knn

August 25, 2025

[5]:
```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive')

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

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

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

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
/content/drive/My Drive/cs231n/assignments/assignment1/cs231n/datasets
--2025-08-25 03:07:49--  http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
Resolving www.cs.toronto.edu (www.cs.toronto.edu)… 128.100.3.30
Connecting to www.cs.toronto.edu (www.cs.toronto.edu)|128.100.3.30|:80…
connected.
HTTP request sent, awaiting response… 200 OK
Length: 170498071 (163M) [application/x-gzip]
Saving to: ‘cifar-10-python.tar.gz’

cifar-10-python.tar 100%[===================>] 162.60M  47.6MB/s    in 3.6s

2025-08-25 03:07:52 (45.6 MB/s) - ‘cifar-10-python.tar.gz’ saved
[170498071/170498071]

cifar-10-batches-py/
```

```
cifar-10-batches-py/data_batch_4
cifar-10-batches-py/readme.html
cifar-10-batches-py/test_batch
cifar-10-batches-py/data_batch_3
cifar-10-batches-py/batches.meta
cifar-10-batches-py/data_batch_2
cifar-10-batches-py/data_batch_5
cifar-10-batches-py/data_batch_1
--2025-08-25 03:07:57--  http://cs231n.stanford.edu/imagenet_val_25.npz
Resolving cs231n.stanford.edu (cs231n.stanford.edu)… 171.64.64.64
Connecting to cs231n.stanford.edu (cs231n.stanford.edu)|171.64.64.64|:80…
connected.
HTTP request sent, awaiting response… 301 Moved Permanently
Location: https://cs231n.stanford.edu/imagenet_val_25.npz [following]
--2025-08-25 03:07:58--  https://cs231n.stanford.edu/imagenet_val_25.npz
Connecting to cs231n.stanford.edu (cs231n.stanford.edu)|171.64.64.64|:443…
connected.
HTTP request sent, awaiting response… 200 OK
Length: 3940548 (3.8M)
Saving to: 'imagenet_val_25.npz'

imagenet_val_25.npz 100%[===================>]   3.76M  6.14MB/s    in 0.6s

2025-08-25 03:07:59 (6.14 MB/s) - 'imagenet_val_25.npz' saved [3940548/3940548]

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

1 k-Nearest Neighbor (kNN) exercise

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

The kNN classifier consists of two stages:

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

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

[9]:
```
# 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
```

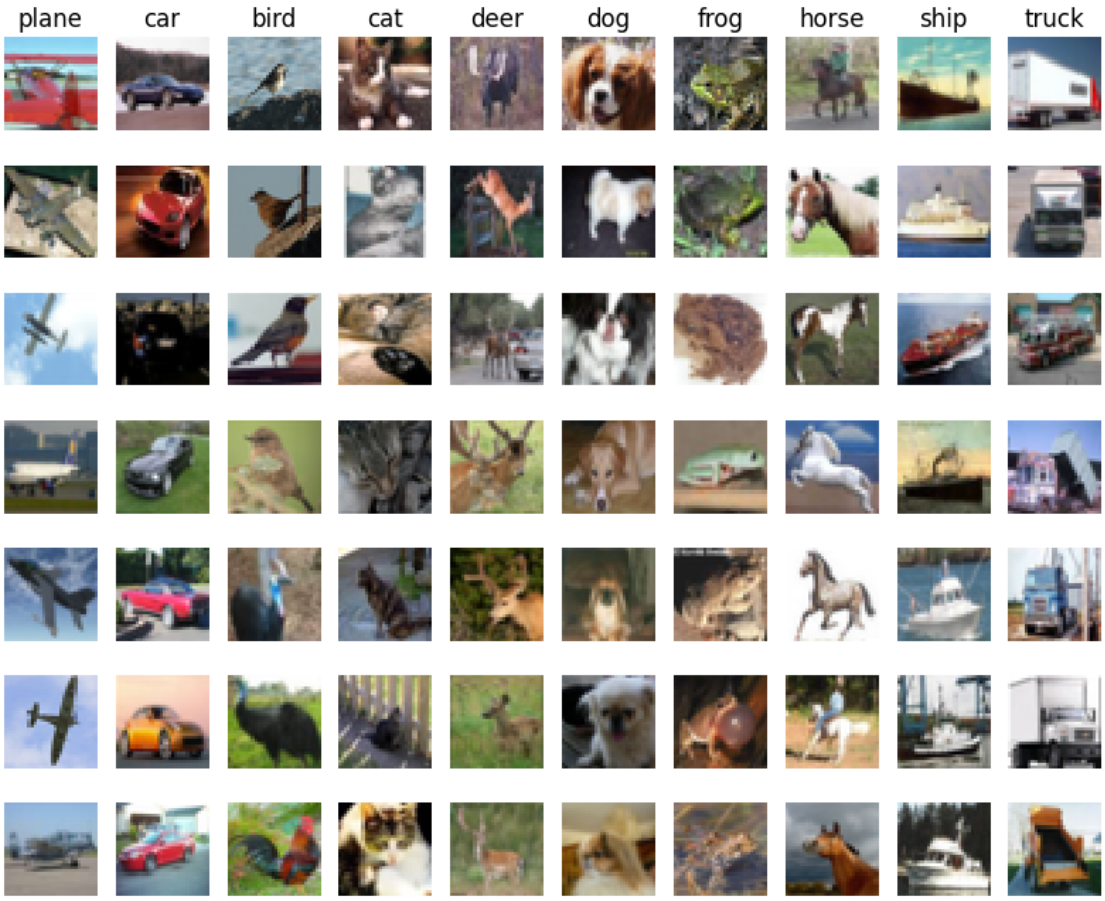
```
import importlib
# This is a bit of magic to make matplotlib figures appear inline in the␣
 ↪notebook
# rather than in a new window.
%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap']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

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

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

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

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

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

[11]:
```
# 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()
```



[12]:
```
# 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)
```

[13]:
```
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.

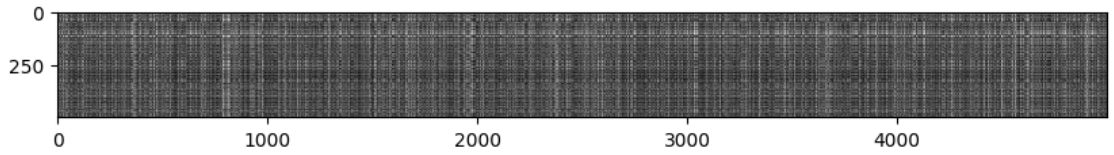
Note: For the three distance computations that we require you to implement in this notebook, you may not use the np.linalg.norm() function that numpy provides.

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.

[14]:
```
# 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)
```

[15]:
```
# 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 : i i j j

[21]:
```
# 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`:

[22]:
```
y_test_pred = classifier.predict_labels(dists, k=5)
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
```

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

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

Inline Question 2

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

the mean μ across all pixels over all images is

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

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

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

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

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

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

YourAnswer :*1,2,5*

YourExplanation :*1/2/5 3/4*

[24]:
```
# 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
```

[25]:
```
# 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
```

[26]:
```
# 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 37.301511 seconds
One loop version took 52.591960 seconds
No loop version took 0.626420 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.

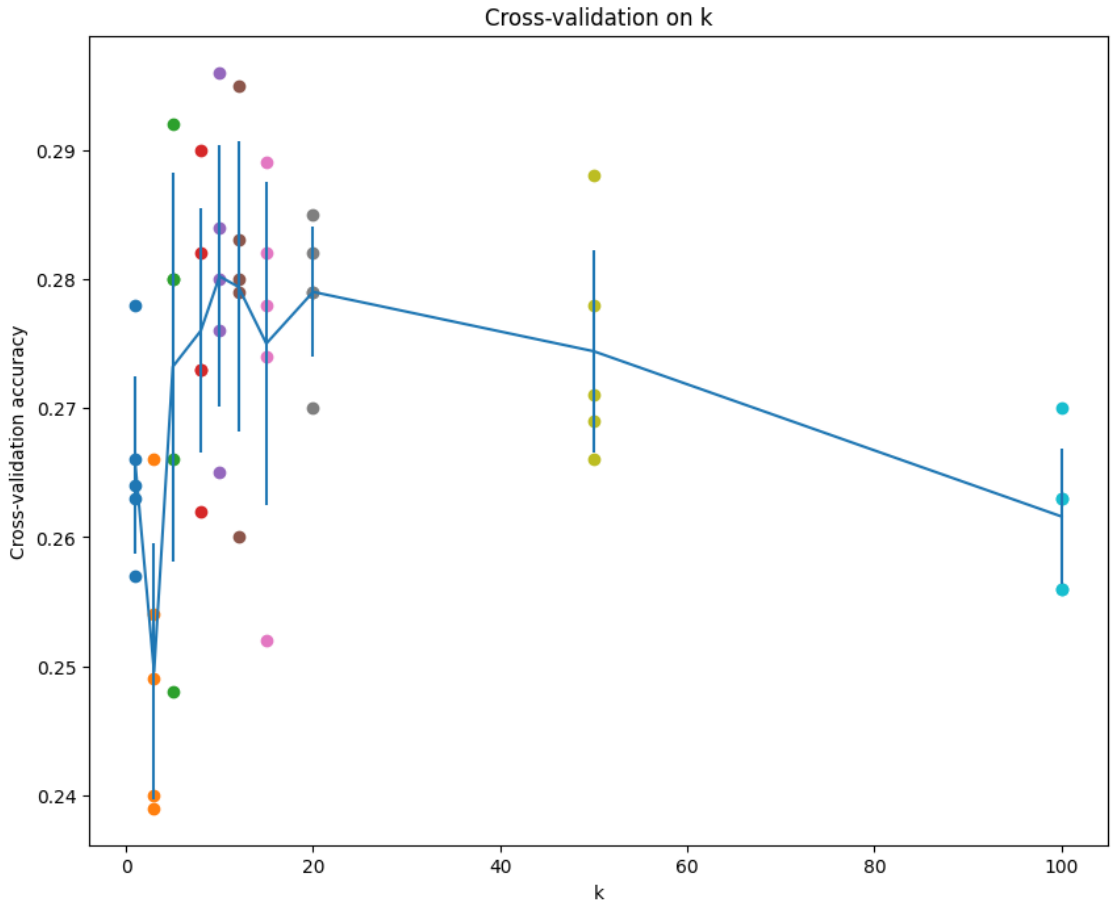
[27]:
```
num_folds = 5
k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50, 100]

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.                                #
################################################################################
X_train_folds = np.array_split(X_train, num_folds)
y_train_folds = np.array_split(y_train, num_folds)
################################################################################
#                                 END OF YOUR CODE                             #
################################################################################

# 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.                               #
################################################################################
for k in k_choices:
    accuracies = []
    for i in range(num_folds):
        X_train_cv = np.vstack(X_train_folds[0:i] + X_train_folds[i+1:])
        y_train_cv = np.hstack(y_train_folds[0:i] + y_train_folds[i+1:])
        X_valid_cv = X_train_folds[i]
        y_valid_cv = y_train_folds[i]

        classifier.train(X_train_cv, y_train_cv)
        dists = classifier.compute_distances_no_loops(X_valid_cv)
        accuracy = float(np.sum(classifier.predict_labels(dists, k) ==␣
 ↪y_valid_cv)) / y_valid_cv.shape[0]
        accuracies.append(accuracy)
    k_to_accuracies[k] = accuracies
```

