 -0.09253319] [-5.78588657
-2.14288946 -3.93648137 -4.10664587 -0.09253319]
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:

[6]:
```
# 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:

[7]:
```
np.random.seed(231)
x = np.random.randn(10, 10)
dout = np.random.randn(*x.shape)

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

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

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

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

5.1 Inline Question 1:

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

5.2 Answer:

1, Sigmoid yields a near zero gradient when its input is too large or too small, ie +-inf

2, ReLU ignores what happens when the input dips below 0, so it always produces a gradient of 0 when the input is negative.

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:

[8]:
```
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:

[14]:
```
np.random.seed(231)
num_classes, num_inputs = 10, 50
x = 0.001 * np.random.randn(num_inputs, num_classes)
```

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

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

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

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

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

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

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

8 Two-layer network

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

[22]:
```
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)
```

```python
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: 4.48e-10
b1 relative error: 1.02e-08
b2 relative error: 1.28e-09
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: 8.89e-10
```

9 Solver

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

[39]:
```
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
model = TwoLayerNet(input_size, hidden_size, num_classes)
solver = None

##############################################################################
# TODO: Use a Solver instance to train a TwoLayerNet that achieves about 36% #
# accuracy on the validation set.                                            #
##############################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
solver = Solver(model, data,
                update_rule='sgd',
                optim_config={"learning_rate" : 1e-4},
                print_every=100,
                verbose=True,
                )
solver.train()
print(solver.check_accuracy(data["X_test"], data["y_test"]))
pass

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

```
(Iteration 1 / 4900) loss: 2.302141
(Epoch 0 / 10) train acc: 0.090000; val_acc: 0.082000
(Iteration 101 / 4900) loss: 2.259294
(Iteration 201 / 4900) loss: 2.137232
```

```
(Iteration 301 / 4900) loss: 2.023864
(Iteration 401 / 4900) loss: 1.930679
(Epoch 1 / 10) train acc: 0.271000; val_acc: 0.298000
(Iteration 501 / 4900) loss: 1.915765
(Iteration 601 / 4900) loss: 1.943824
(Iteration 701 / 4900) loss: 1.871475
(Iteration 801 / 4900) loss: 1.834302
(Iteration 901 / 4900) loss: 1.844593
(Epoch 2 / 10) train acc: 0.354000; val_acc: 0.368000
(Iteration 1001 / 4900) loss: 1.754067
(Iteration 1101 / 4900) loss: 1.737627
(Iteration 1201 / 4900) loss: 1.829923
(Iteration 1301 / 4900) loss: 1.807035
(Iteration 1401 / 4900) loss: 1.734696
(Epoch 3 / 10) train acc: 0.380000; val_acc: 0.389000
(Iteration 1501 / 4900) loss: 1.605982
(Iteration 1601 / 4900) loss: 1.603989
(Iteration 1701 / 4900) loss: 1.827841
(Iteration 1801 / 4900) loss: 1.734655
(Iteration 1901 / 4900) loss: 1.510981
(Epoch 4 / 10) train acc: 0.418000; val_acc: 0.402000
(Iteration 2001 / 4900) loss: 1.584240
(Iteration 2101 / 4900) loss: 1.637732
(Iteration 2201 / 4900) loss: 1.645530
(Iteration 2301 / 4900) loss: 1.615908
(Iteration 2401 / 4900) loss: 1.619432
(Epoch 5 / 10) train acc: 0.411000; val_acc: 0.450000
(Iteration 2501 / 4900) loss: 1.410676
(Iteration 2601 / 4900) loss: 1.604972
(Iteration 2701 / 4900) loss: 1.446726
(Iteration 2801 / 4900) loss: 1.673368
(Iteration 2901 / 4900) loss: 1.522190
(Epoch 6 / 10) train acc: 0.460000; val_acc: 0.455000
(Iteration 3001 / 4900) loss: 1.454931
(Iteration 3101 / 4900) loss: 1.610937
(Iteration 3201 / 4900) loss: 1.446659
(Iteration 3301 / 4900) loss: 1.440608
(Iteration 3401 / 4900) loss: 1.515097
(Epoch 7 / 10) train acc: 0.470000; val_acc: 0.456000
(Iteration 3501 / 4900) loss: 1.632408
(Iteration 3601 / 4900) loss: 1.446348
(Iteration 3701 / 4900) loss: 1.455543
(Iteration 3801 / 4900) loss: 1.530741
(Iteration 3901 / 4900) loss: 1.389851
(Epoch 8 / 10) train acc: 0.481000; val_acc: 0.465000
(Iteration 4001 / 4900) loss: 1.532374
(Iteration 4101 / 4900) loss: 1.373307
(Iteration 4201 / 4900) loss: 1.443720
```

```
(Iteration 4301 / 4900) loss: 1.367583
(Iteration 4401 / 4900) loss: 1.451003
(Epoch 9 / 10) train acc: 0.456000; val_acc: 0.467000
(Iteration 4501 / 4900) loss: 1.545870
(Iteration 4601 / 4900) loss: 1.503116
(Iteration 4701 / 4900) loss: 1.415303
(Iteration 4801 / 4900) loss: 1.406984
(Epoch 10 / 10) train acc: 0.490000; val_acc: 0.468000
0.465
```

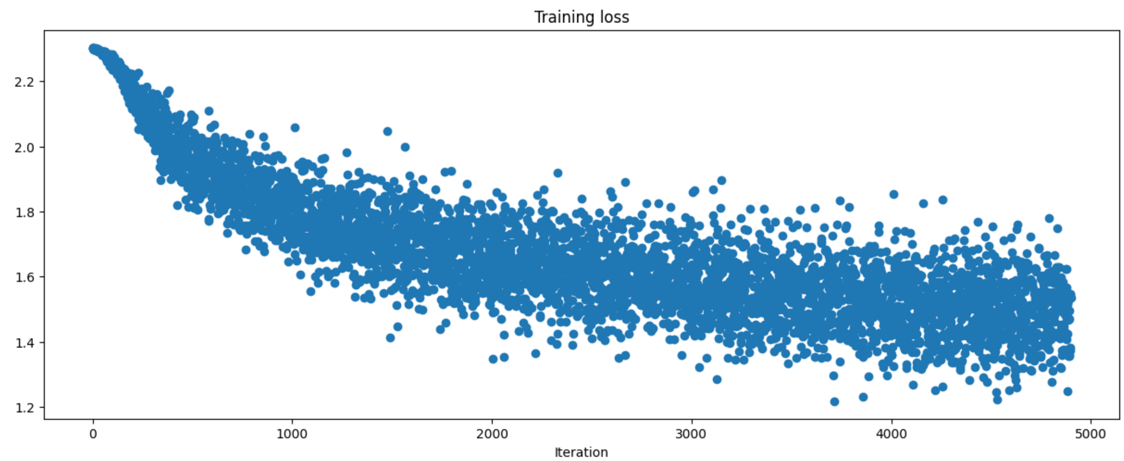
10 Debug the training

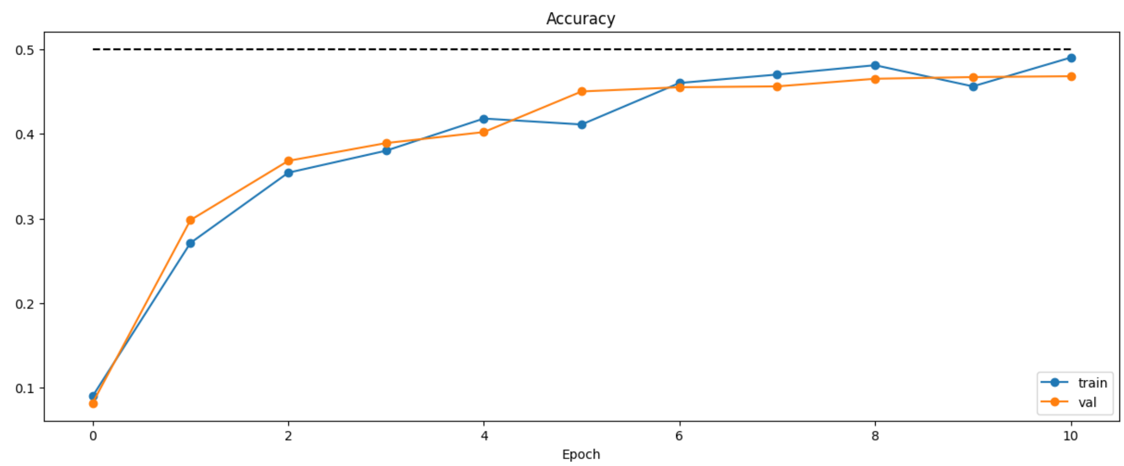
With the default parameters we provided above, you should get a validation accuracy of about 0.36 on the validation set. This isn't very good.

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.

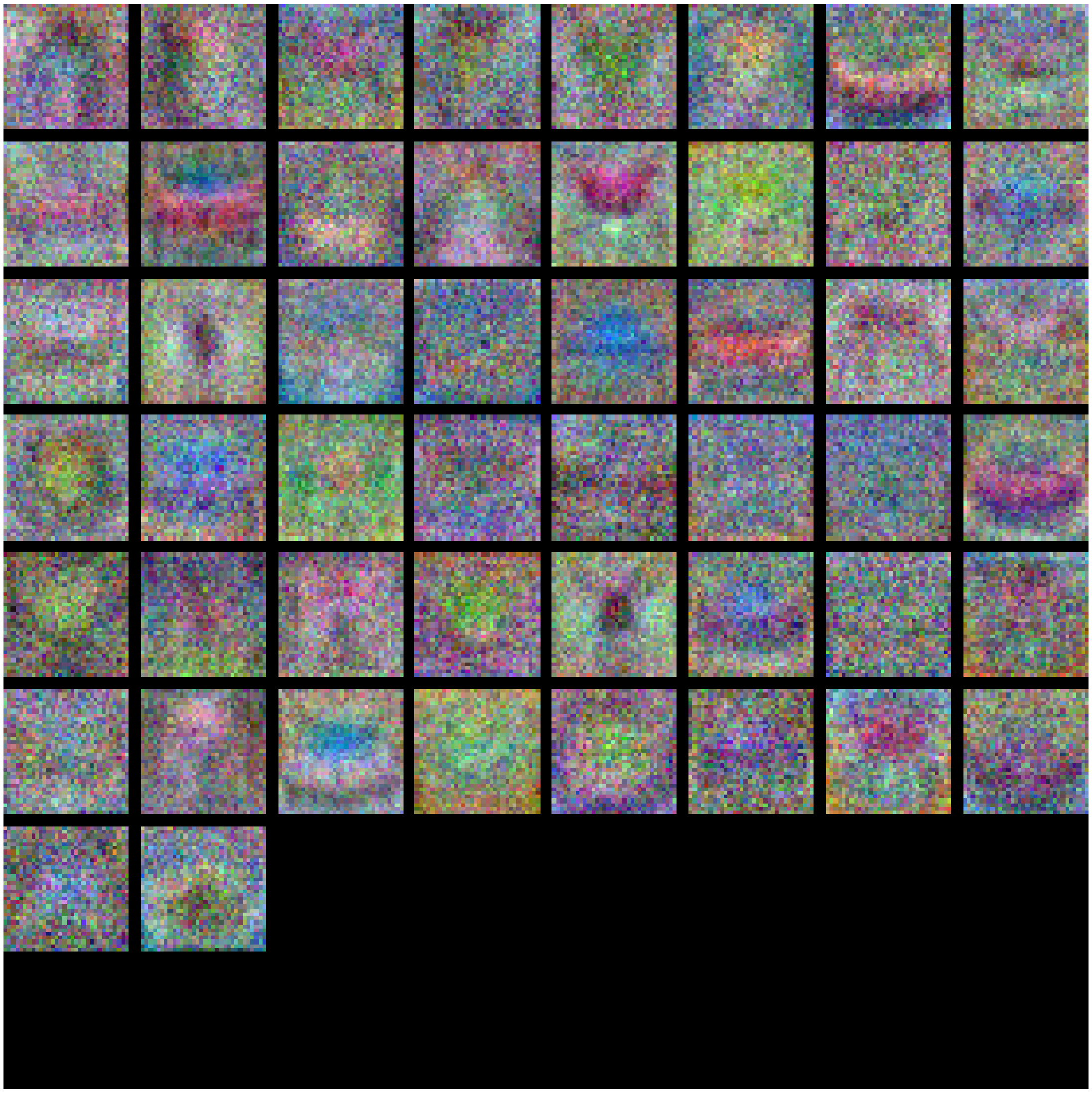
Another strategy is to visualize the weights that were learned in the first layer of the network. In most neural networks trained on visual data, the first layer weights typically show some visible structure when visualized.

[40]:
```
# Run this cell to visualize training loss and train / val accuracy

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

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





[41]:
```
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.).

[48]:
```
best_model = None


################################################################################
# TODO: Tune hyperparameters using the validation set. Store your best trained ␣
↪#
# model in best_model.                                                          ␣
↪#
#                                                                               ␣
↪#
# To help debug your network, it may help to use visualizations similar to the ␣
↪#
# ones we used above; these visualizations will have significant qualitative    ␣
↪#
# differences from the ones we saw above for the poorly tuned network.          ␣
↪#
#                                                                               ␣
↪#
# Tweaking hyperparameters by hand can be fun, but you might find it useful to ␣
↪#
# write code to sweep through possible combinations of hyperparameters          ␣
↪#
# automatically like we did on thexs previous exercises.                        ␣
↪  #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
best_val_acc = 0
# learning_rates = [1e-4, 1e-3, 1]
# lr_decays = [1, 0.9, 0.80]
# hidden_sizes = [50, 100]
# reg_strengths = [1, 1.5,  2]

learning_rates = [1e-3]
lr_decays = [0.9]
hidden_sizes = [100]
reg_strengths = [1.5]
for reg_str in reg_strengths:
```

```
  for lr in learning_rates:
    for h_size in hidden_sizes:
      for lr_dec in lr_decays:
        model = TwoLayerNet(input_size, h_size, num_classes)
        model.reg = reg_str
        solver = Solver(model, data,
                        update_rule='sgd',
                        optim_config={"learning_rate" : lr},
                        lr_decay = lr_dec,
                        verbose=True,
                        )
        solver.train()
        val_acc = solver.check_accuracy(data['X_test'], data['y_test'],␣
↪num_samples=500)
        if val_acc > best_val_acc:
          show_net_weights(model)
          print("rate: ", lr, "decay: ", lr_dec, "size: ", h_size, "reg: ",␣
↪reg_str, "acc: ", val_acc)
          best_model = model
          best_val_acc = val_acc


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

```
(Iteration 1 / 4900) loss: 2.535917
(Epoch 0 / 10) train acc: 0.135000; val_acc: 0.116000
(Iteration 11 / 4900) loss: 2.461780
(Iteration 21 / 4900) loss: 2.419824
(Iteration 31 / 4900) loss: 2.265204
(Iteration 41 / 4900) loss: 2.222421
```

```
(Iteration 51 / 4900) loss: 2.155204
(Iteration 61 / 4900) loss: 2.043371
(Iteration 71 / 4900) loss: 2.148803
(Iteration 81 / 4900) loss: 2.049006
(Iteration 91 / 4900) loss: 2.107303
(Iteration 101 / 4900) loss: 1.999389
(Iteration 111 / 4900) loss: 2.044995
(Iteration 121 / 4900) loss: 1.958615
(Iteration 131 / 4900) loss: 1.958148
(Iteration 141 / 4900) loss: 2.084269
(Iteration 151 / 4900) loss: 2.119764
(Iteration 161 / 4900) loss: 1.902024
(Iteration 171 / 4900) loss: 1.823547
(Iteration 181 / 4900) loss: 1.863793
(Iteration 191 / 4900) loss: 1.813701
(Iteration 201 / 4900) loss: 1.623181
(Iteration 211 / 4900) loss: 1.857096
(Iteration 221 / 4900) loss: 1.870340
(Iteration 231 / 4900) loss: 1.798633
(Iteration 241 / 4900) loss: 1.973649
(Iteration 251 / 4900) loss: 1.774786
(Iteration 261 / 4900) loss: 1.703025
(Iteration 271 / 4900) loss: 1.800845
(Iteration 281 / 4900) loss: 1.828464
(Iteration 291 / 4900) loss: 2.026223
(Iteration 301 / 4900) loss: 1.892378
(Iteration 311 / 4900) loss: 1.917421
(Iteration 321 / 4900) loss: 1.786579
(Iteration 331 / 4900) loss: 1.653373
(Iteration 341 / 4900) loss: 1.950165
(Iteration 351 / 4900) loss: 1.757145
(Iteration 361 / 4900) loss: 1.833896
(Iteration 371 / 4900) loss: 1.587047
(Iteration 381 / 4900) loss: 1.689755
(Iteration 391 / 4900) loss: 1.775785
(Iteration 401 / 4900) loss: 1.615491
(Iteration 411 / 4900) loss: 1.854960
(Iteration 421 / 4900) loss: 1.795681
(Iteration 431 / 4900) loss: 1.717875
(Iteration 441 / 4900) loss: 1.778226
(Iteration 451 / 4900) loss: 1.715462
(Iteration 461 / 4900) loss: 1.675978
(Iteration 471 / 4900) loss: 1.820023
(Iteration 481 / 4900) loss: 1.779396
(Epoch 1 / 10) train acc: 0.470000; val_acc: 0.445000
(Iteration 491 / 4900) loss: 1.754718
(Iteration 501 / 4900) loss: 1.839588
(Iteration 511 / 4900) loss: 1.711345
```

```
(Iteration 521 / 4900) loss: 1.718406
(Iteration 531 / 4900) loss: 1.685445
(Iteration 541 / 4900) loss: 1.589183
(Iteration 551 / 4900) loss: 1.532496
(Iteration 561 / 4900) loss: 1.647836
(Iteration 571 / 4900) loss: 1.667740
(Iteration 581 / 4900) loss: 1.617279
(Iteration 591 / 4900) loss: 1.702520
(Iteration 601 / 4900) loss: 1.776811
(Iteration 611 / 4900) loss: 1.921727
(Iteration 621 / 4900) loss: 1.720654
(Iteration 631 / 4900) loss: 1.606880
(Iteration 641 / 4900) loss: 1.541608
(Iteration 651 / 4900) loss: 1.741332
(Iteration 661 / 4900) loss: 1.846052
(Iteration 671 / 4900) loss: 1.649896
(Iteration 681 / 4900) loss: 1.756592
(Iteration 691 / 4900) loss: 1.774822
(Iteration 701 / 4900) loss: 1.802912
(Iteration 711 / 4900) loss: 1.637255
(Iteration 721 / 4900) loss: 1.766144
(Iteration 731 / 4900) loss: 1.721383
(Iteration 741 / 4900) loss: 1.565537
(Iteration 751 / 4900) loss: 1.637088
(Iteration 761 / 4900) loss: 1.699200
(Iteration 771 / 4900) loss: 1.647744
(Iteration 781 / 4900) loss: 1.719960
(Iteration 791 / 4900) loss: 1.997053
(Iteration 801 / 4900) loss: 1.871084
(Iteration 811 / 4900) loss: 1.666414
(Iteration 821 / 4900) loss: 1.445032
(Iteration 831 / 4900) loss: 1.590016
(Iteration 841 / 4900) loss: 1.846674
(Iteration 851 / 4900) loss: 1.619241
(Iteration 861 / 4900) loss: 1.762209
(Iteration 871 / 4900) loss: 1.684865
(Iteration 881 / 4900) loss: 1.636697
(Iteration 891 / 4900) loss: 1.692846
(Iteration 901 / 4900) loss: 1.658841
(Iteration 911 / 4900) loss: 1.793990
(Iteration 921 / 4900) loss: 1.557587
(Iteration 931 / 4900) loss: 1.796557
(Iteration 941 / 4900) loss: 1.588471
(Iteration 951 / 4900) loss: 1.674817
(Iteration 961 / 4900) loss: 1.680322
(Iteration 971 / 4900) loss: 1.701271
(Epoch 2 / 10) train acc: 0.490000; val_acc: 0.464000
(Iteration 981 / 4900) loss: 1.626983
```