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

```
k = 1, accuracy = 0.263000
k = 1, accuracy = 0.257000
k = 1, accuracy = 0.264000
k = 1, accuracy = 0.278000
k = 1, accuracy = 0.266000
k = 3, accuracy = 0.239000
k = 3, accuracy = 0.249000
k = 3, accuracy = 0.240000
k = 3, accuracy = 0.266000
k = 3, accuracy = 0.254000
k = 5, accuracy = 0.248000
k = 5, accuracy = 0.266000
k = 5, accuracy = 0.280000
k = 5, accuracy = 0.292000
k = 5, accuracy = 0.280000
k = 8, accuracy = 0.262000
k = 8, accuracy = 0.282000
k = 8, accuracy = 0.273000
k = 8, accuracy = 0.290000
k = 8, accuracy = 0.273000
k = 10, accuracy = 0.265000
k = 10, accuracy = 0.296000
k = 10, accuracy = 0.276000
k = 10, accuracy = 0.284000
k = 10, accuracy = 0.280000
k = 12, accuracy = 0.260000
k = 12, accuracy = 0.295000
k = 12, accuracy = 0.279000
k = 12, accuracy = 0.283000
k = 12, accuracy = 0.280000
k = 15, accuracy = 0.252000
k = 15, accuracy = 0.289000
k = 15, accuracy = 0.278000
k = 15, accuracy = 0.282000
k = 15, accuracy = 0.274000
k = 20, accuracy = 0.270000
k = 20, accuracy = 0.279000
k = 20, accuracy = 0.279000
k = 20, accuracy = 0.282000
k = 20, accuracy = 0.285000
```

```
k = 50, accuracy = 0.271000
k = 50, accuracy = 0.288000
k = 50, accuracy = 0.278000
k = 50, accuracy = 0.269000
k = 50, accuracy = 0.266000
k = 100, accuracy = 0.256000
k = 100, accuracy = 0.270000
k = 100, accuracy = 0.263000
k = 100, accuracy = 0.256000
k = 100, accuracy = 0.263000
```

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



[30]:
```
# Based on the cross-validation results above, choose the best value for k,
# retrain the classifier using all the training data, and test it on the test
# data. You should be able to get above 28% accuracy on the test data.
best_k = 10

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

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

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

Inline Question 3

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

linear. 2. The training error of a 1-NN will always be lower than or equal to that of 5-NN. 3. The test error of a 1-NN will always be lower than that of a 5-NN. 4. The time needed to classify a test example with the k-NN classifier grows with the size of the training set. 5. None of the above.

YourAnswer :*2,4*

YourExplanation :*knn 1nn*

softmax

August 25, 2025

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

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

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

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

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

1 Softmax Classifier 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 Softmax classifier.
- 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

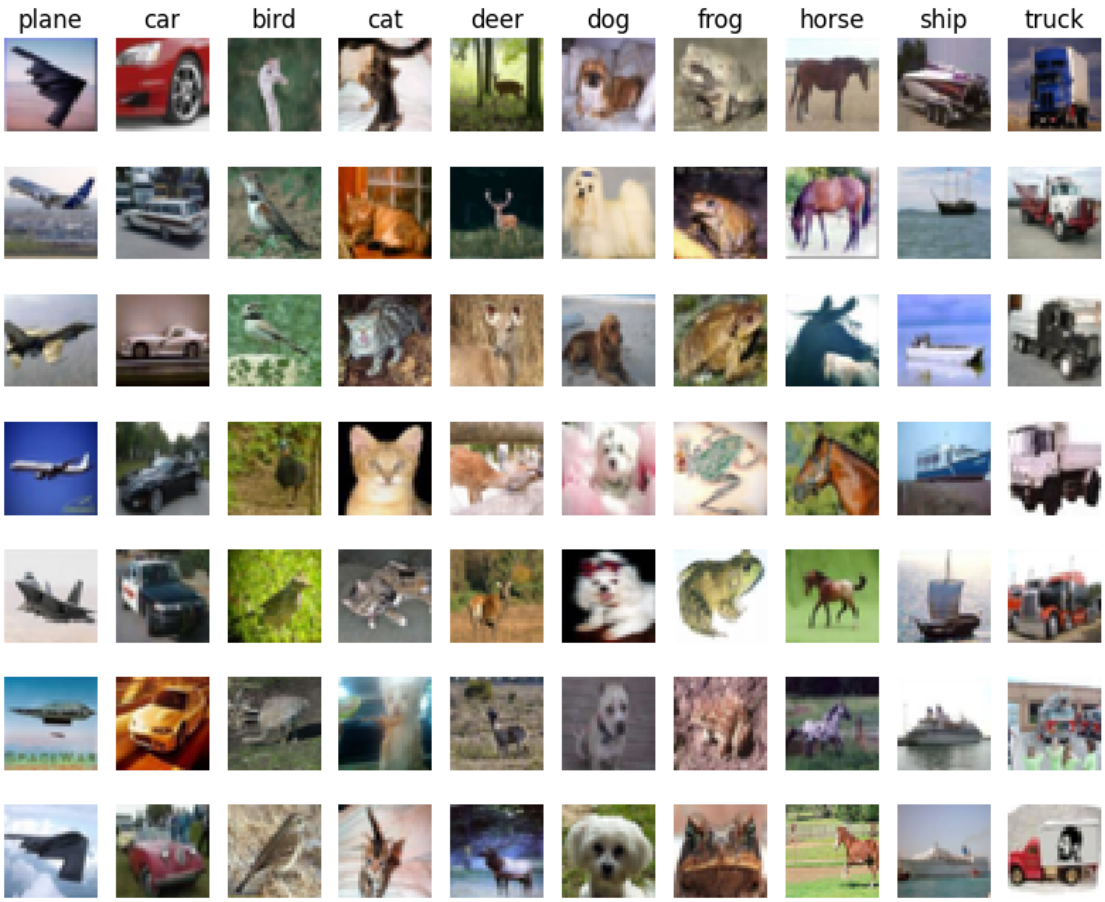
[3]:
```
# Run some setup code for this notebook.
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

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

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

1.1 CIFAR-10 Data Loading and Preprocessing

[4]:
```
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

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

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

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

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

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



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

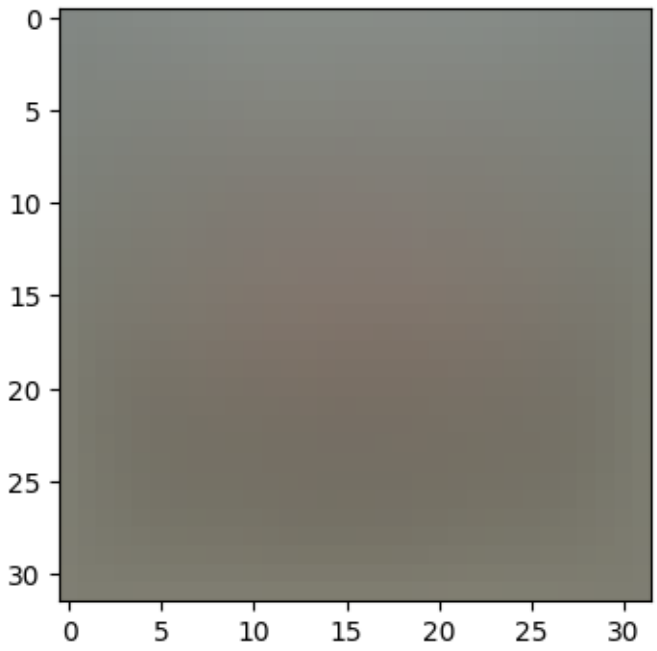
```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

[7]:
```
# 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)
```

[8]:
```
# 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␣
 ↪classifier
# 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 Softmax Classifier

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

As you can see, we have prefilled the function `softmax_loss_naive` which uses for loops to evaluate the softmax loss function.

[9]:
```
# Evaluate the naive implementation of the loss we provided for you:
from cs231n.classifiers.softmax import softmax_loss_naive
import time

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

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

# 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: 2.387084
loss: 2.387084
sanity check: 2.302585
```

Inline Question 1

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

YourAnswer : *10* *-log(0.1)*

The `grad` returned from the function above is right now all zero. Derive and implement the gradient for the softmax loss function and implement it inline inside the function `softmax_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:

[10]:
```
# 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 = softmax_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: softmax_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 = 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)
```

```
numerical: -2.727826 analytic: -2.727826, relative error: 3.006446e-09
numerical: 3.020940 analytic: 3.020940, relative error: 7.823699e-09
numerical: -0.203974 analytic: -0.203974, relative error: 1.382604e-08
numerical: -1.335979 analytic: -1.335979, relative error: 8.001343e-09
numerical: 1.868170 analytic: 1.868170, relative error: 2.525139e-09
numerical: 1.380688 analytic: 1.380688, relative error: 2.404359e-08
numerical: 3.625581 analytic: 3.625581, relative error: 2.403986e-08
numerical: -0.007904 analytic: -0.007904, relative error: 4.326429e-06
numerical: -0.364646 analytic: -0.364646, relative error: 1.061023e-07
numerical: -0.907408 analytic: -0.907408, relative error: 8.827617e-09
numerical: 0.930268 analytic: 0.930268, relative error: 5.081194e-08
numerical: 2.674086 analytic: 2.674086, relative error: 3.522200e-08
numerical: 2.151446 analytic: 2.151446, relative error: 3.625622e-08
numerical: 3.086982 analytic: 3.086982, relative error: 7.122865e-09
numerical: -2.990443 analytic: -2.990443, relative error: 2.438965e-09
numerical: -0.537000 analytic: -0.537000, relative error: 3.901060e-08
```

```
numerical: 3.308303 analytic: 3.308303, relative error: 1.201469e-08
numerical: 0.824303 analytic: 0.824303, relative error: 7.937680e-08
numerical: 2.890657 analytic: 2.890657, relative error: 2.370783e-08
numerical: -0.350152 analytic: -0.350152, relative error: 5.885145e-08
```

Inline Question 2

Although gradcheck is reliable softmax loss, it is possible that for SVM loss, 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 svm loss gradient check could fail? How would change the margin affect of the frequency of this happening?

Note that SVM loss for a sample (x_i, y_i) is defined as:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + \Delta)$$

where j iterates over all classes except the correct class y_i and s_j denotes the classifier score for j^{th} class. Δ is a scalar margin. For more information, refer to 'Multiclass Support Vector Machine loss' on this page.

Hint: the SVM loss function is not strictly speaking differentiable.

YourAnswer : SVM Δ

[11]:
```
# Next implement the function softmax_loss_vectorized; for now only compute the loss;
# we will implement the gradient in a moment.
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, _ = softmax_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: 2.387084e+00 computed in 0.076677s
Vectorized loss: 2.387084e+00 computed in 0.011908s
difference: 0.000000
```

[12]:
```
# Complete the implementation of softmax_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 = softmax_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 = softmax_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 0.081968s
Vectorized loss and gradient: computed in 0.010062s
difference: 0.000000
```

1.2.1 Stochastic Gradient Descent

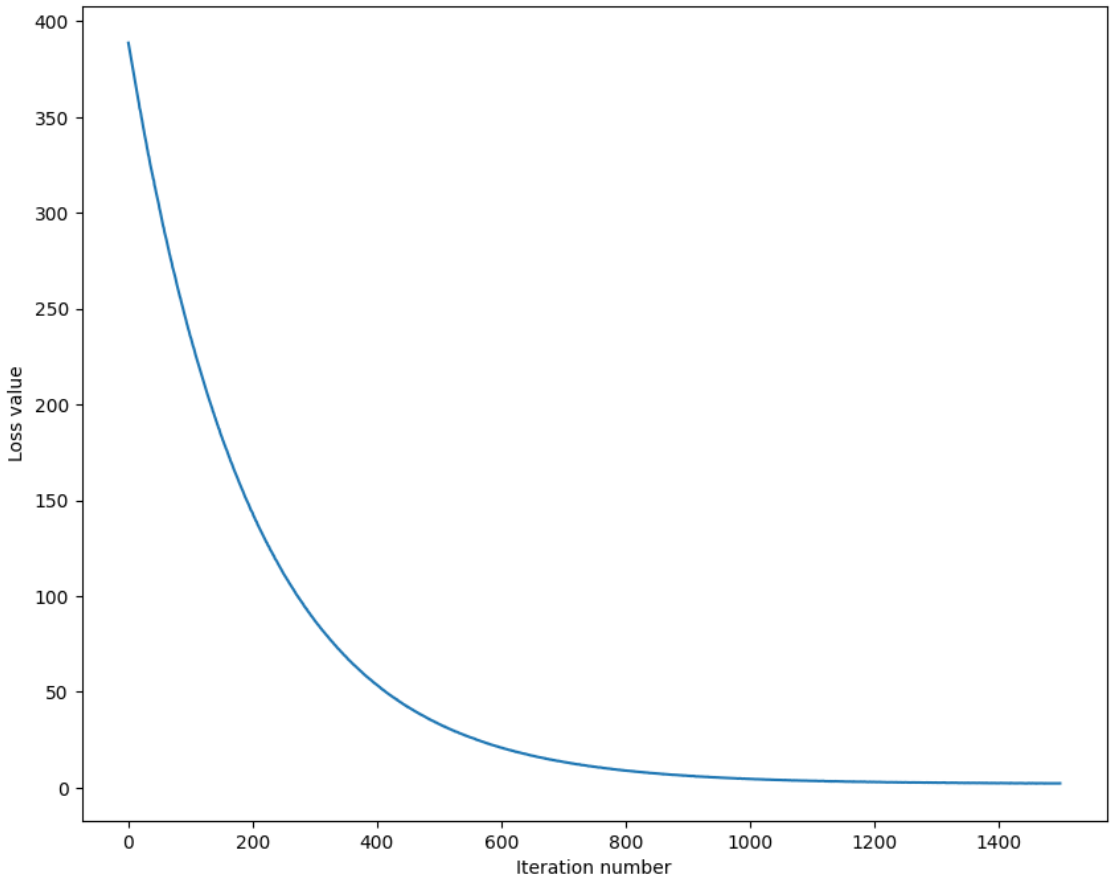
We now have vectorized and efficient expressions for the loss, the gradient and our gradient matches the numerical gradient. We are therefore ready to do SGD to minimize the loss. Your code for this part will be written inside `cs231n/classifiers/linear_classifier.py`.

[13]:
```
# 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 Softmax
softmax = Softmax()
tic = time.time()
loss_hist = softmax.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 388.690061
iteration 100 / 1500: loss 235.248239
iteration 200 / 1500: loss 143.245945
iteration 300 / 1500: loss 87.181282
iteration 400 / 1500: loss 53.639276
iteration 500 / 1500: loss 33.142876
iteration 600 / 1500: loss 20.876610
iteration 700 / 1500: loss 13.396849
iteration 800 / 1500: loss 8.930040
iteration 900 / 1500: loss 6.182943
```