```
(Iteration 991 / 4900) loss: 1.716028
(Iteration 1001 / 4900) loss: 1.666446
(Iteration 1011 / 4900) loss: 1.769607
(Iteration 1021 / 4900) loss: 1.681368
(Iteration 1031 / 4900) loss: 1.685215
(Iteration 1041 / 4900) loss: 1.593389
(Iteration 1051 / 4900) loss: 1.736450
(Iteration 1061 / 4900) loss: 1.587354
(Iteration 1071 / 4900) loss: 1.929315
(Iteration 1081 / 4900) loss: 1.496213
(Iteration 1091 / 4900) loss: 1.852744
(Iteration 1101 / 4900) loss: 1.709877
(Iteration 1111 / 4900) loss: 1.714794
(Iteration 1121 / 4900) loss: 1.596609
(Iteration 1131 / 4900) loss: 1.675499
(Iteration 1141 / 4900) loss: 1.566929
(Iteration 1151 / 4900) loss: 1.598207
(Iteration 1161 / 4900) loss: 1.628591
(Iteration 1171 / 4900) loss: 1.588280
(Iteration 1181 / 4900) loss: 1.880313
(Iteration 1191 / 4900) loss: 1.602815
(Iteration 1201 / 4900) loss: 1.546384
(Iteration 1211 / 4900) loss: 1.561647
(Iteration 1221 / 4900) loss: 1.568441
(Iteration 1231 / 4900) loss: 1.616605
(Iteration 1241 / 4900) loss: 1.849452
(Iteration 1251 / 4900) loss: 1.522187
(Iteration 1261 / 4900) loss: 1.708325
(Iteration 1271 / 4900) loss: 1.622386
(Iteration 1281 / 4900) loss: 1.791382
(Iteration 1291 / 4900) loss: 1.577943
(Iteration 1301 / 4900) loss: 1.686558
(Iteration 1311 / 4900) loss: 1.724218
(Iteration 1321 / 4900) loss: 1.585011
(Iteration 1331 / 4900) loss: 1.695711
(Iteration 1341 / 4900) loss: 1.811967
(Iteration 1351 / 4900) loss: 1.847804
(Iteration 1361 / 4900) loss: 1.706963
(Iteration 1371 / 4900) loss: 1.647196
(Iteration 1381 / 4900) loss: 1.893362
(Iteration 1391 / 4900) loss: 1.614232
(Iteration 1401 / 4900) loss: 1.591988
(Iteration 1411 / 4900) loss: 1.606071
(Iteration 1421 / 4900) loss: 1.653350
(Iteration 1431 / 4900) loss: 1.699838
(Iteration 1441 / 4900) loss: 1.564287
(Iteration 1451 / 4900) loss: 1.759228
(Iteration 1461 / 4900) loss: 1.739914
```

```
(Epoch 3 / 10) train acc: 0.491000; val_acc: 0.477000
(Iteration 1471 / 4900) loss: 1.647935
(Iteration 1481 / 4900) loss: 1.731183
(Iteration 1491 / 4900) loss: 1.667366
(Iteration 1501 / 4900) loss: 1.743861
(Iteration 1511 / 4900) loss: 1.699459
(Iteration 1521 / 4900) loss: 1.601369
(Iteration 1531 / 4900) loss: 1.440965
(Iteration 1541 / 4900) loss: 1.566928
(Iteration 1551 / 4900) loss: 1.686336
(Iteration 1561 / 4900) loss: 1.691915
(Iteration 1571 / 4900) loss: 1.690127
(Iteration 1581 / 4900) loss: 1.692570
(Iteration 1591 / 4900) loss: 1.658590
(Iteration 1601 / 4900) loss: 1.463167
(Iteration 1611 / 4900) loss: 1.797847
(Iteration 1621 / 4900) loss: 1.557575
(Iteration 1631 / 4900) loss: 1.640158
(Iteration 1641 / 4900) loss: 1.597134
(Iteration 1651 / 4900) loss: 1.707786
(Iteration 1661 / 4900) loss: 1.615929
(Iteration 1671 / 4900) loss: 1.649196
(Iteration 1681 / 4900) loss: 1.623735
(Iteration 1691 / 4900) loss: 1.706724
(Iteration 1701 / 4900) loss: 1.594829
(Iteration 1711 / 4900) loss: 1.558807
(Iteration 1721 / 4900) loss: 1.557020
(Iteration 1731 / 4900) loss: 1.696102
(Iteration 1741 / 4900) loss: 1.574471
(Iteration 1751 / 4900) loss: 1.673151
(Iteration 1761 / 4900) loss: 1.661282
(Iteration 1771 / 4900) loss: 1.732003
(Iteration 1781 / 4900) loss: 1.628344
(Iteration 1791 / 4900) loss: 1.491130
(Iteration 1801 / 4900) loss: 1.649885
(Iteration 1811 / 4900) loss: 1.571139
(Iteration 1821 / 4900) loss: 1.619911
(Iteration 1831 / 4900) loss: 1.619017
(Iteration 1841 / 4900) loss: 1.630063
(Iteration 1851 / 4900) loss: 1.669721
(Iteration 1861 / 4900) loss: 1.678505
(Iteration 1871 / 4900) loss: 1.667347
(Iteration 1881 / 4900) loss: 1.647249
(Iteration 1891 / 4900) loss: 1.612820
(Iteration 1901 / 4900) loss: 1.638832
(Iteration 1911 / 4900) loss: 1.652459
(Iteration 1921 / 4900) loss: 1.583535
(Iteration 1931 / 4900) loss: 1.464967
```

```
(Iteration 1941 / 4900) loss: 1.585437
(Iteration 1951 / 4900) loss: 1.643124
(Epoch 4 / 10) train acc: 0.526000; val_acc: 0.473000
(Iteration 1961 / 4900) loss: 1.673754
(Iteration 1971 / 4900) loss: 1.644953
(Iteration 1981 / 4900) loss: 1.693926
(Iteration 1991 / 4900) loss: 1.462351
(Iteration 2001 / 4900) loss: 1.621956
(Iteration 2011 / 4900) loss: 1.590207
(Iteration 2021 / 4900) loss: 1.805538
(Iteration 2031 / 4900) loss: 1.728753
(Iteration 2041 / 4900) loss: 1.482578
(Iteration 2051 / 4900) loss: 1.659863
(Iteration 2061 / 4900) loss: 1.546243
(Iteration 2071 / 4900) loss: 1.608826
(Iteration 2081 / 4900) loss: 1.593666
(Iteration 2091 / 4900) loss: 1.551230
(Iteration 2101 / 4900) loss: 1.673987
(Iteration 2111 / 4900) loss: 1.589709
(Iteration 2121 / 4900) loss: 1.692709
(Iteration 2131 / 4900) loss: 1.411109
(Iteration 2141 / 4900) loss: 1.588755
(Iteration 2151 / 4900) loss: 1.356627
(Iteration 2161 / 4900) loss: 1.512411
(Iteration 2171 / 4900) loss: 1.453321
(Iteration 2181 / 4900) loss: 1.515951
(Iteration 2191 / 4900) loss: 1.815953
(Iteration 2201 / 4900) loss: 1.567988
(Iteration 2211 / 4900) loss: 1.444466
(Iteration 2221 / 4900) loss: 1.349773
(Iteration 2231 / 4900) loss: 1.541487
(Iteration 2241 / 4900) loss: 1.518835
(Iteration 2251 / 4900) loss: 1.743894
(Iteration 2261 / 4900) loss: 1.733920
(Iteration 2271 / 4900) loss: 1.702526
(Iteration 2281 / 4900) loss: 1.605479
(Iteration 2291 / 4900) loss: 1.544812
(Iteration 2301 / 4900) loss: 1.576284
(Iteration 2311 / 4900) loss: 1.668349
(Iteration 2321 / 4900) loss: 1.433738
(Iteration 2331 / 4900) loss: 1.602332
(Iteration 2341 / 4900) loss: 1.531487
(Iteration 2351 / 4900) loss: 1.599876
(Iteration 2361 / 4900) loss: 1.682114
(Iteration 2371 / 4900) loss: 1.675401
(Iteration 2381 / 4900) loss: 1.468103
(Iteration 2391 / 4900) loss: 1.546186
(Iteration 2401 / 4900) loss: 1.527419
```