```
iteration 1000 / 1500: loss 4.553215
iteration 1100 / 1500: loss 3.568394
iteration 1200 / 1500: loss 2.970628
iteration 1300 / 1500: loss 2.581539
iteration 1400 / 1500: loss 2.351680
That took 8.301981s
```

[14]:
```
# 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()
```



[15]:
```
# Write the LinearClassifier.predict function and evaluate the performance on
# both the training and validation set
# You should get validation accuracy of about 0.34 (> 0.33).
y_train_pred = softmax.predict(X_train)
print('training accuracy: %f' % (np.mean(y_train == y_train_pred), ))
```

```
y_val_pred = softmax.predict(X_val)
print('validation accuracy: %f' % (np.mean(y_val == y_val_pred), ))
```

```
training accuracy: 0.351347
validation accuracy: 0.368000
```

[16]:
```
# Save the trained model for autograder.
softmax.save("softmax.npy")
```

```
softmax.npy saved.
```

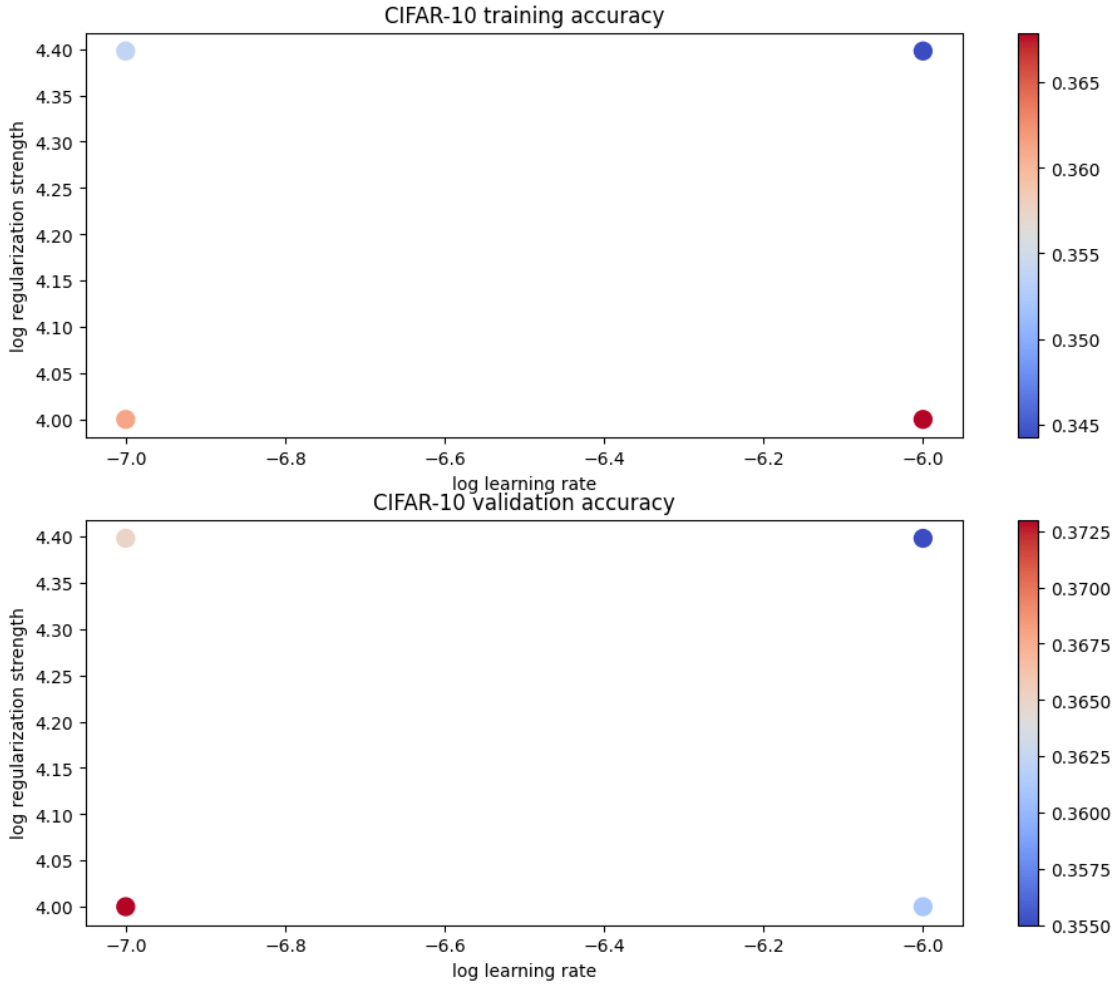
[19]:
```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of about 0.365 (> 0.36) on the validation set.

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

# results is dictionary mapping tuples of the form
# (learning_rate, regularization_strength) to tuples of the form
# (training_accuracy, validation_accuracy). The accuracy is simply the fraction
# of data points that are correctly classified.
results = {}
best_val = -1   # The highest validation accuracy that we have seen so far.
best_softmax = None # The Softmax 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 Softmax 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 Softmax object that achieves this.   #
# accuracy in best_softmax.                                                    #
#                                                                              #
# Hint: You should use a small value for num_iters as you develop your         #
# validation code so that the classifiers don't take much time to train; once  #
# you are confident that your validation code works, you should rerun the      #
# 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-6]
regularization_strengths = [2.5e4, 1e4]
```

```
for lr in learning_rates:
    for rs in regularization_strengths:
        softmax = Softmax()
        softmax.train(X_train, y_train, lr, rs, num_iters=2000)
        y_train_pred = softmax.predict(X_train)
        train_accuracy = np.mean(y_train == y_train_pred)
        y_val_pred = softmax.predict(X_val)
        val_accuracy = np.mean(y_val == y_val_pred)
        if val_accuracy > best_val:
            best_val = val_accuracy
            best_softmax = softmax
        results[(lr,rs)] = train_accuracy, val_accuracy

# 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-07 reg 1.000000e+04 train accuracy: 0.361245 val accuracy: 0.373000
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.353837 val accuracy: 0.365000
lr 1.000000e-06 reg 1.000000e+04 train accuracy: 0.367898 val accuracy: 0.361000
lr 1.000000e-06 reg 2.500000e+04 train accuracy: 0.344224 val accuracy: 0.355000
best validation accuracy achieved during cross-validation: 0.373000
```

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



[21]:
```
# Evaluate the best softmax on test set
y_test_pred = best_softmax.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
```

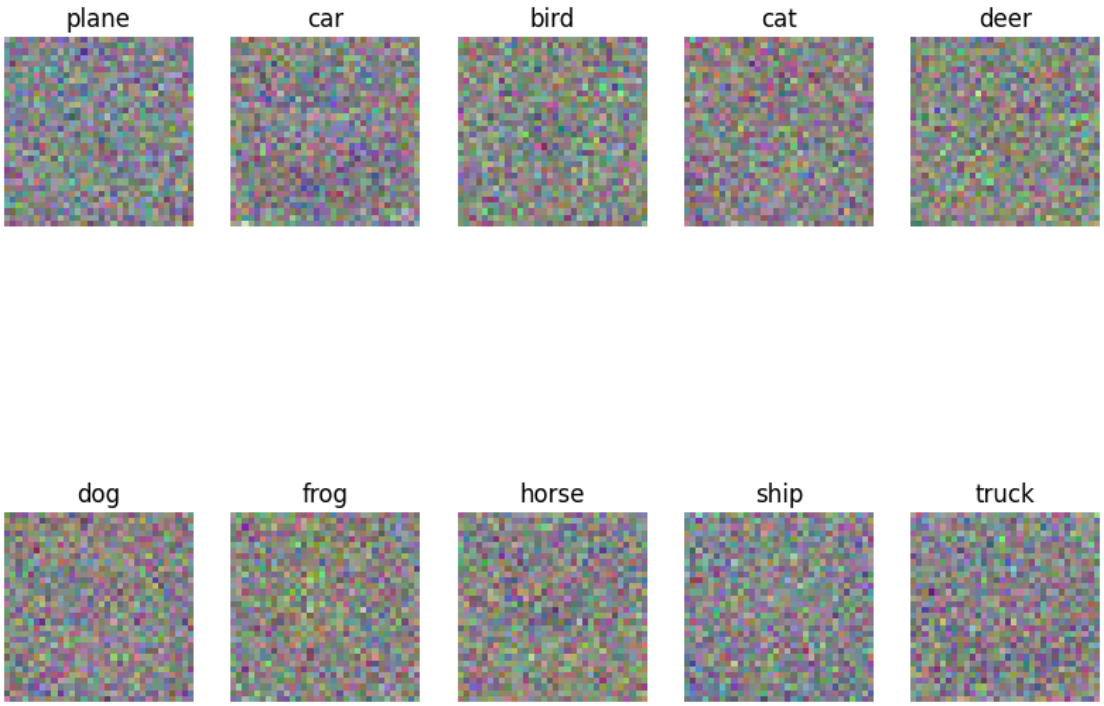
```
print('Softmax classifier on raw pixels final test set accuracy: %f' %␣
 ↪test_accuracy)
```

```
Softmax classifier on raw pixels final test set accuracy: 0.353000
```

[22]:
```
# Save best softmax model
best_softmax.save("best_softmax.npy")
```

```
best_softmax.npy saved.
```

[23]:
```
# 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_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])
```



Inline question 3

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

YourAnswer :

Inline Question 4 - *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 change the softmax loss, but leave the SVM loss unchanged.

YourAnswer :*True*

YourExplanation : *softmax* *SVM*

two_layer_net

August 25, 2025

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

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

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

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

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

1 Fully-Connected Neural Nets

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

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

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

  return out, cache
```

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

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

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

  return dx, dw
```

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

[3]:
```
# 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))))
```

[4]:
```
# Load the (preprocessed) CIFAR10 data.

data = get_CIFAR10_data()
for k, v in list(data.items()):
  print(('%s: ' % k, v.shape))
```

```
('X_train: ', (49000, 3, 32, 32))
('y_train: ', (49000,))
('X_val: ', (1000, 3, 32, 32))
('y_val: ', (1000,))
('X_test: ', (1000, 3, 32, 32))
('y_test: ', (1000,))
```

2 Affine layer: forward

Open the file `cs231n/layers.py` and implement the `affine_forward` function.

Once you are done you can test your implementaion by running the following:

[7]:
```
# Test the affine_forward function

num_inputs = 2
input_shape = (4, 5, 6)
output_dim = 3

input_size = num_inputs * np.prod(input_shape)
weight_size = output_dim * np.prod(input_shape)

x = np.linspace(-0.1, 0.5, num=input_size).reshape(num_inputs, *input_shape)
w = np.linspace(-0.2, 0.3, num=weight_size).reshape(np.prod(input_shape),␣
  ↪output_dim)
b = np.linspace(-0.3, 0.1, num=output_dim)

out, _ = affine_forward(x, w, b)
correct_out = np.array([[ 1.49834967,  1.70660132,  1.91485297],
                        [ 3.25553199,  3.5141327,   3.77273342]])

# Compare your output with ours. The error should be around e-9 or less.
print('Testing affine_forward function:')
print('difference: ', rel_error(out, correct_out))
```

```
Testing affine_forward function:
difference:  9.769849468192957e-10
```

3 Affine layer: backward

Now implement the `affine_backward` function and test your implementation using numeric gradient checking.

[8]:
```
# Test the affine_backward function
np.random.seed(231)
x = np.random.randn(10, 2, 3)
w = np.random.randn(6, 5)
b = np.random.randn(5)
dout = np.random.randn(10, 5)

dx_num = eval_numerical_gradient_array(lambda x: affine_forward(x, w, b)[0], x,␣
 ↪dout)
dw_num = eval_numerical_gradient_array(lambda w: affine_forward(x, w, b)[0], w,␣
 ↪dout)
db_num = eval_numerical_gradient_array(lambda b: affine_forward(x, w, b)[0], b,␣
 ↪dout)

_, cache = affine_forward(x, w, b)
dx, dw, db = affine_backward(dout, cache)

# The error should be around e-10 or less
print('Testing affine_backward function:')
print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

```
Testing affine_backward function:
dx error:  5.399100368651805e-11
dw error:  9.904211865398145e-11
db error:  2.4122867568119087e-11
```

4 ReLU activation: forward

Implement the forward pass for the ReLU activation function in the `relu_forward` function and test your implementation using the following:

[9]:
```
# Test the relu_forward function

x = np.linspace(-0.5, 0.5, num=12).reshape(3, 4)

out, _ = relu_forward(x)
correct_out = np.array([[ 0.,          0.,          0.,          0.,        ],
                        [ 0.,          0.,          0.04545455,  0.13636364,],
                        [ 0.22727273,  0.31818182,  0.40909091,  0.5,       ]])
```

```
# Compare your output with ours. The error should be on the order of e-8
print('Testing relu_forward function:')
print('difference: ', rel_error(out, correct_out))
```

```
Testing relu_forward function:
difference:  4.999999798022158e-08
```

5 ReLU activation: backward

Now implement the backward pass for the ReLU activation function in the `relu_backward` function and test your implementation using numeric gradient checking:

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

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

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

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

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

5.1 Inline Question 1:

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

YourAnswer : *Sigmoid&RELU Sigmoid* *RELU*

6 "Sandwich" layers

There are some common patterns of layers that are frequently used in neural nets. For example, affine layers are frequently followed by a ReLU nonlinearity. To make these common patterns easy, we define several convenience layers in the file `cs231n/layer_utils.py`.

For now take a look at the `affine_relu_forward` and `affine_relu_backward` functions, and run the following to numerically gradient check the backward pass:

[11]:
```
from cs231n.layer_utils import affine_relu_forward, affine_relu_backward
np.random.seed(231)
x = np.random.randn(2, 3, 4)
w = np.random.randn(12, 10)
b = np.random.randn(10)
dout = np.random.randn(2, 10)

out, cache = affine_relu_forward(x, w, b)
dx, dw, db = affine_relu_backward(dout, cache)

dx_num = eval_numerical_gradient_array(lambda x: affine_relu_forward(x, w,␣
 ↪b)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: affine_relu_forward(x, w,␣
 ↪b)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: affine_relu_forward(x, w,␣
 ↪b)[0], b, dout)

# Relative error should be around e-10 or less
print('Testing affine_relu_forward and affine_relu_backward:')
print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

```
Testing affine_relu_forward and affine_relu_backward:
dx error:  2.299579177309368e-11
dw error:  8.162011105764925e-11
db error:  7.826724021458994e-12
```

7 Loss layers: Softmax

Now implement the loss and gradient for softmax in the `softmax_loss` function in `cs231n/layers.py`. These should be similar to what you implemented in `cs231n/classifiers/softmax.py`. Other loss functions (e.g. `svm_loss`) can also be implemented in a modular way, however, it is not required for this assignment.

You can make sure that the implementations are correct by running the following:

[12]:
```
np.random.seed(231)
num_classes, num_inputs = 10, 50
x = 0.001 * np.random.randn(num_inputs, num_classes)
y = np.random.randint(num_classes, size=num_inputs)


dx_num = eval_numerical_gradient(lambda x: 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 softmax_loss:
loss:  2.3025458445007376
dx error:  8.234144091578429e-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.