```
(Iteration 2411 / 4900) loss: 1.533881
(Iteration 2421 / 4900) loss: 1.497331
(Iteration 2431 / 4900) loss: 1.557125
(Iteration 2441 / 4900) loss: 1.722981
(Epoch 5 / 10) train acc: 0.505000; val_acc: 0.478000
(Iteration 2451 / 4900) loss: 1.692364
(Iteration 2461 / 4900) loss: 1.520971
(Iteration 2471 / 4900) loss: 1.668541
(Iteration 2481 / 4900) loss: 1.753210
(Iteration 2491 / 4900) loss: 1.738496
(Iteration 2501 / 4900) loss: 1.459870
(Iteration 2511 / 4900) loss: 1.563438
(Iteration 2521 / 4900) loss: 1.550029
(Iteration 2531 / 4900) loss: 1.479471
(Iteration 2541 / 4900) loss: 1.679894
(Iteration 2551 / 4900) loss: 1.507413
(Iteration 2561 / 4900) loss: 1.475613
(Iteration 2571 / 4900) loss: 1.553115
(Iteration 2581 / 4900) loss: 1.579025
(Iteration 2591 / 4900) loss: 1.354438
(Iteration 2601 / 4900) loss: 1.429018
(Iteration 2611 / 4900) loss: 1.589929
(Iteration 2621 / 4900) loss: 1.450085
(Iteration 2631 / 4900) loss: 1.621613
(Iteration 2641 / 4900) loss: 1.503358
(Iteration 2651 / 4900) loss: 1.630748
(Iteration 2661 / 4900) loss: 1.582536
(Iteration 2671 / 4900) loss: 1.684127
(Iteration 2681 / 4900) loss: 1.437425
(Iteration 2691 / 4900) loss: 1.637508
(Iteration 2701 / 4900) loss: 1.671868
(Iteration 2711 / 4900) loss: 1.882296
(Iteration 2721 / 4900) loss: 1.615215
(Iteration 2731 / 4900) loss: 1.666302
(Iteration 2741 / 4900) loss: 1.746458
(Iteration 2751 / 4900) loss: 1.476668
(Iteration 2761 / 4900) loss: 1.724181
(Iteration 2771 / 4900) loss: 1.639708
(Iteration 2781 / 4900) loss: 1.587104
(Iteration 2791 / 4900) loss: 1.513335
(Iteration 2801 / 4900) loss: 1.525152
(Iteration 2811 / 4900) loss: 1.492566
(Iteration 2821 / 4900) loss: 1.608241
(Iteration 2831 / 4900) loss: 1.515757
(Iteration 2841 / 4900) loss: 1.611670
(Iteration 2851 / 4900) loss: 1.603583
(Iteration 2861 / 4900) loss: 1.444740
(Iteration 2871 / 4900) loss: 1.537249
```

```
(Iteration 2881 / 4900) loss: 1.509741
(Iteration 2891 / 4900) loss: 1.580896
(Iteration 2901 / 4900) loss: 1.658380
(Iteration 2911 / 4900) loss: 1.575081
(Iteration 2921 / 4900) loss: 1.416225
(Iteration 2931 / 4900) loss: 1.506875
(Epoch 6 / 10) train acc: 0.534000; val_acc: 0.489000
(Iteration 2941 / 4900) loss: 1.456504
(Iteration 2951 / 4900) loss: 1.587616
(Iteration 2961 / 4900) loss: 1.519747
(Iteration 2971 / 4900) loss: 1.548557
(Iteration 2981 / 4900) loss: 1.479010
(Iteration 2991 / 4900) loss: 1.485778
(Iteration 3001 / 4900) loss: 1.568601
(Iteration 3011 / 4900) loss: 1.690244
(Iteration 3021 / 4900) loss: 1.626242
(Iteration 3031 / 4900) loss: 1.728441
(Iteration 3041 / 4900) loss: 1.514471
(Iteration 3051 / 4900) loss: 1.400100
(Iteration 3061 / 4900) loss: 1.537162
(Iteration 3071 / 4900) loss: 1.626224
(Iteration 3081 / 4900) loss: 1.636041
(Iteration 3091 / 4900) loss: 1.612020
(Iteration 3101 / 4900) loss: 1.614256
(Iteration 3111 / 4900) loss: 1.482318
(Iteration 3121 / 4900) loss: 1.634157
(Iteration 3131 / 4900) loss: 1.478872
(Iteration 3141 / 4900) loss: 1.605981
(Iteration 3151 / 4900) loss: 1.474352
(Iteration 3161 / 4900) loss: 1.689889
(Iteration 3171 / 4900) loss: 1.776640
(Iteration 3181 / 4900) loss: 1.448703
(Iteration 3191 / 4900) loss: 1.518240
(Iteration 3201 / 4900) loss: 1.604032
(Iteration 3211 / 4900) loss: 1.572033
(Iteration 3221 / 4900) loss: 1.587146
(Iteration 3231 / 4900) loss: 1.536611
(Iteration 3241 / 4900) loss: 1.709755
(Iteration 3251 / 4900) loss: 1.514085
(Iteration 3261 / 4900) loss: 1.558934
(Iteration 3271 / 4900) loss: 1.559300
(Iteration 3281 / 4900) loss: 1.546304
(Iteration 3291 / 4900) loss: 1.594386
(Iteration 3301 / 4900) loss: 1.392433
(Iteration 3311 / 4900) loss: 1.564327
(Iteration 3321 / 4900) loss: 1.588974
(Iteration 3331 / 4900) loss: 1.667132
(Iteration 3341 / 4900) loss: 1.712863
```

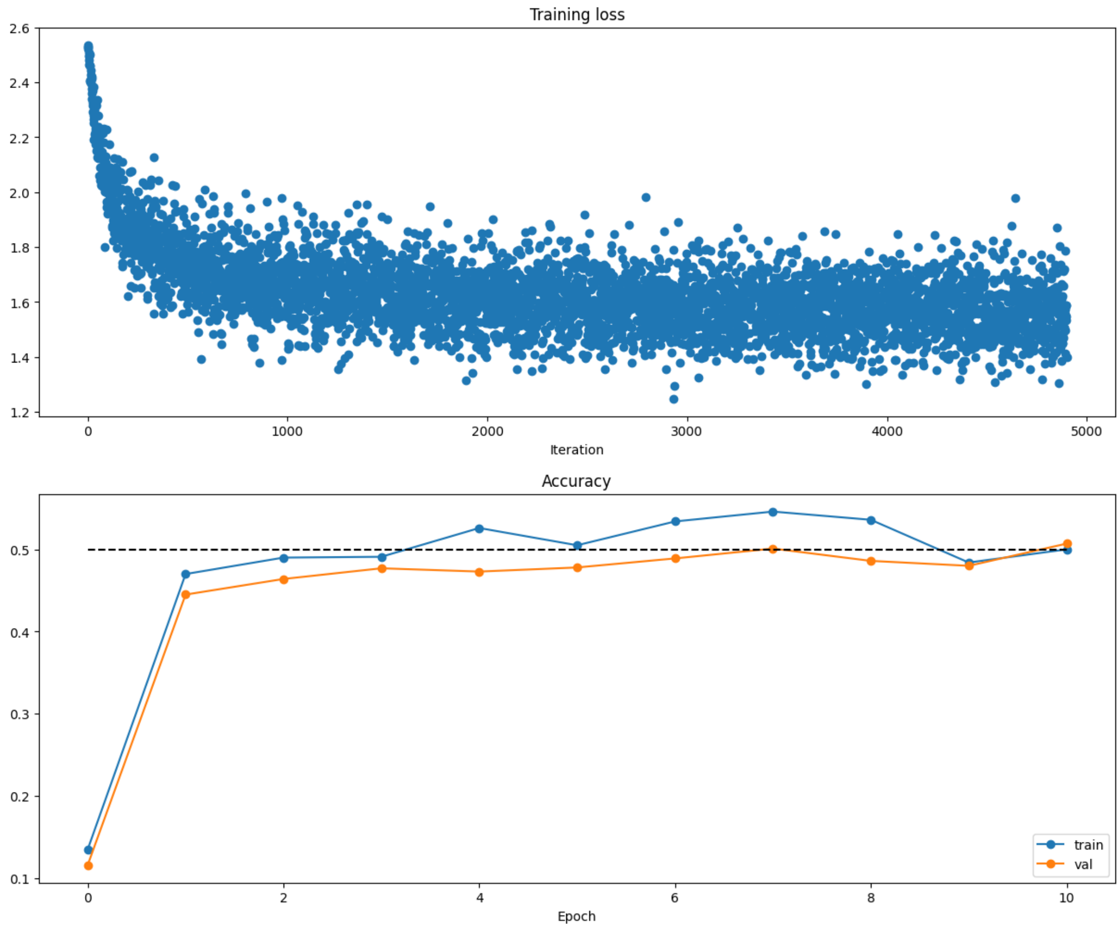
```
(Iteration 3351 / 4900) loss: 1.637820
(Iteration 3361 / 4900) loss: 1.504435
(Iteration 3371 / 4900) loss: 1.476990
(Iteration 3381 / 4900) loss: 1.632405
(Iteration 3391 / 4900) loss: 1.474492
(Iteration 3401 / 4900) loss: 1.724965
(Iteration 3411 / 4900) loss: 1.496573
(Iteration 3421 / 4900) loss: 1.498383
(Epoch 7 / 10) train acc: 0.546000; val_acc: 0.501000
(Iteration 3431 / 4900) loss: 1.597321
(Iteration 3441 / 4900) loss: 1.548957
(Iteration 3451 / 4900) loss: 1.444424
(Iteration 3461 / 4900) loss: 1.647912
(Iteration 3471 / 4900) loss: 1.779134
(Iteration 3481 / 4900) loss: 1.578546
(Iteration 3491 / 4900) loss: 1.458954
(Iteration 3501 / 4900) loss: 1.529832
(Iteration 3511 / 4900) loss: 1.533319
(Iteration 3521 / 4900) loss: 1.420105
(Iteration 3531 / 4900) loss: 1.503119
(Iteration 3541 / 4900) loss: 1.487279
(Iteration 3551 / 4900) loss: 1.523899
(Iteration 3561 / 4900) loss: 1.561601
(Iteration 3571 / 4900) loss: 1.575320
(Iteration 3581 / 4900) loss: 1.669551
(Iteration 3591 / 4900) loss: 1.611775
(Iteration 3601 / 4900) loss: 1.601485
(Iteration 3611 / 4900) loss: 1.661940
(Iteration 3621 / 4900) loss: 1.596142
(Iteration 3631 / 4900) loss: 1.621545
(Iteration 3641 / 4900) loss: 1.740501
(Iteration 3651 / 4900) loss: 1.703060
(Iteration 3661 / 4900) loss: 1.664913
(Iteration 3671 / 4900) loss: 1.509203
(Iteration 3681 / 4900) loss: 1.630026
(Iteration 3691 / 4900) loss: 1.742793
(Iteration 3701 / 4900) loss: 1.536853
(Iteration 3711 / 4900) loss: 1.376293
(Iteration 3721 / 4900) loss: 1.712431
(Iteration 3731 / 4900) loss: 1.413057
(Iteration 3741 / 4900) loss: 1.639629
(Iteration 3751 / 4900) loss: 1.726473
(Iteration 3761 / 4900) loss: 1.609599
(Iteration 3771 / 4900) loss: 1.706229
(Iteration 3781 / 4900) loss: 1.576271
(Iteration 3791 / 4900) loss: 1.627914
(Iteration 3801 / 4900) loss: 1.506361
(Iteration 3811 / 4900) loss: 1.450497
```

```
(Iteration 3821 / 4900) loss: 1.723642
(Iteration 3831 / 4900) loss: 1.725479
(Iteration 3841 / 4900) loss: 1.486949
(Iteration 3851 / 4900) loss: 1.550626
(Iteration 3861 / 4900) loss: 1.463073
(Iteration 3871 / 4900) loss: 1.384863
(Iteration 3881 / 4900) loss: 1.730614
(Iteration 3891 / 4900) loss: 1.558369
(Iteration 3901 / 4900) loss: 1.631694
(Iteration 3911 / 4900) loss: 1.652856
(Epoch 8 / 10) train acc: 0.536000; val_acc: 0.486000
(Iteration 3921 / 4900) loss: 1.524536
(Iteration 3931 / 4900) loss: 1.621412
(Iteration 3941 / 4900) loss: 1.655419
(Iteration 3951 / 4900) loss: 1.518247
(Iteration 3961 / 4900) loss: 1.563778
(Iteration 3971 / 4900) loss: 1.513611
(Iteration 3981 / 4900) loss: 1.355777
(Iteration 3991 / 4900) loss: 1.565154
(Iteration 4001 / 4900) loss: 1.585191
(Iteration 4011 / 4900) loss: 1.561144
(Iteration 4021 / 4900) loss: 1.599865
(Iteration 4031 / 4900) loss: 1.634146
(Iteration 4041 / 4900) loss: 1.639942
(Iteration 4051 / 4900) loss: 1.544032
(Iteration 4061 / 4900) loss: 1.633111
(Iteration 4071 / 4900) loss: 1.591156
(Iteration 4081 / 4900) loss: 1.608551
(Iteration 4091 / 4900) loss: 1.473646
(Iteration 4101 / 4900) loss: 1.678195
(Iteration 4111 / 4900) loss: 1.646820
(Iteration 4121 / 4900) loss: 1.435872
(Iteration 4131 / 4900) loss: 1.657644
(Iteration 4141 / 4900) loss: 1.577040
(Iteration 4151 / 4900) loss: 1.667659
(Iteration 4161 / 4900) loss: 1.561179
(Iteration 4171 / 4900) loss: 1.518486
(Iteration 4181 / 4900) loss: 1.521008
(Iteration 4191 / 4900) loss: 1.452559
(Iteration 4201 / 4900) loss: 1.453106
(Iteration 4211 / 4900) loss: 1.572655
(Iteration 4221 / 4900) loss: 1.717733
(Iteration 4231 / 4900) loss: 1.530545
(Iteration 4241 / 4900) loss: 1.466916
(Iteration 4251 / 4900) loss: 1.558621
(Iteration 4261 / 4900) loss: 1.517706
(Iteration 4271 / 4900) loss: 1.537455
(Iteration 4281 / 4900) loss: 1.440777
```

```
(Iteration 4291 / 4900) loss: 1.403617
(Iteration 4301 / 4900) loss: 1.595973
(Iteration 4311 / 4900) loss: 1.541750
(Iteration 4321 / 4900) loss: 1.610910
(Iteration 4331 / 4900) loss: 1.587097
(Iteration 4341 / 4900) loss: 1.712866
(Iteration 4351 / 4900) loss: 1.507407
(Iteration 4361 / 4900) loss: 1.456759
(Iteration 4371 / 4900) loss: 1.770508
(Iteration 4381 / 4900) loss: 1.385092
(Iteration 4391 / 4900) loss: 1.591654
(Iteration 4401 / 4900) loss: 1.449761
(Epoch 9 / 10) train acc: 0.484000; val_acc: 0.480000
(Iteration 4411 / 4900) loss: 1.511178
(Iteration 4421 / 4900) loss: 1.570689
(Iteration 4431 / 4900) loss: 1.447864
(Iteration 4441 / 4900) loss: 1.457875
(Iteration 4451 / 4900) loss: 1.488508
(Iteration 4461 / 4900) loss: 1.638895
(Iteration 4471 / 4900) loss: 1.488004
(Iteration 4481 / 4900) loss: 1.557544
(Iteration 4491 / 4900) loss: 1.498989
(Iteration 4501 / 4900) loss: 1.583330
(Iteration 4511 / 4900) loss: 1.476706
(Iteration 4521 / 4900) loss: 1.351307
(Iteration 4531 / 4900) loss: 1.538418
(Iteration 4541 / 4900) loss: 1.570777
(Iteration 4551 / 4900) loss: 1.570620
(Iteration 4561 / 4900) loss: 1.564196
(Iteration 4571 / 4900) loss: 1.573398
(Iteration 4581 / 4900) loss: 1.458831
(Iteration 4591 / 4900) loss: 1.498079
(Iteration 4601 / 4900) loss: 1.476942
(Iteration 4611 / 4900) loss: 1.622115
(Iteration 4621 / 4900) loss: 1.476614
(Iteration 4631 / 4900) loss: 1.483931
(Iteration 4641 / 4900) loss: 1.531083
(Iteration 4651 / 4900) loss: 1.508887
(Iteration 4661 / 4900) loss: 1.375934
(Iteration 4671 / 4900) loss: 1.595179
(Iteration 4681 / 4900) loss: 1.409205
(Iteration 4691 / 4900) loss: 1.492501
(Iteration 4701 / 4900) loss: 1.606100
(Iteration 4711 / 4900) loss: 1.413423
(Iteration 4721 / 4900) loss: 1.452241
(Iteration 4731 / 4900) loss: 1.567338
(Iteration 4741 / 4900) loss: 1.512792
(Iteration 4751 / 4900) loss: 1.536257
```

```
(Iteration 4761 / 4900) loss: 1.505830
(Iteration 4771 / 4900) loss: 1.555878
(Iteration 4781 / 4900) loss: 1.480157
(Iteration 4791 / 4900) loss: 1.413923
(Iteration 4801 / 4900) loss: 1.589703
(Iteration 4811 / 4900) loss: 1.504176
(Iteration 4821 / 4900) loss: 1.470949
(Iteration 4831 / 4900) loss: 1.629794
(Iteration 4841 / 4900) loss: 1.735203
(Iteration 4851 / 4900) loss: 1.871321
(Iteration 4861 / 4900) loss: 1.525394
(Iteration 4871 / 4900) loss: 1.578540
(Iteration 4881 / 4900) loss: 1.622767
(Iteration 4891 / 4900) loss: 1.575791
(Epoch 10 / 10) train acc: 0.500000; val_acc: 0.507000
```

```
rate:  0.001 decay:  0.9 size:  100 reg:  1.5 acc:  0.524
```



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.

[49]:
```
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:  0.507
```

[50]:
```
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:  0.504
```

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.

$\color{blue}{\textit{Your Answer:}}$ 1, 3

$\color{blue}{\textit{Your Explanation:}}$ 1) Training on a larger dataset would reduce the impact of outlier datapoints which distort the weight of the system by not reflecting the general behaviour of the data. Provided that the new larger dataset has a diverse group of new samples, such extremities would be "smooothed out" by its use.

3) Increasing the regularization strength prevents overfitting which would cause the system to learn correlations specific to its training set and aren't well generizable.

[ ]:

```

```

features

June 29, 2023

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

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

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

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

```
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.

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


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

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

1.1 Load data

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

[4]:
```
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.

[5]:
```
from cs231n.features import *

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

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

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

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

```
Done extracting features for 1000 / 49000 images
Done extracting features for 2000 / 49000 images
Done extracting features for 3000 / 49000 images
Done extracting features for 4000 / 49000 images
Done extracting features for 5000 / 49000 images
Done extracting features for 6000 / 49000 images
Done extracting features for 7000 / 49000 images
Done extracting features for 8000 / 49000 images
Done extracting features for 9000 / 49000 images
Done extracting features for 10000 / 49000 images
Done extracting features for 11000 / 49000 images
Done extracting features for 12000 / 49000 images
Done extracting features for 13000 / 49000 images
Done extracting features for 14000 / 49000 images
Done extracting features for 15000 / 49000 images
Done extracting features for 16000 / 49000 images
Done extracting features for 17000 / 49000 images
Done extracting features for 18000 / 49000 images
Done extracting features for 19000 / 49000 images
Done extracting features for 20000 / 49000 images
Done extracting features for 21000 / 49000 images
Done extracting features for 22000 / 49000 images
Done extracting features for 23000 / 49000 images
Done extracting features for 24000 / 49000 images
Done extracting features for 25000 / 49000 images
Done extracting features for 26000 / 49000 images
Done extracting features for 27000 / 49000 images
Done extracting features for 28000 / 49000 images
Done extracting features for 29000 / 49000 images
Done extracting features for 30000 / 49000 images
Done extracting features for 31000 / 49000 images
Done extracting features for 32000 / 49000 images
Done extracting features for 33000 / 49000 images
Done extracting features for 34000 / 49000 images
Done extracting features for 35000 / 49000 images
Done extracting features for 36000 / 49000 images
Done extracting features for 37000 / 49000 images
Done extracting features for 38000 / 49000 images
Done extracting features for 39000 / 49000 images
Done extracting features for 40000 / 49000 images
Done extracting features for 41000 / 49000 images
Done extracting features for 42000 / 49000 images
Done extracting features for 43000 / 49000 images
Done extracting features for 44000 / 49000 images
Done extracting features for 45000 / 49000 images
Done extracting features for 46000 / 49000 images
Done extracting features for 47000 / 49000 images
Done extracting features for 48000 / 49000 images
```

```
Done extracting features for 49000 / 49000 images
```

1.3 Train SVM on features

Using the multiclass SVM code developed earlier in the assignment, train SVMs on top of the features extracted above; this should achieve better results than training SVMs directly on top of raw pixels.