[18]:
```
np.random.seed(231)
N, D, H, C = 3, 5, 50, 7
X = np.random.randn(N, D)
y = np.random.randint(C, size=N)

std = 1e-3
model = TwoLayerNet(input_dim=D, hidden_dim=H, num_classes=C, weight_scale=std)

print('Testing initialization ... ')
W1_std = abs(model.params['W1'].std() - std)
b1 = model.params['b1']
W2_std = abs(model.params['W2'].std() - std)
b2 = model.params['b2']
assert W1_std < std / 10, 'First layer weights do not seem right'
assert np.all(b1 == 0), 'First layer biases do not seem right'
assert W2_std < std / 10, 'Second layer weights do not seem right'
assert np.all(b2 == 0), 'Second layer biases do not seem right'

print('Testing test-time forward pass ... ')
model.params['W1'] = np.linspace(-0.7, 0.3, num=D*H).reshape(D, H)
model.params['b1'] = np.linspace(-0.1, 0.9, num=H)
model.params['W2'] = np.linspace(-0.3, 0.4, num=H*C).reshape(H, C)
model.params['b2'] = np.linspace(-0.9, 0.1, num=C)
X = np.linspace(-5.5, 4.5, num=N*D).reshape(D, N).T
scores = model.loss(X)
correct_scores = np.asarray(
  [[11.53165108,  12.2917344,   13.05181771,  13.81190102,  14.57198434, 15.
 ↪33206765,  16.09215096],
   [12.05769098,  12.74614105,  13.43459113,  14.1230412,   14.81149128, 15.
 ↪49994135,  16.18839143],
```

```
   [12.58373087,  13.20054771,  13.81736455,  14.43418138,  15.05099822, 15.
 ↪66781506,  16.2846319 ]])
scores_diff = np.abs(scores - correct_scores).sum()
assert scores_diff < 1e-6, 'Problem with test-time forward pass'

print('Testing training loss (no regularization)')
y = np.asarray([0, 5, 1])
loss, grads = model.loss(X, y)
correct_loss = 3.4702243556
assert abs(loss - correct_loss) < 1e-10, 'Problem with training-time loss'

model.reg = 1.0
loss, grads = model.loss(X, y)
correct_loss = 26.5948426952
assert abs(loss - correct_loss) < 1e-10, 'Problem with regularization loss'

# Errors should be around e-7 or less
for reg in [0.0, 0.7]:
  print('Running numeric gradient check with reg = ', reg)
  model.reg = reg
  loss, grads = model.loss(X, y)

  for name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    grad_num = eval_numerical_gradient(f, model.params[name], verbose=False)
    print('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
```

```
Testing initialization …
Testing test-time forward pass …
Testing training loss (no regularization)
Running numeric gradient check with reg =  0.0
W1 relative error: 1.83e-08
W2 relative error: 3.20e-10
b1 relative error: 9.83e-09
b2 relative error: 4.33e-10
Running numeric gradient check with reg =  0.7
W1 relative error: 2.53e-07
W2 relative error: 2.85e-08
b1 relative error: 1.56e-08
b2 relative error: 9.09e-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.                                              #
################################################################################
solver = Solver(model, data,
                update_rule='sgd',
                optim_config={'learning_rate': 1e-3},
                batch_size=100,
                num_epochs=10)
solver.train()
################################################################################
#                              END OF YOUR CODE                                #
################################################################################
```

```
(Iteration 1 / 4900) loss: 2.303487
(Epoch 0 / 10) train acc: 0.149000; val_acc: 0.147000
(Iteration 11 / 4900) loss: 2.254723
(Iteration 21 / 4900) loss: 2.135893
(Iteration 31 / 4900) loss: 2.091274
(Iteration 41 / 4900) loss: 2.030154
(Iteration 51 / 4900) loss: 1.968785
(Iteration 61 / 4900) loss: 1.954877
(Iteration 71 / 4900) loss: 1.967826
(Iteration 81 / 4900) loss: 1.979861
(Iteration 91 / 4900) loss: 1.940563
(Iteration 101 / 4900) loss: 1.790740
(Iteration 111 / 4900) loss: 1.636812
(Iteration 121 / 4900) loss: 1.707289
(Iteration 131 / 4900) loss: 1.841381
(Iteration 141 / 4900) loss: 1.728396
(Iteration 151 / 4900) loss: 1.783529
(Iteration 161 / 4900) loss: 1.708264
(Iteration 171 / 4900) loss: 1.752471
(Iteration 181 / 4900) loss: 1.700871
(Iteration 191 / 4900) loss: 1.745772
(Iteration 201 / 4900) loss: 1.877936
(Iteration 211 / 4900) loss: 1.793323
(Iteration 221 / 4900) loss: 1.803462
(Iteration 231 / 4900) loss: 1.732091
(Iteration 241 / 4900) loss: 1.584830
(Iteration 251 / 4900) loss: 1.634570
```

```
(Iteration 261 / 4900) loss: 1.567214
(Iteration 271 / 4900) loss: 1.641573
(Iteration 281 / 4900) loss: 1.517941
(Iteration 291 / 4900) loss: 1.514731
(Iteration 301 / 4900) loss: 1.540405
(Iteration 311 / 4900) loss: 1.541268
(Iteration 321 / 4900) loss: 1.644996
(Iteration 331 / 4900) loss: 1.671601
(Iteration 341 / 4900) loss: 1.688402
(Iteration 351 / 4900) loss: 1.688736
(Iteration 361 / 4900) loss: 1.498027
(Iteration 371 / 4900) loss: 1.568010
(Iteration 381 / 4900) loss: 1.639801
(Iteration 391 / 4900) loss: 1.627260
(Iteration 401 / 4900) loss: 1.521181
(Iteration 411 / 4900) loss: 1.503599
(Iteration 421 / 4900) loss: 1.563415
(Iteration 431 / 4900) loss: 1.486540
(Iteration 441 / 4900) loss: 1.348060
(Iteration 451 / 4900) loss: 1.609946
(Iteration 461 / 4900) loss: 1.686762
(Iteration 471 / 4900) loss: 1.635020
(Iteration 481 / 4900) loss: 1.558150
(Epoch 1 / 10) train acc: 0.427000; val_acc: 0.418000
(Iteration 491 / 4900) loss: 1.594659
(Iteration 501 / 4900) loss: 1.685742
(Iteration 511 / 4900) loss: 1.626713
(Iteration 521 / 4900) loss: 1.608847
(Iteration 531 / 4900) loss: 1.423092
(Iteration 541 / 4900) loss: 1.534463
(Iteration 551 / 4900) loss: 1.400493
(Iteration 561 / 4900) loss: 1.479693
(Iteration 571 / 4900) loss: 1.369642
(Iteration 581 / 4900) loss: 1.597254
(Iteration 591 / 4900) loss: 1.410743
(Iteration 601 / 4900) loss: 1.527036
(Iteration 611 / 4900) loss: 1.607009
(Iteration 621 / 4900) loss: 1.720941
(Iteration 631 / 4900) loss: 1.456196
(Iteration 641 / 4900) loss: 1.441328
(Iteration 651 / 4900) loss: 1.553721
(Iteration 661 / 4900) loss: 1.587415
(Iteration 671 / 4900) loss: 1.612159
(Iteration 681 / 4900) loss: 1.696691
(Iteration 691 / 4900) loss: 1.563208
(Iteration 701 / 4900) loss: 1.555693
(Iteration 711 / 4900) loss: 1.343112
(Iteration 721 / 4900) loss: 1.671042
```

```
(Iteration 731 / 4900) loss: 1.684535
(Iteration 741 / 4900) loss: 1.445350
(Iteration 751 / 4900) loss: 1.473687
(Iteration 761 / 4900) loss: 1.532464
(Iteration 771 / 4900) loss: 1.549338
(Iteration 781 / 4900) loss: 1.753818
(Iteration 791 / 4900) loss: 1.429340
(Iteration 801 / 4900) loss: 1.437582
(Iteration 811 / 4900) loss: 1.509476
(Iteration 821 / 4900) loss: 1.619343
(Iteration 831 / 4900) loss: 1.547170
(Iteration 841 / 4900) loss: 1.391021
(Iteration 851 / 4900) loss: 1.593820
(Iteration 861 / 4900) loss: 1.485180
(Iteration 871 / 4900) loss: 1.282316
(Iteration 881 / 4900) loss: 1.408055
(Iteration 891 / 4900) loss: 1.498773
(Iteration 901 / 4900) loss: 1.624439
(Iteration 911 / 4900) loss: 1.707720
(Iteration 921 / 4900) loss: 1.599250
(Iteration 931 / 4900) loss: 1.433914
(Iteration 941 / 4900) loss: 1.637841
(Iteration 951 / 4900) loss: 1.253039
(Iteration 961 / 4900) loss: 1.647855
(Iteration 971 / 4900) loss: 1.388573
(Epoch 2 / 10) train acc: 0.445000; val_acc: 0.444000
(Iteration 981 / 4900) loss: 1.471087
(Iteration 991 / 4900) loss: 1.305904
(Iteration 1001 / 4900) loss: 1.466251
(Iteration 1011 / 4900) loss: 1.408778
(Iteration 1021 / 4900) loss: 1.347607
(Iteration 1031 / 4900) loss: 1.071513
(Iteration 1041 / 4900) loss: 1.571106
(Iteration 1051 / 4900) loss: 1.395886
(Iteration 1061 / 4900) loss: 1.399213
(Iteration 1071 / 4900) loss: 1.620174
(Iteration 1081 / 4900) loss: 1.349864
(Iteration 1091 / 4900) loss: 1.466674
(Iteration 1101 / 4900) loss: 1.376463
(Iteration 1111 / 4900) loss: 1.434514
(Iteration 1121 / 4900) loss: 1.362771
(Iteration 1131 / 4900) loss: 1.616556
(Iteration 1141 / 4900) loss: 1.576131
(Iteration 1151 / 4900) loss: 1.341089
(Iteration 1161 / 4900) loss: 1.367591
(Iteration 1171 / 4900) loss: 1.668873
(Iteration 1181 / 4900) loss: 1.665671
(Iteration 1191 / 4900) loss: 1.699496
```

```
(Iteration 1201 / 4900) loss: 1.352155
(Iteration 1211 / 4900) loss: 1.457973
(Iteration 1221 / 4900) loss: 1.576693
(Iteration 1231 / 4900) loss: 1.517705
(Iteration 1241 / 4900) loss: 1.605472
(Iteration 1251 / 4900) loss: 1.643659
(Iteration 1261 / 4900) loss: 1.458922
(Iteration 1271 / 4900) loss: 1.439436
(Iteration 1281 / 4900) loss: 1.300111
(Iteration 1291 / 4900) loss: 1.415811
(Iteration 1301 / 4900) loss: 1.456798
(Iteration 1311 / 4900) loss: 1.368590
(Iteration 1321 / 4900) loss: 1.165217
(Iteration 1331 / 4900) loss: 1.524306
(Iteration 1341 / 4900) loss: 1.541058
(Iteration 1351 / 4900) loss: 1.392691
(Iteration 1361 / 4900) loss: 1.293754
(Iteration 1371 / 4900) loss: 1.331391
(Iteration 1381 / 4900) loss: 1.417275
(Iteration 1391 / 4900) loss: 1.446179
(Iteration 1401 / 4900) loss: 1.377617
(Iteration 1411 / 4900) loss: 1.508604
(Iteration 1421 / 4900) loss: 1.573271
(Iteration 1431 / 4900) loss: 1.636821
(Iteration 1441 / 4900) loss: 1.517650
(Iteration 1451 / 4900) loss: 1.305768
(Iteration 1461 / 4900) loss: 1.538476
(Epoch 3 / 10) train acc: 0.503000; val_acc: 0.478000
(Iteration 1471 / 4900) loss: 1.682668
(Iteration 1481 / 4900) loss: 1.501692
(Iteration 1491 / 4900) loss: 1.502066
(Iteration 1501 / 4900) loss: 1.520728
(Iteration 1511 / 4900) loss: 1.574113
(Iteration 1521 / 4900) loss: 1.532827
(Iteration 1531 / 4900) loss: 1.355544
(Iteration 1541 / 4900) loss: 1.520821
(Iteration 1551 / 4900) loss: 1.549930
(Iteration 1561 / 4900) loss: 1.513822
(Iteration 1571 / 4900) loss: 1.512529
(Iteration 1581 / 4900) loss: 1.513984
(Iteration 1591 / 4900) loss: 1.331149
(Iteration 1601 / 4900) loss: 1.451304
(Iteration 1611 / 4900) loss: 1.345584
(Iteration 1621 / 4900) loss: 1.493005
(Iteration 1631 / 4900) loss: 1.426951
(Iteration 1641 / 4900) loss: 1.430975
(Iteration 1651 / 4900) loss: 1.416253
(Iteration 1661 / 4900) loss: 1.275402
```

```
(Iteration 1671 / 4900) loss: 1.474441
(Iteration 1681 / 4900) loss: 1.382272
(Iteration 1691 / 4900) loss: 1.228981
(Iteration 1701 / 4900) loss: 1.401551
(Iteration 1711 / 4900) loss: 1.491151
(Iteration 1721 / 4900) loss: 1.456593
(Iteration 1731 / 4900) loss: 1.475202
(Iteration 1741 / 4900) loss: 1.667269
(Iteration 1751 / 4900) loss: 1.530020
(Iteration 1761 / 4900) loss: 1.596710
(Iteration 1771 / 4900) loss: 1.447997
(Iteration 1781 / 4900) loss: 1.404700
(Iteration 1791 / 4900) loss: 1.406534
(Iteration 1801 / 4900) loss: 1.322726
(Iteration 1811 / 4900) loss: 1.478753
(Iteration 1821 / 4900) loss: 1.488606
(Iteration 1831 / 4900) loss: 1.513000
(Iteration 1841 / 4900) loss: 1.568397
(Iteration 1851 / 4900) loss: 1.294681
(Iteration 1861 / 4900) loss: 1.372036
(Iteration 1871 / 4900) loss: 1.370894
(Iteration 1881 / 4900) loss: 1.264203
(Iteration 1891 / 4900) loss: 1.242944
(Iteration 1901 / 4900) loss: 1.397906
(Iteration 1911 / 4900) loss: 1.548630
(Iteration 1921 / 4900) loss: 1.420571
(Iteration 1931 / 4900) loss: 1.566935
(Iteration 1941 / 4900) loss: 1.674050
(Iteration 1951 / 4900) loss: 1.242533
(Epoch 4 / 10) train acc: 0.508000; val_acc: 0.490000
(Iteration 1961 / 4900) loss: 1.227845
(Iteration 1971 / 4900) loss: 1.306863
(Iteration 1981 / 4900) loss: 1.219366
(Iteration 1991 / 4900) loss: 1.338999
(Iteration 2001 / 4900) loss: 1.442515
(Iteration 2011 / 4900) loss: 1.443315
(Iteration 2021 / 4900) loss: 1.429054
(Iteration 2031 / 4900) loss: 1.466453
(Iteration 2041 / 4900) loss: 1.571518
(Iteration 2051 / 4900) loss: 1.432460