[6]:
```python
# Use the validation set to tune the learning rate and regularization strength

from cs231n.classifiers.linear_classifier import LinearSVM

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

results = {}
best_val = -1
best_svm = None

################################################################################
# TODO:                                                                        #
# Use the validation set to set the learning rate and regularization strength. #
# This should be identical to the validation that you did for the SVM; save    #
# the best trained classifer in best_svm. You might also want to play          #
# with different numbers of bins in the color histogram. If you are careful    #
# you should be able to get accuracy of near 0.44 on the validation set.       #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
for lr in learning_rates:
  for reg_str in regularization_strengths:
    svm = LinearSVM()
    svm.train(X_train_feats, y_train, learning_rate=lr, reg=reg_str,
                      num_iters=1500, verbose=False)

    y_pred = svm.predict(X_val_feats)
    validation_accuracy = np.sum(y_pred == y_val) / len(y_val)

    y_pred = svm.predict(X_train_feats)
    training_accuracy = np.sum(y_pred == y_train) / len(y_train)
    results[(lr, reg_str)] = (training_accuracy, validation_accuracy)

    if validation_accuracy > best_val:
      best_val = validation_accuracy
      best_svm = svm
pass

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

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

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

```
lr 1.000000e-09 reg 5.000000e+04 train accuracy: 0.090286 val accuracy: 0.097000
lr 1.000000e-09 reg 5.000000e+05 train accuracy: 0.106306 val accuracy: 0.095000
lr 1.000000e-09 reg 5.000000e+06 train accuracy: 0.411245 val accuracy: 0.409000
lr 1.000000e-08 reg 5.000000e+04 train accuracy: 0.099224 val accuracy: 0.090000
lr 1.000000e-08 reg 5.000000e+05 train accuracy: 0.414429 val accuracy: 0.422000
lr 1.000000e-08 reg 5.000000e+06 train accuracy: 0.406082 val accuracy: 0.402000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.413735 val accuracy: 0.414000
lr 1.000000e-07 reg 5.000000e+05 train accuracy: 0.402000 val accuracy: 0.413000
lr 1.000000e-07 reg 5.000000e+06 train accuracy: 0.315102 val accuracy: 0.299000
best validation accuracy achieved: 0.422000
```

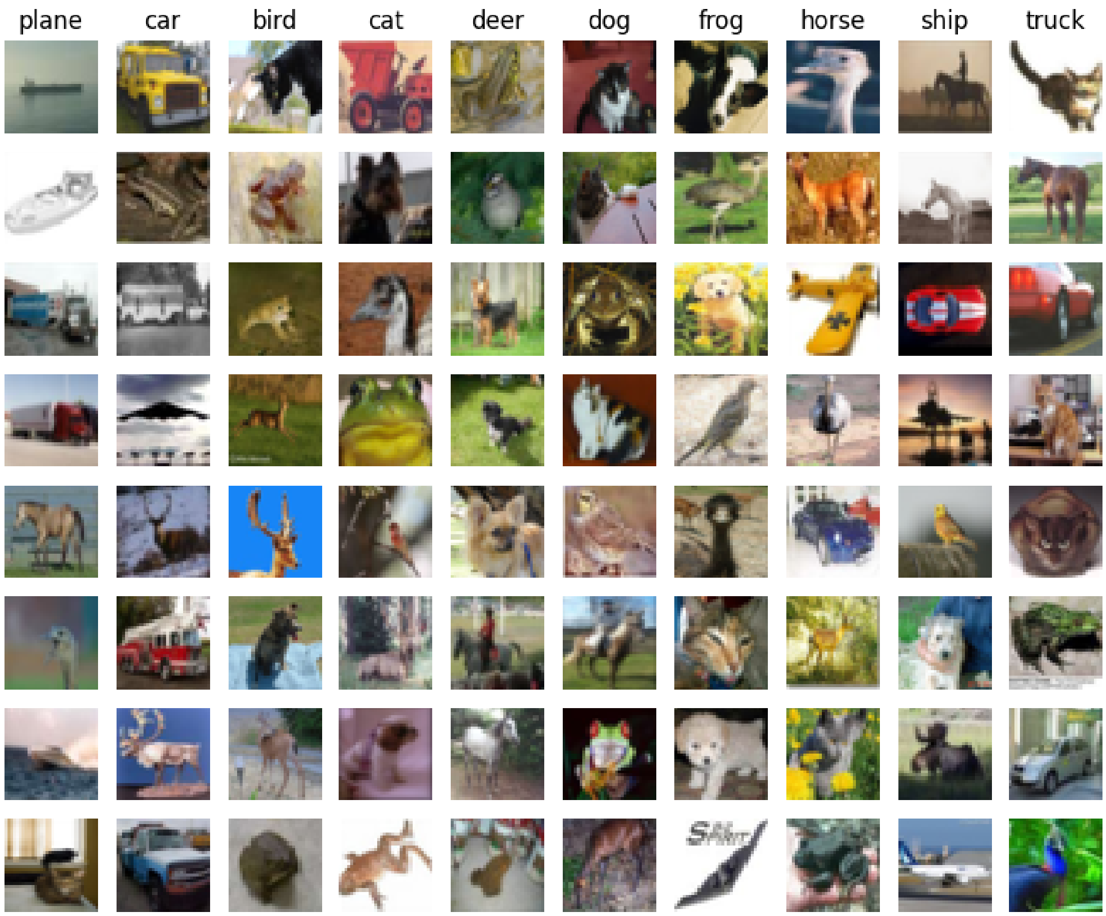
[7]:

```
# 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)
```

```
0.421
```

[8]:

```
# 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.

[9]:
```
# 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)
```

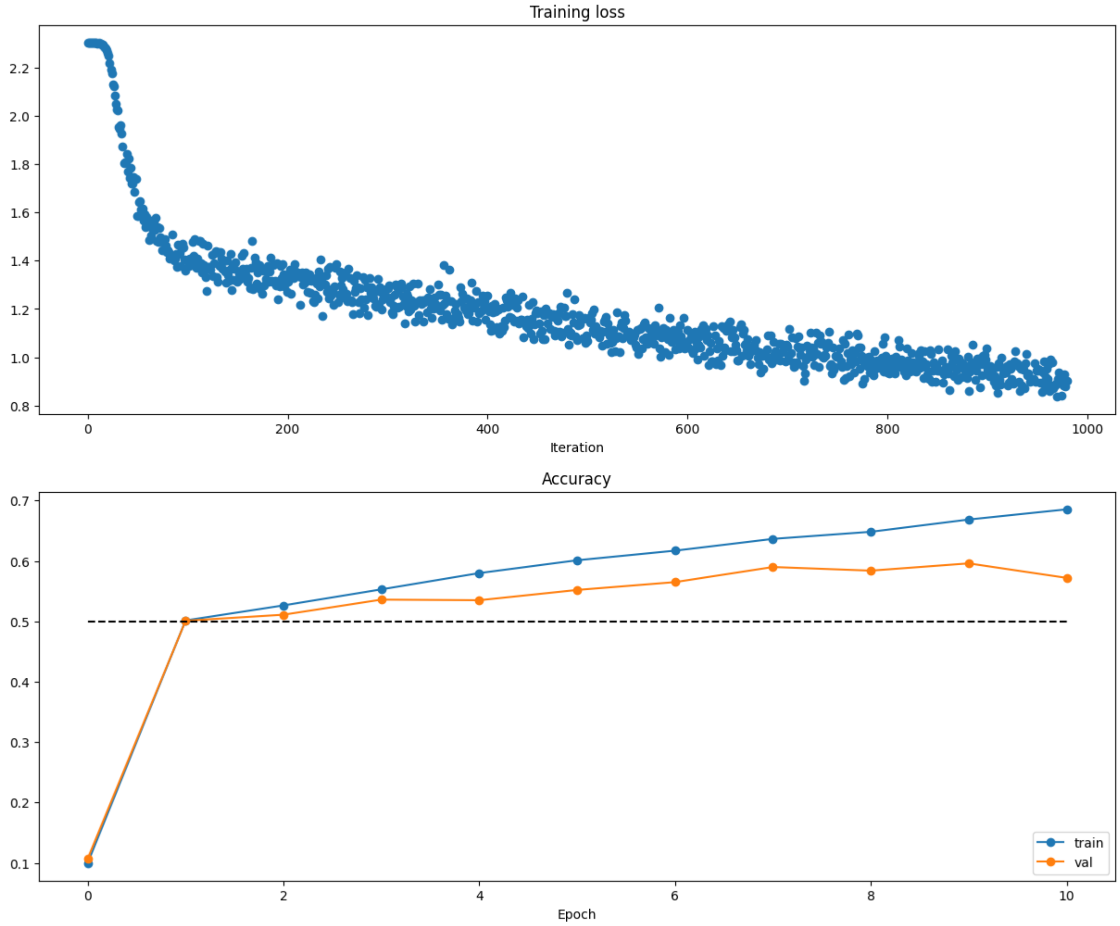
```
(49000, 155)
(49000, 154)
```

[6]:

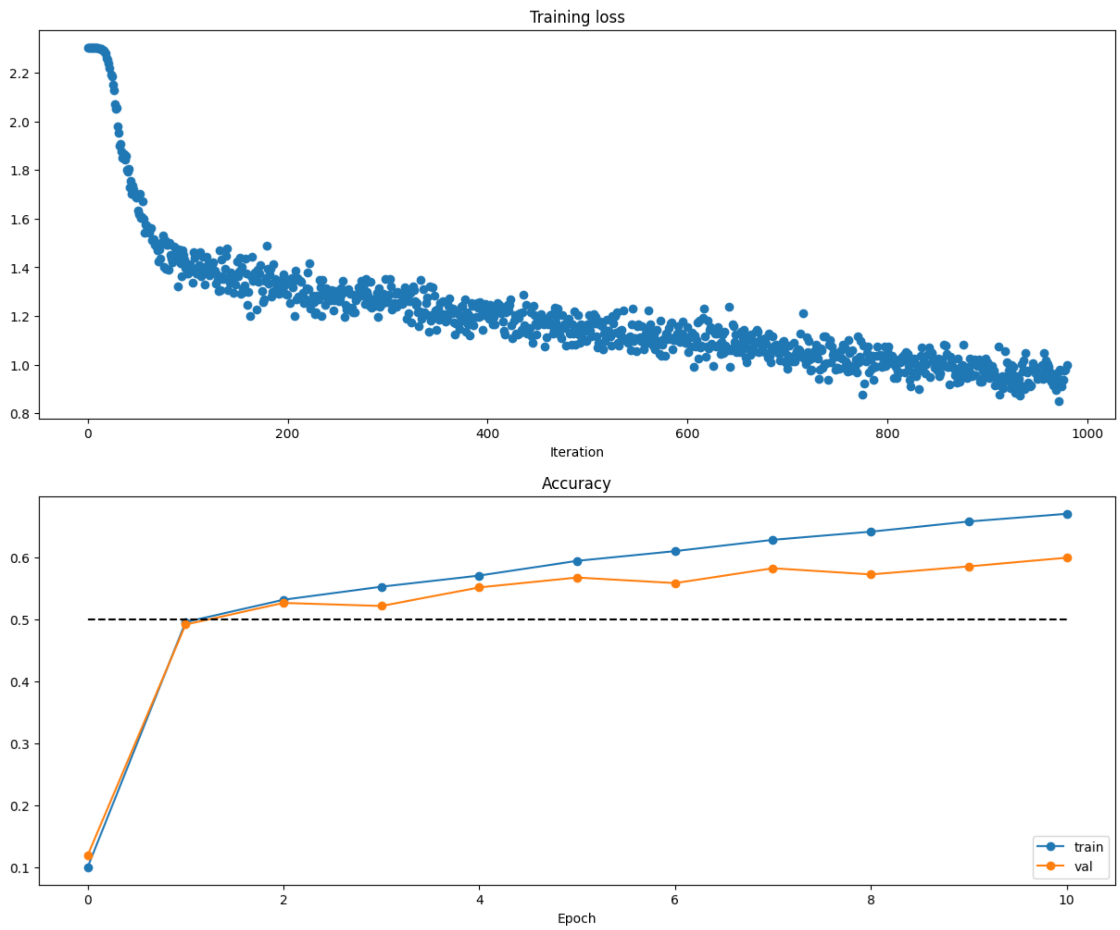
```
from cs231n.classifiers.fc_net import TwoLayerNet
from cs231n.solver import Solver

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,
}

learning_rates = [0.5, 1, 2.5]
learning_decays = [1, 0.95, 0.7]
regularization_strengths = [0, 1e-4, 1e-2, 0.5, 1, 2.5]

results = {}
best_val = -1
best_net = None

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

for reg_str in regularization_strengths:
  for lr in learning_rates:
    for decay in learning_decays:
      net = TwoLayerNet(input_dim, hidden_dim, num_classes)
      net.reg = reg_str
      solver = Solver(net, data,
                      update_rule='sgd',
                      optim_config={"learning_rate" : lr},
```

```
                    lr_decay = decay,
                    batch_size = 500,
                    num_train_samples = None,
                    verbose = False,
                    print_every = 500
                    )
      solver.train()
      results[(lr, reg_str, decay)] = (solver.train_acc_history[-1], solver.
↪val_acc_history[-1])
      if results[(lr, reg_str, decay)][1] > best_val:
        best_val = results[(lr, reg_str, decay)][1]
        best_net = net

        print("better config found:")
        print("lr:", lr, "dec:", decay, "reg:", reg_str)
        print("val acc:", results[(lr, reg_str, decay)][1])
        print("train acc:", results[(lr, reg_str, decay)][0])

        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()

print(results)

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

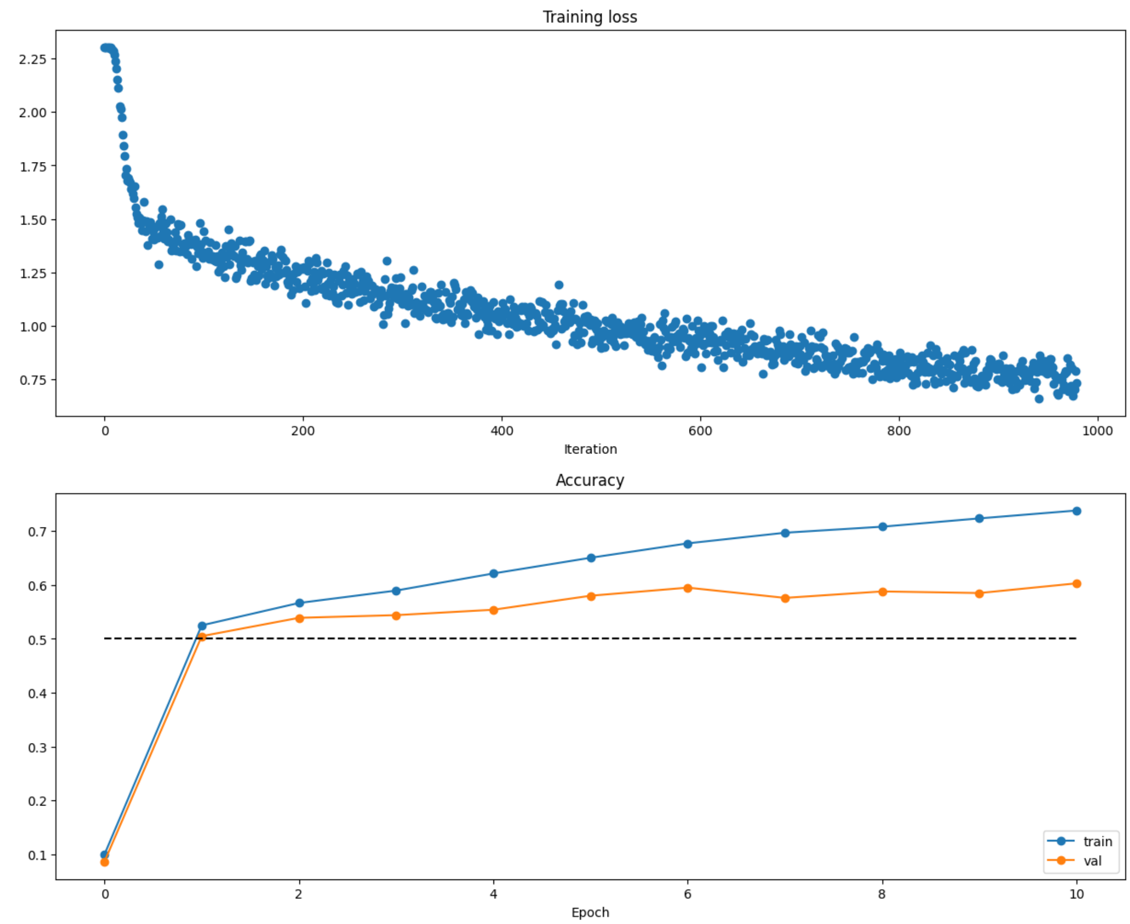
```
better config found:
lr: 0.5 dec: 1 reg: 0
val acc: 0.572
train acc: 0.6855102040816327
```



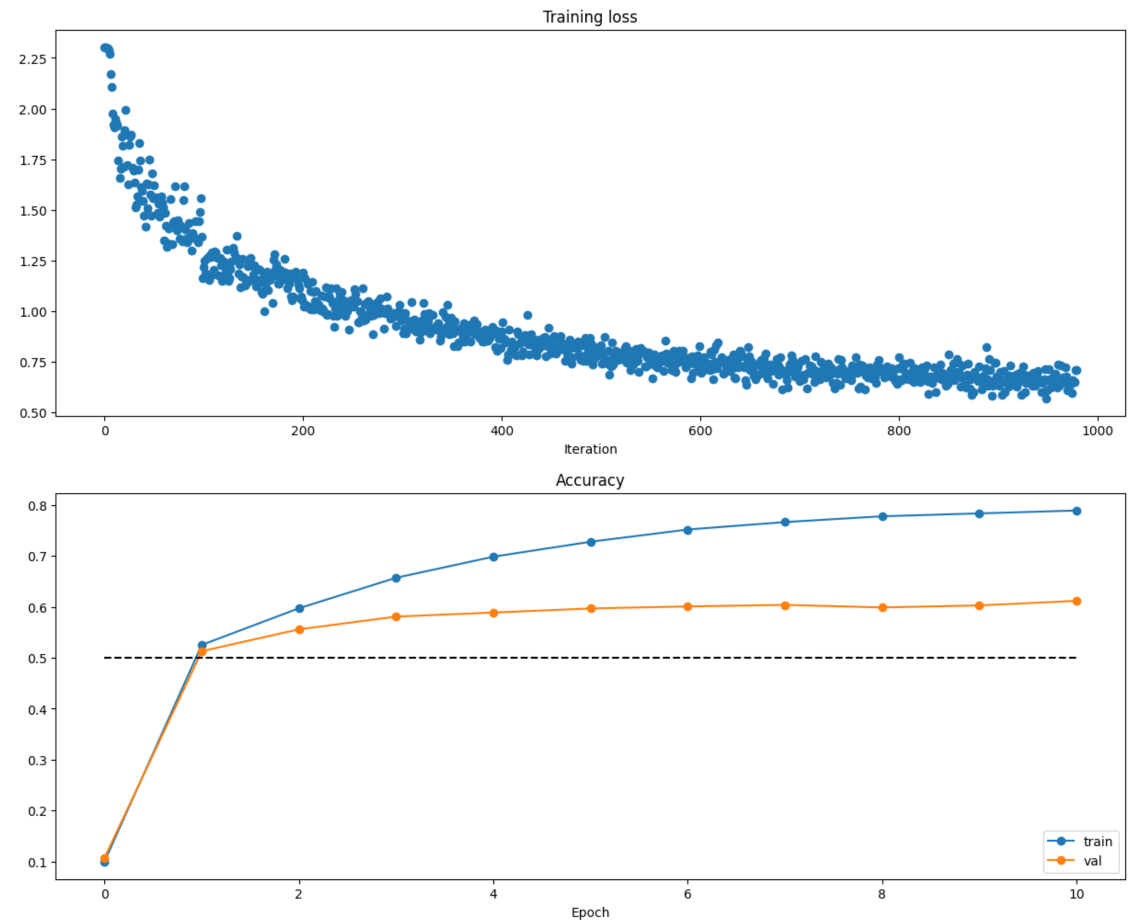