(Iteration 2061 / 4900) loss: 1.281146
(Iteration 2071 / 4900) loss: 1.287704
(Iteration 2081 / 4900) loss: 1.590442
(Iteration 2091 / 4900) loss: 1.400346
(Iteration 2101 / 4900) loss: 1.478099
(Iteration 2111 / 4900) loss: 1.472024
(Iteration 2121 / 4900) loss: 1.620603
(Iteration 2131 / 4900) loss: 1.339908
```

```
(Iteration 2141 / 4900) loss: 1.400660
(Iteration 2151 / 4900) loss: 1.307383
(Iteration 2161 / 4900) loss: 1.398068
(Iteration 2171 / 4900) loss: 1.552050
(Iteration 2181 / 4900) loss: 1.242888
(Iteration 2191 / 4900) loss: 1.534167
(Iteration 2201 / 4900) loss: 1.464243
(Iteration 2211 / 4900) loss: 1.328941
(Iteration 2221 / 4900) loss: 1.430504
(Iteration 2231 / 4900) loss: 1.467019
(Iteration 2241 / 4900) loss: 1.654324
(Iteration 2251 / 4900) loss: 1.529471
(Iteration 2261 / 4900) loss: 1.452062
(Iteration 2271 / 4900) loss: 1.473781
(Iteration 2281 / 4900) loss: 1.412426
(Iteration 2291 / 4900) loss: 1.536222
(Iteration 2301 / 4900) loss: 1.444413
(Iteration 2311 / 4900) loss: 1.419637
(Iteration 2321 / 4900) loss: 1.338542
(Iteration 2331 / 4900) loss: 1.407574
(Iteration 2341 / 4900) loss: 1.477994
(Iteration 2351 / 4900) loss: 1.212322
(Iteration 2361 / 4900) loss: 1.422135
(Iteration 2371 / 4900) loss: 1.537894
(Iteration 2381 / 4900) loss: 1.480643
(Iteration 2391 / 4900) loss: 1.395270
(Iteration 2401 / 4900) loss: 1.553300
(Iteration 2411 / 4900) loss: 1.408745
(Iteration 2421 / 4900) loss: 1.336088
(Iteration 2431 / 4900) loss: 1.299220
(Iteration 2441 / 4900) loss: 1.550683
(Epoch 5 / 10) train acc: 0.500000; val_acc: 0.473000
(Iteration 2451 / 4900) loss: 1.495309
(Iteration 2461 / 4900) loss: 1.436777
(Iteration 2471 / 4900) loss: 1.293561
(Iteration 2481 / 4900) loss: 1.399096
(Iteration 2491 / 4900) loss: 1.372715
(Iteration 2501 / 4900) loss: 1.251013
(Iteration 2511 / 4900) loss: 1.093590
(Iteration 2521 / 4900) loss: 1.318738
(Iteration 2531 / 4900) loss: 1.216006
(Iteration 2541 / 4900) loss: 1.367427
(Iteration 2551 / 4900) loss: 1.301243
(Iteration 2561 / 4900) loss: 1.179023
(Iteration 2571 / 4900) loss: 1.325414
(Iteration 2581 / 4900) loss: 1.380697
(Iteration 2591 / 4900) loss: 1.497630
(Iteration 2601 / 4900) loss: 1.370788
```

```
(Iteration 2611 / 4900) loss: 1.177841
(Iteration 2621 / 4900) loss: 1.436057
(Iteration 2631 / 4900) loss: 1.357116
(Iteration 2641 / 4900) loss: 1.227267
(Iteration 2651 / 4900) loss: 1.413695
(Iteration 2661 / 4900) loss: 1.245986
(Iteration 2671 / 4900) loss: 1.258711
(Iteration 2681 / 4900) loss: 1.274348
(Iteration 2691 / 4900) loss: 1.296396
(Iteration 2701 / 4900) loss: 1.162785
(Iteration 2711 / 4900) loss: 1.624301
(Iteration 2721 / 4900) loss: 1.417348
(Iteration 2731 / 4900) loss: 1.377316
(Iteration 2741 / 4900) loss: 1.225495
(Iteration 2751 / 4900) loss: 1.546758
(Iteration 2761 / 4900) loss: 1.301341
(Iteration 2771 / 4900) loss: 1.184911
(Iteration 2781 / 4900) loss: 1.439999
(Iteration 2791 / 4900) loss: 1.192931
(Iteration 2801 / 4900) loss: 1.239439
(Iteration 2811 / 4900) loss: 1.362503
(Iteration 2821 / 4900) loss: 1.381676
(Iteration 2831 / 4900) loss: 1.673327
(Iteration 2841 / 4900) loss: 1.354501
(Iteration 2851 / 4900) loss: 1.310560
(Iteration 2861 / 4900) loss: 1.236617
(Iteration 2871 / 4900) loss: 1.441874
(Iteration 2881 / 4900) loss: 1.336798
(Iteration 2891 / 4900) loss: 1.351488
(Iteration 2901 / 4900) loss: 1.541048
(Iteration 2911 / 4900) loss: 1.717349
(Iteration 2921 / 4900) loss: 1.555052
(Iteration 2931 / 4900) loss: 1.353467
(Epoch 6 / 10) train acc: 0.507000; val_acc: 0.488000
(Iteration 2941 / 4900) loss: 1.452485
(Iteration 2951 / 4900) loss: 1.399589
(Iteration 2961 / 4900) loss: 1.282509
(Iteration 2971 / 4900) loss: 1.550815
(Iteration 2981 / 4900) loss: 1.533155
(Iteration 2991 / 4900) loss: 1.441940
(Iteration 3001 / 4900) loss: 1.534632
(Iteration 3011 / 4900) loss: 1.102935
(Iteration 3021 / 4900) loss: 1.237741
(Iteration 3031 / 4900) loss: 1.497372
(Iteration 3041 / 4900) loss: 1.274386
(Iteration 3051 / 4900) loss: 1.496054
(Iteration 3061 / 4900) loss: 1.484403
(Iteration 3071 / 4900) loss: 1.330205
```

```
(Iteration 3081 / 4900) loss: 1.307539
(Iteration 3091 / 4900) loss: 1.151905
(Iteration 3101 / 4900) loss: 1.341648
(Iteration 3111 / 4900) loss: 1.372302
(Iteration 3121 / 4900) loss: 1.074381
(Iteration 3131 / 4900) loss: 1.372191
(Iteration 3141 / 4900) loss: 1.262287
(Iteration 3151 / 4900) loss: 1.172670
(Iteration 3161 / 4900) loss: 1.100097
(Iteration 3171 / 4900) loss: 1.477392
(Iteration 3181 / 4900) loss: 1.408941
(Iteration 3191 / 4900) loss: 1.459297
(Iteration 3201 / 4900) loss: 1.311059
(Iteration 3211 / 4900) loss: 1.368012
(Iteration 3221 / 4900) loss: 1.356880
(Iteration 3231 / 4900) loss: 1.488049
(Iteration 3241 / 4900) loss: 1.473089
(Iteration 3251 / 4900) loss: 1.216076
(Iteration 3261 / 4900) loss: 1.240122
(Iteration 3271 / 4900) loss: 1.473366
(Iteration 3281 / 4900) loss: 1.330930
(Iteration 3291 / 4900) loss: 1.386380
(Iteration 3301 / 4900) loss: 1.328927
(Iteration 3311 / 4900) loss: 1.458909
(Iteration 3321 / 4900) loss: 1.331222
(Iteration 3331 / 4900) loss: 1.275402
(Iteration 3341 / 4900) loss: 1.280304
(Iteration 3351 / 4900) loss: 1.366501
(Iteration 3361 / 4900) loss: 1.467283
(Iteration 3371 / 4900) loss: 1.398850
(Iteration 3381 / 4900) loss: 1.537274
(Iteration 3391 / 4900) loss: 1.384283
(Iteration 3401 / 4900) loss: 1.471928
(Iteration 3411 / 4900) loss: 1.390523
(Iteration 3421 / 4900) loss: 1.386682
(Epoch 7 / 10) train acc: 0.461000; val_acc: 0.462000
(Iteration 3431 / 4900) loss: 1.526743
(Iteration 3441 / 4900) loss: 1.308900
(Iteration 3451 / 4900) loss: 1.504086
(Iteration 3461 / 4900) loss: 1.316280
(Iteration 3471 / 4900) loss: 1.379210
(Iteration 3481 / 4900) loss: 1.607882
(Iteration 3491 / 4900) loss: 1.401041
(Iteration 3501 / 4900) loss: 1.332430
(Iteration 3511 / 4900) loss: 1.302347
(Iteration 3521 / 4900) loss: 1.187151
(Iteration 3531 / 4900) loss: 1.461725
(Iteration 3541 / 4900) loss: 1.428191
```

```
(Iteration 3551 / 4900) loss: 1.438755
(Iteration 3561 / 4900) loss: 1.445622
(Iteration 3571 / 4900) loss: 1.170521
(Iteration 3581 / 4900) loss: 1.350610
(Iteration 3591 / 4900) loss: 1.270479
(Iteration 3601 / 4900) loss: 1.196299
(Iteration 3611 / 4900) loss: 1.596599
(Iteration 3621 / 4900) loss: 1.410003
(Iteration 3631 / 4900) loss: 1.479319
(Iteration 3641 / 4900) loss: 1.453291
(Iteration 3651 / 4900) loss: 1.082351
(Iteration 3661 / 4900) loss: 1.556995
(Iteration 3671 / 4900) loss: 1.291175
(Iteration 3681 / 4900) loss: 1.400324
(Iteration 3691 / 4900) loss: 1.241501
(Iteration 3701 / 4900) loss: 1.284805
(Iteration 3711 / 4900) loss: 1.411856
(Iteration 3721 / 4900) loss: 1.206465
(Iteration 3731 / 4900) loss: 1.476810
(Iteration 3741 / 4900) loss: 1.405465
(Iteration 3751 / 4900) loss: 1.149903
(Iteration 3761 / 4900) loss: 1.374013
(Iteration 3771 / 4900) loss: 1.156698
(Iteration 3781 / 4900) loss: 1.166845
(Iteration 3791 / 4900) loss: 1.319101
(Iteration 3801 / 4900) loss: 1.179914
(Iteration 3811 / 4900) loss: 1.238093
(Iteration 3821 / 4900) loss: 1.366272
(Iteration 3831 / 4900) loss: 1.377238
(Iteration 3841 / 4900) loss: 1.270142
(Iteration 3851 / 4900) loss: 1.088256
(Iteration 3861 / 4900) loss: 1.218211
(Iteration 3871 / 4900) loss: 1.206768
(Iteration 3881 / 4900) loss: 1.368871
(Iteration 3891 / 4900) loss: 1.412026
(Iteration 3901 / 4900) loss: 1.098908
(Iteration 3911 / 4900) loss: 1.254894
(Epoch 8 / 10) train acc: 0.542000; val_acc: 0.481000
(Iteration 3921 / 4900) loss: 1.293192
(Iteration 3931 / 4900) loss: 1.436098
(Iteration 3941 / 4900) loss: 1.247927
(Iteration 3951 / 4900) loss: 1.245119
(Iteration 3961 / 4900) loss: 1.444173
(Iteration 3971 / 4900) loss: 1.413343
(Iteration 3981 / 4900) loss: 1.526975
(Iteration 3991 / 4900) loss: 1.247483
(Iteration 4001 / 4900) loss: 1.357966
(Iteration 4011 / 4900) loss: 1.242586
```

```
(Iteration 4021 / 4900) loss: 1.519155
(Iteration 4031 / 4900) loss: 1.151253
(Iteration 4041 / 4900) loss: 1.168217
(Iteration 4051 / 4900) loss: 1.226382
(Iteration 4061 / 4900) loss: 1.337897
(Iteration 4071 / 4900) loss: 1.382107
(Iteration 4081 / 4900) loss: 1.374753
(Iteration 4091 / 4900) loss: 1.210904
(Iteration 4101 / 4900) loss: 1.287255
(Iteration 4111 / 4900) loss: 1.374047
(Iteration 4121 / 4900) loss: 1.490457
(Iteration 4131 / 4900) loss: 1.290114
(Iteration 4141 / 4900) loss: 1.289684
(Iteration 4151 / 4900) loss: 1.297028
(Iteration 4161 / 4900) loss: 1.358617
(Iteration 4171 / 4900) loss: 1.226070
(Iteration 4181 / 4900) loss: 1.084983
(Iteration 4191 / 4900) loss: 1.377632
(Iteration 4201 / 4900) loss: 1.124430
(Iteration 4211 / 4900) loss: 1.405868
(Iteration 4221 / 4900) loss: 1.162044
(Iteration 4231 / 4900) loss: 1.401668
(Iteration 4241 / 4900) loss: 1.216035
(Iteration 4251 / 4900) loss: 1.205128
(Iteration 4261 / 4900) loss: 1.303566
(Iteration 4271 / 4900) loss: 1.168465
(Iteration 4281 / 4900) loss: 1.498610
(Iteration 4291 / 4900) loss: 1.194960
(Iteration 4301 / 4900) loss: 1.096140
(Iteration 4311 / 4900) loss: 1.465805
(Iteration 4321 / 4900) loss: 1.401990
(Iteration 4331 / 4900) loss: 1.145709
(Iteration 4341 / 4900) loss: 1.347424
(Iteration 4351 / 4900) loss: 1.275433
(Iteration 4361 / 4900) loss: 1.132879
(Iteration 4371 / 4900) loss: 1.417004
(Iteration 4381 / 4900) loss: 1.178638
(Iteration 4391 / 4900) loss: 1.367753
(Iteration 4401 / 4900) loss: 1.014826
(Epoch 9 / 10) train acc: 0.518000; val_acc: 0.517000
(Iteration 4411 / 4900) loss: 1.363271
(Iteration 4421 / 4900) loss: 1.449370
(Iteration 4431 / 4900) loss: 1.622466
(Iteration 4441 / 4900) loss: 1.370894
(Iteration 4451 / 4900) loss: 1.516499
(Iteration 4461 / 4900) loss: 1.257458
(Iteration 4471 / 4900) loss: 1.296261
(Iteration 4481 / 4900) loss: 1.190516
```

```
(Iteration 4491 / 4900) loss: 1.256757
(Iteration 4501 / 4900) loss: 1.357874
(Iteration 4511 / 4900) loss: 1.330419
(Iteration 4521 / 4900) loss: 1.392591
(Iteration 4531 / 4900) loss: 1.277303
(Iteration 4541 / 4900) loss: 1.368520
(Iteration 4551 / 4900) loss: 1.406442
(Iteration 4561 / 4900) loss: 1.386743
(Iteration 4571 / 4900) loss: 1.101593
(Iteration 4581 / 4900) loss: 1.220628
(Iteration 4591 / 4900) loss: 1.160502
(Iteration 4601 / 4900) loss: 0.997767
(Iteration 4611 / 4900) loss: 1.205034
(Iteration 4621 / 4900) loss: 1.245780
(Iteration 4631 / 4900) loss: 1.380370
(Iteration 4641 / 4900) loss: 1.259005
(Iteration 4651 / 4900) loss: 1.278861
(Iteration 4661 / 4900) loss: 1.369618
(Iteration 4671 / 4900) loss: 1.395404
(Iteration 4681 / 4900) loss: 1.211073
(Iteration 4691 / 4900) loss: 1.323061
(Iteration 4701 / 4900) loss: 1.231985
(Iteration 4711 / 4900) loss: 1.173902
(Iteration 4721 / 4900) loss: 1.378262
(Iteration 4731 / 4900) loss: 1.299821
(Iteration 4741 / 4900) loss: 1.298937
(Iteration 4751 / 4900) loss: 1.278151
(Iteration 4761 / 4900) loss: 1.294622
(Iteration 4771 / 4900) loss: 1.155663
(Iteration 4781 / 4900) loss: 1.289296
(Iteration 4791 / 4900) loss: 1.221972
(Iteration 4801 / 4900) loss: 1.231049
(Iteration 4811 / 4900) loss: 1.307716
(Iteration 4821 / 4900) loss: 1.424832
(Iteration 4831 / 4900) loss: 1.257097
(Iteration 4841 / 4900) loss: 1.328903
(Iteration 4851 / 4900) loss: 1.218810
(Iteration 4861 / 4900) loss: 1.289409
(Iteration 4871 / 4900) loss: 1.443368
(Iteration 4881 / 4900) loss: 1.313781
(Iteration 4891 / 4900) loss: 1.355657
(Epoch 10 / 10) train acc: 0.582000; val_acc: 0.492000
```

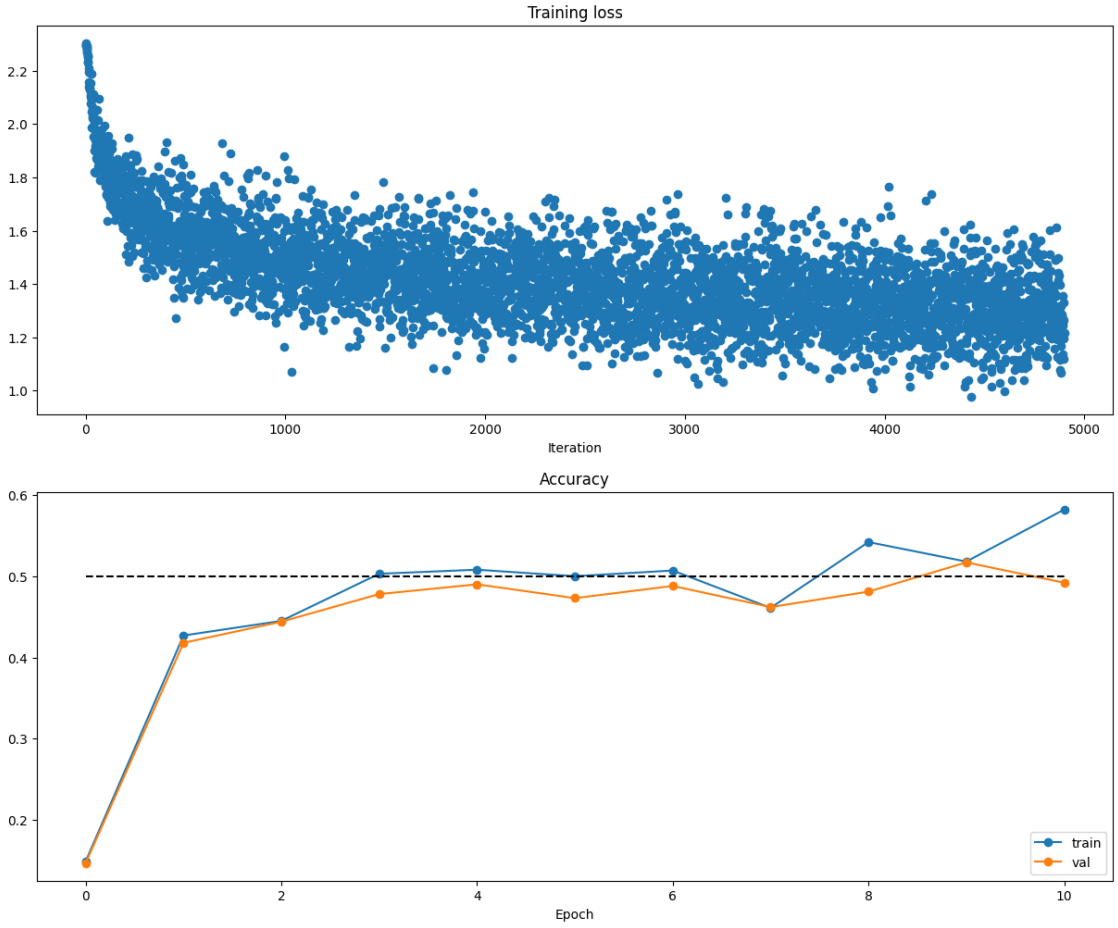
10 Debug the training

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.

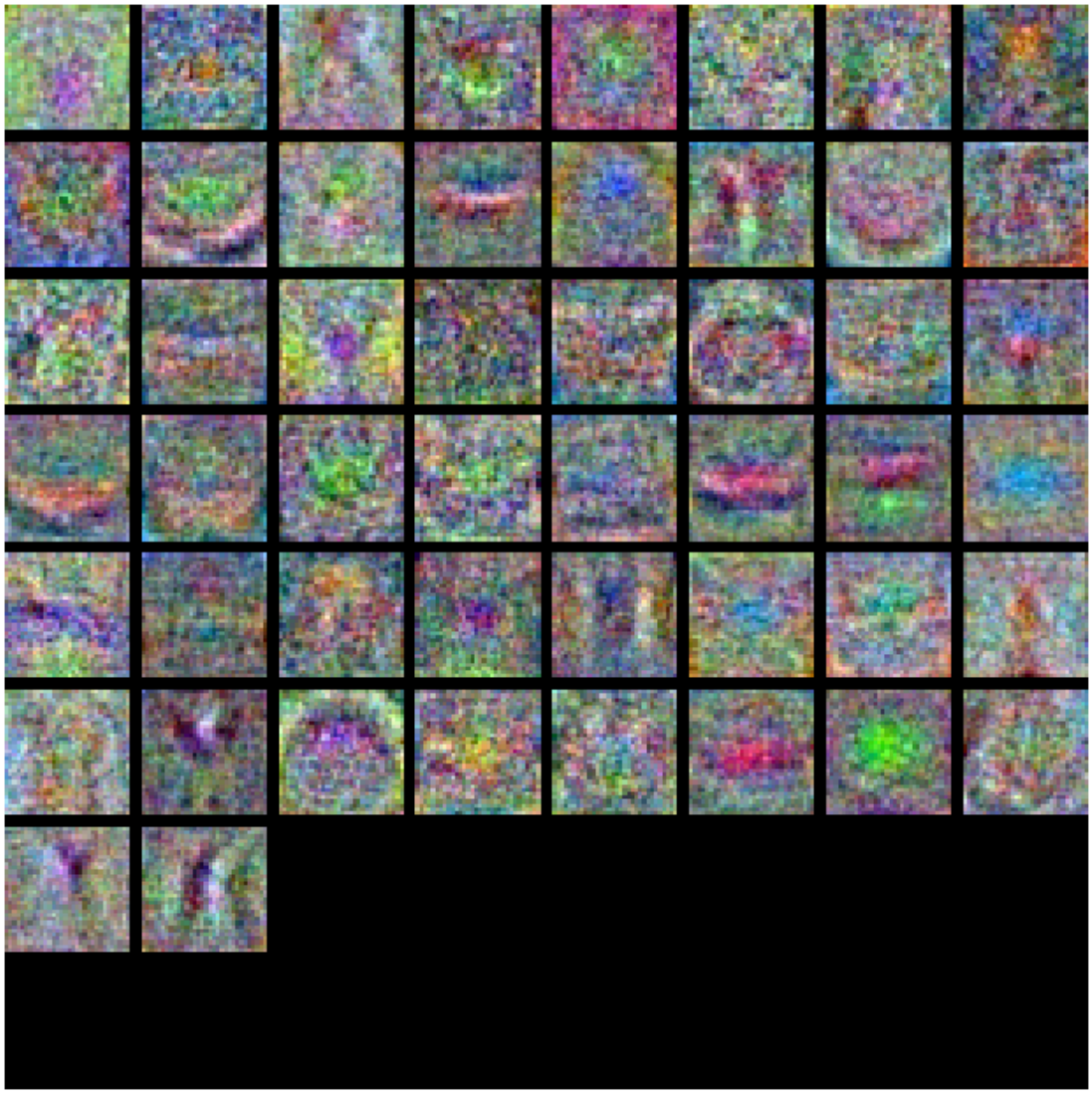
One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

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.).

[54]:
```
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.                       ␣
 ↪  #
################################################################################
results = {}
best_val = -1

learning_rates = np.array([0.7, 0.8, 0.9, 1, 1.1])*1e-3
regularization_strengths = [0.75, 1, 1.25]

import itertools

for lr, reg in itertools.product(learning_rates, regularization_strengths):
    # Create Two Layer Net and train it with Solver
    model = TwoLayerNet(hidden_dim=75, reg=reg)
```

```
    solver = Solver(model, data,
                update_rule='sgd',
                optim_config={'learning_rate': lr},
                batch_size=100,
                num_epochs=10,verbose=False)
    solver.train()

    # Compute validation set accuracy and append to the dictionary
    results[(lr, reg)] = solver.best_val_acc

    # Save if validation accuracy is the best
    if results[(lr, reg)] > best_val:
        best_val = results[(lr, reg)]
        best_model = model

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

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

################################################################################
#                              END OF YOUR CODE                                #
################################################################################
```

```
lr 7.000000e-04 reg 7.500000e-01 val accuracy: 0.504000
lr 7.000000e-04 reg 1.000000e+00 val accuracy: 0.498000
lr 7.000000e-04 reg 1.250000e+00 val accuracy: 0.510000
lr 8.000000e-04 reg 7.500000e-01 val accuracy: 0.503000
lr 8.000000e-04 reg 1.000000e+00 val accuracy: 0.495000
lr 8.000000e-04 reg 1.250000e+00 val accuracy: 0.484000
lr 9.000000e-04 reg 7.500000e-01 val accuracy: 0.494000
lr 9.000000e-04 reg 1.000000e+00 val accuracy: 0.492000
lr 9.000000e-04 reg 1.250000e+00 val accuracy: 0.495000
lr 1.000000e-03 reg 7.500000e-01 val accuracy: 0.500000
lr 1.000000e-03 reg 1.000000e+00 val accuracy: 0.486000
lr 1.000000e-03 reg 1.250000e+00 val accuracy: 0.501000
lr 1.100000e-03 reg 7.500000e-01 val accuracy: 0.500000
lr 1.100000e-03 reg 1.000000e+00 val accuracy: 0.489000
lr 1.100000e-03 reg 1.250000e+00 val accuracy: 0.497000
best validation accuracy achieved during cross-validation: 0.510000
```

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.

[55]:
```
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:  0.51
```

[56]:
```
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:  0.511
```

[57]:
```
# Save best model
best_model.save("best_two_layer_net.npy")
```

```
best_two_layer_net.npy saved.
```

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.