```
better config found:
lr: 0.5 dec: 0.95 reg: 0
val acc: 0.6
train acc: 0.6710204081632654
```




```
better config found:
lr: 1 dec: 0.95 reg: 0
val acc: 0.603
train acc: 0.7381224489795918
```




```
better config found:
lr: 2.5 dec: 0.7 reg: 0
val acc: 0.612
train acc: 0.7894897959183673
```



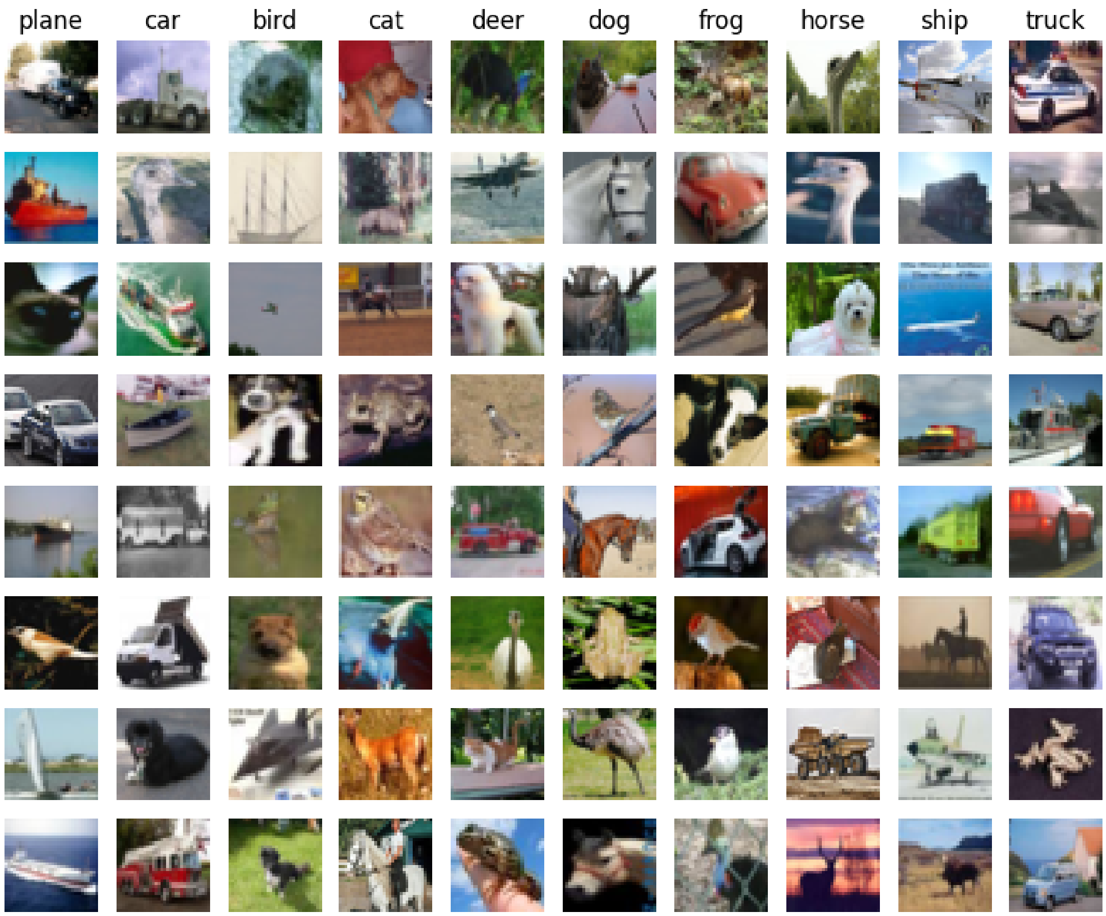

```
/content/drive/My Drive/cs231n/assignments/assignment1/cs231n/layers.py:821:
RuntimeWarning: overflow encountered in exp
  x_exp = np.exp(x)
/content/drive/My Drive/cs231n/assignments/assignment1/cs231n/layers.py:827:
RuntimeWarning: invalid value encountered in true_divide
  dx *= x_exp / total[:,np.newaxis]
```

```
{(0.5, 0, 1): (0.6855102040816327, 0.572), (0.5, 0, 0.95): (0.6710204081632654,
0.6), (0.5, 0, 0.7): (0.5723469387755102, 0.55), (1, 0, 1): (0.7184081632653061,
0.547), (1, 0, 0.95): (0.7381224489795918, 0.603), (1, 0, 0.7):
(0.6521836734693878, 0.595), (2.5, 0, 1): (0.7906326530612245, 0.566), (2.5, 0,
0.95): (0.8277959183673469, 0.575), (2.5, 0, 0.7): (0.7894897959183673, 0.612),
(0.5, 0.0001, 1): (0.6844489795918367, 0.592), (0.5, 0.0001, 0.95):
(0.6710612244897959, 0.6), (0.5, 0.0001, 0.7): (0.5733265306122449, 0.549), (1,
0.0001, 1): (0.7431836734693877, 0.569), (1, 0.0001, 0.95): (0.7434897959183674,
0.573), (1, 0.0001, 0.7): (0.6524897959183673, 0.596), (2.5, 0.0001, 1):
(0.7717142857142857, 0.543), (2.5, 0.0001, 0.95): (0.806469387755102, 0.536),
(2.5, 0.0001, 0.7): (0.7840816326530612, 0.604), (0.5, 0.01, 1):
(0.5694285714285714, 0.539), (0.5, 0.01, 0.95): (0.5672857142857143, 0.538),
```

```
(0.5, 0.01, 0.7): (0.5426326530612244, 0.525), (1, 0.01, 1):
(0.5623061224489796, 0.543), (1, 0.01, 0.95): (0.540530612244898, 0.512), (1,
0.01, 0.7): (0.5678163265306122, 0.56), (2.5, 0.01, 1): (0.47455102040816327,
0.448), (2.5, 0.01, 0.95): (0.5231428571428571, 0.493), (2.5, 0.01, 0.7):
(0.5931836734693877, 0.567), (0.5, 0.5, 1): (0.17795918367346938, 0.166), (0.5,
0.5, 0.95): (0.10010204081632654, 0.098), (0.5, 0.5, 0.7): (0.10042857142857142,
0.079), (1, 0.5, 1): (0.174, 0.157), (1, 0.5, 0.95): (0.15553061224489795,
0.149), (1, 0.5, 0.7): (0.10042857142857142, 0.079), (2.5, 0.5, 1):
(0.09989795918367347, 0.105), (2.5, 0.5, 0.95): (0.10044897959183674, 0.078),
(2.5, 0.5, 0.7): (0.09985714285714285, 0.107), (0.5, 1, 1):
(0.10026530612244898, 0.087), (0.5, 1, 0.95): (0.10004081632653061, 0.098),
(0.5, 1, 0.7): (0.09995918367346938, 0.102), (1, 1, 1): (0.10004081632653061,
0.098), (1, 1, 0.95): (0.09985714285714285, 0.107), (1, 1, 0.7):
(0.10044897959183674, 0.078), (2.5, 1, 1): (0.10026530612244898, 0.087), (2.5,
1, 0.95): (0.10026530612244898, 0.087), (2.5, 1, 0.7): (0.10026530612244898,
0.087), (0.5, 2.5, 1): (0.09985714285714285, 0.107), (0.5, 2.5, 0.95):
(0.10044897959183674, 0.078), (0.5, 2.5, 0.7): (0.10044897959183674, 0.078), (1,
2.5, 1): (0.10026530612244898, 0.087), (1, 2.5, 0.95): (0.10026530612244898,
0.087), (1, 2.5, 0.7): (0.10026530612244898, 0.087), (2.5, 2.5, 1):
(0.10026530612244898, 0.087), (2.5, 2.5, 0.95): (0.10026530612244898, 0.087),
(2.5, 2.5, 0.7): (0.10026530612244898, 0.087)}
```

[8]:

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

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

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()

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




```
0.598
```