Your Answer :*1,3*

Your Explanation :*1 3 2*

features

August 25, 2025

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

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

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

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

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

1 Image features exercise

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

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

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

[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
num_color_bins = 25 # 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 Softmax classifier on features

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

[52]:
```
# Use the validation set to tune the learning rate and regularization strength

from cs231n.classifiers.linear_classifier import Softmax

learning_rates = [1e-7, 1e-6]
regularization_strengths = [5e5, 5e6]

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 Softmax; save#
# the best trained classifer in best_softmax. If you carefully tune the model, #
# you should be able to get accuracy of above 0.42 on the validation set.      #
################################################################################
for rs in regularization_strengths:
    for lr in learning_rates:
        svm = Softmax()
        loss_hist = svm.train(X_train_feats, y_train, lr, rs, num_iters=6000)
        y_train_pred = svm.predict(X_train_feats)
        train_accuracy = np.mean(y_train == y_train_pred)
        y_val_pred = svm.predict(X_val_feats)
        val_accuracy = np.mean(y_val == y_val_pred)
        if val_accuracy > best_val:
            best_val = val_accuracy
            best_softmax = svm
        results[(lr,rs)] = train_accuracy, val_accuracy

# 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)
```

```
/content/drive/My
Drive/cs231n/assignments/assignment1/cs231n/classifiers/softmax.py:95:
RuntimeWarning: overflow encountered in scalar multiply
  # loss, storing the result in dW.                                      #
/content/drive/My
Drive/cs231n/assignments/assignment1/cs231n/classifiers/softmax.py:95:
RuntimeWarning: overflow encountered in multiply
  # loss, storing the result in dW.                                      #
/content/drive/My
Drive/cs231n/assignments/assignment1/cs231n/classifiers/softmax.py:101:
RuntimeWarning: overflow encountered in multiply

/content/drive/My
Drive/cs231n/assignments/assignment1/cs231n/classifiers/softmax.py:82:
RuntimeWarning: invalid value encountered in subtract
  softmax_output = np.exp(shift_scores)/np.sum(np.exp(shift_scores), axis =
1).reshape(-1,1)
lr 1.000000e-07 reg 5.000000e+05 train accuracy: 0.421163 val accuracy: 0.430000
lr 1.000000e-07 reg 5.000000e+06 train accuracy: 0.389612 val accuracy: 0.389000
lr 1.000000e-06 reg 5.000000e+05 train accuracy: 0.370796 val accuracy: 0.360000
lr 1.000000e-06 reg 5.000000e+06 train accuracy: 0.100265 val accuracy: 0.087000
best validation accuracy achieved: 0.430000
```

[53]:
```
# Evaluate your trained Softmax on the test set: you should be able to get at␣
 ↪least 0.42
y_test_pred = best_softmax.predict(X_test_feats)
test_accuracy = np.mean(y_test == y_test_pred)
print(test_accuracy)
```

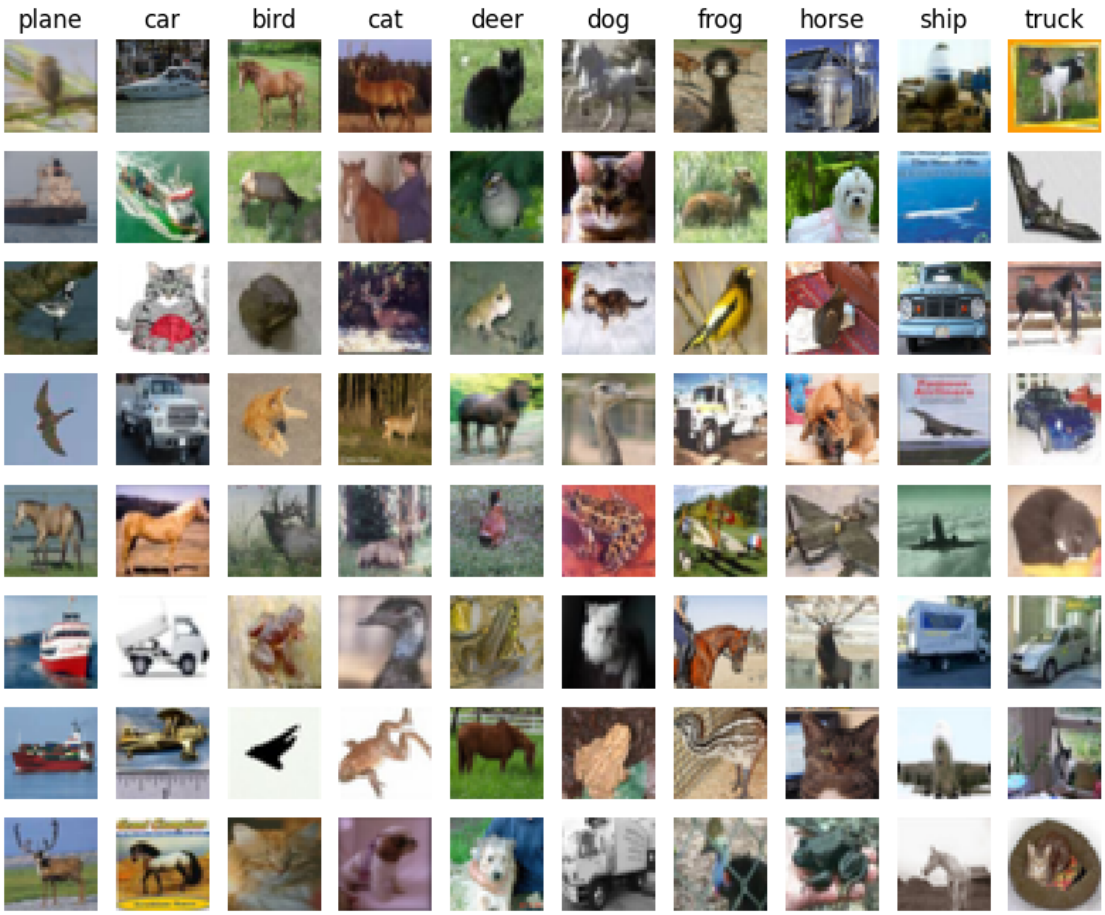
```
0.42
```

[47]:
```
# Save best softmax model
best_softmax.save("best_softmax_features.npy")
```

```
Model saved to best_softmax_features.npy
```

[46]:
```
# An important way to gain intuition about how an algorithm works is to
# visualize the mistakes that it makes. In this visualization, we show examples
# of images that are misclassified by our current system. The first column
# shows images that our system labeled as "plane" but whose true label is
# something other than "plane".

examples_per_class = 8
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
 ↪'ship', 'truck']
```

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



1.3.1 Inline question 1:

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

YourAnswer : *Yes*

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.

[48]:
```
# 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, 170)
(49000, 169)
```

[56]:
```
from cs231n.classifiers.fc_net import TwoLayerNet
from cs231n.solver import Solver

input_dim = X_train_feats.shape[1]
hidden_dim = 500
num_classes = 10

data = {
    'X_train': X_train_feats,
    'y_train': y_train,
    'X_val': X_val_feats,
    'y_val': y_val,
    'X_test': X_test_feats,
    'y_test': y_test,
}

net = TwoLayerNet(input_dim, hidden_dim, num_classes)
best_net = None

################################################################################
# TODO: Train a two-layer neural network on image features. You may want to    #
# cross-validate various parameters as in previous sections. Store your best   #
# model in the best_net variable.                                              #
################################################################################
results = {}
best_val = -1
```

```
learning_rates = np.linspace(1e-2, 2.75e-2, 4)
regularization_strengths = np.geomspace(1e-6, 1e-4, 3)

data = {
    'X_train': X_train_feats,
    'X_val': X_val_feats,
    'X_test': X_test_feats,
    'y_train': y_train,
    'y_val': y_val,
    'y_test': y_test
}

import itertools

for lr, reg in itertools.product(learning_rates, regularization_strengths):
    # Create Two Layer Net and train it with Solver
    model = net
    solver = Solver(model, data,
                update_rule='sgd',
                optim_config={'learning_rate': lr},
                batch_size=100,
                num_epochs=10,verbose=False)
    solver.train()

    # Compute validation set accuracy and append to the dictionary
    results[(lr, reg)] = solver.best_val_acc

    # Save if validation accuracy is the best
    if results[(lr, reg)] > best_val:
        best_val = results[(lr, reg)]
        best_net = model

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

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

```
lr 1.000000e-02 reg 1.000000e-06 val accuracy: 0.517000
lr 1.000000e-02 reg 1.000000e-05 val accuracy: 0.536000
lr 1.000000e-02 reg 1.000000e-04 val accuracy: 0.555000
lr 1.583333e-02 reg 1.000000e-06 val accuracy: 0.567000
lr 1.583333e-02 reg 1.000000e-05 val accuracy: 0.592000
lr 1.583333e-02 reg 1.000000e-04 val accuracy: 0.603000
lr 2.166667e-02 reg 1.000000e-06 val accuracy: 0.607000
```

```
lr 2.166667e-02 reg 1.000000e-05 val accuracy: 0.608000
lr 2.166667e-02 reg 1.000000e-04 val accuracy: 0.612000
lr 2.750000e-02 reg 1.000000e-06 val accuracy: 0.620000
lr 2.750000e-02 reg 1.000000e-05 val accuracy: 0.616000
lr 2.750000e-02 reg 1.000000e-04 val accuracy: 0.612000
best validation accuracy achieved during cross-validation: 0.620000
```

[57]:
```
# Run your best neural net classifier on the test set. You should be able
# to get more than 58% accuracy. It is also possible to get >60% accuracy
# with careful tuning.

y_test_pred = np.argmax(best_net.loss(data['X_test']), axis=1)
test_acc = (y_test_pred == data['y_test']).mean()
print(test_acc)
```

```
0.602
```

[58]:
```
# Save best model
best_net.save("best_two_layer_net_features.npy")
```

```
best_two_layer_net_features.npy saved.
```

FullyConnectedNets

August 25, 2025

[10]:
```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive')

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

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

# This downloads the CIFAR-10 dataset to your Drive
# if it doesn't already exist.
%cd /content/drive/My\ Drive/$FOLDERNAME/cs231n/datasets/
# %cd /content/drive/My\ Drive/$FOLDERNAME
!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 Multi-Layer Fully Connected Network

In this exercise, you will implement a fully connected network with an arbitrary number of hidden layers.

[11]:
```
# from google.colab import drive
# drive.mount('/content/drive')
```

Read through the `FullyConnectedNet` class in the file `cs231n/classifiers/fc_net.py`.

Implement the network initialization, forward pass, and backward pass. Throughout this assign-

ment, you will be implementing layers in `cs231n/layers.py`. You can re-use your implementations for `affine_forward`, `affine_backward`, `relu_forward`, `relu_backward`, and `softmax_loss` from before. For right now, don't worry about implementing dropout or batch/layer normalization yet, as you will add those features later.

[12]:
```
# Setup cell.
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"

%load_ext autoreload
%autoreload 2

def rel_error(x, y):
    """Returns relative error."""
    return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

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

[13]:
```
# Load the (preprocessed) CIFAR-10 data.
data = get_CIFAR10_data()
for k, v in list(data.items()):
    print(f"{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,)
```

1.1 Initial Loss and Gradient Check

As a sanity check, run the following to check the initial loss and to gradient check the network both with and without regularization. This is a good way to see if the initial losses seem reasonable.

For gradient checking, you should expect to see errors around 1e-7 or less.

[15]:
```
np.random.seed(231)
N, D, H1, H2, C = 2, 15, 20, 30, 10
X = np.random.randn(N, D)
y = np.random.randint(C, size=(N,))

for reg in [0, 3.14]:
    print("Running check with reg = ", reg)
    model = FullyConnectedNet(
        [H1, H2],
        input_dim=D,
        num_classes=C,
        reg=reg,
        weight_scale=5e-2,
        dtype=np.float64
    )

    loss, grads = model.loss(X, y)
    print("Initial loss: ", loss)

    # Most of the errors should be on the order of e-7 or smaller.
    # NOTE: It is fine however to see an error for W2 on the order of e-5
    # for the check when reg = 0.0
    for name in sorted(grads):
        f = lambda _: model.loss(X, y)[0]
        grad_num = eval_numerical_gradient(f, model.params[name],␣
↪verbose=False, h=1e-5)
        print(f"{name} relative error: {rel_error(grad_num, grads[name])}")
```

```
Running check with reg =  0
Initial loss:  2.300479089768492
W1 relative error: 1.0252674471656573e-07
W2 relative error: 2.2120479295080622e-05
W3 relative error: 4.5623278736665505e-07
b1 relative error: 4.6600944653202505e-09
b2 relative error: 2.085654276112763e-09
b3 relative error: 1.689724888469736e-10
Running check with reg =  3.14
Initial loss:  7.052114776533016
W1 relative error: 3.904541941902138e-09
W2 relative error: 6.86942277940646e-08
W3 relative error: 3.483989247437803e-08
b1 relative error: 1.4752427965311745e-08
b2 relative error: 1.4615869332918208e-09
b3 relative error: 1.3200479211447775e-10
```

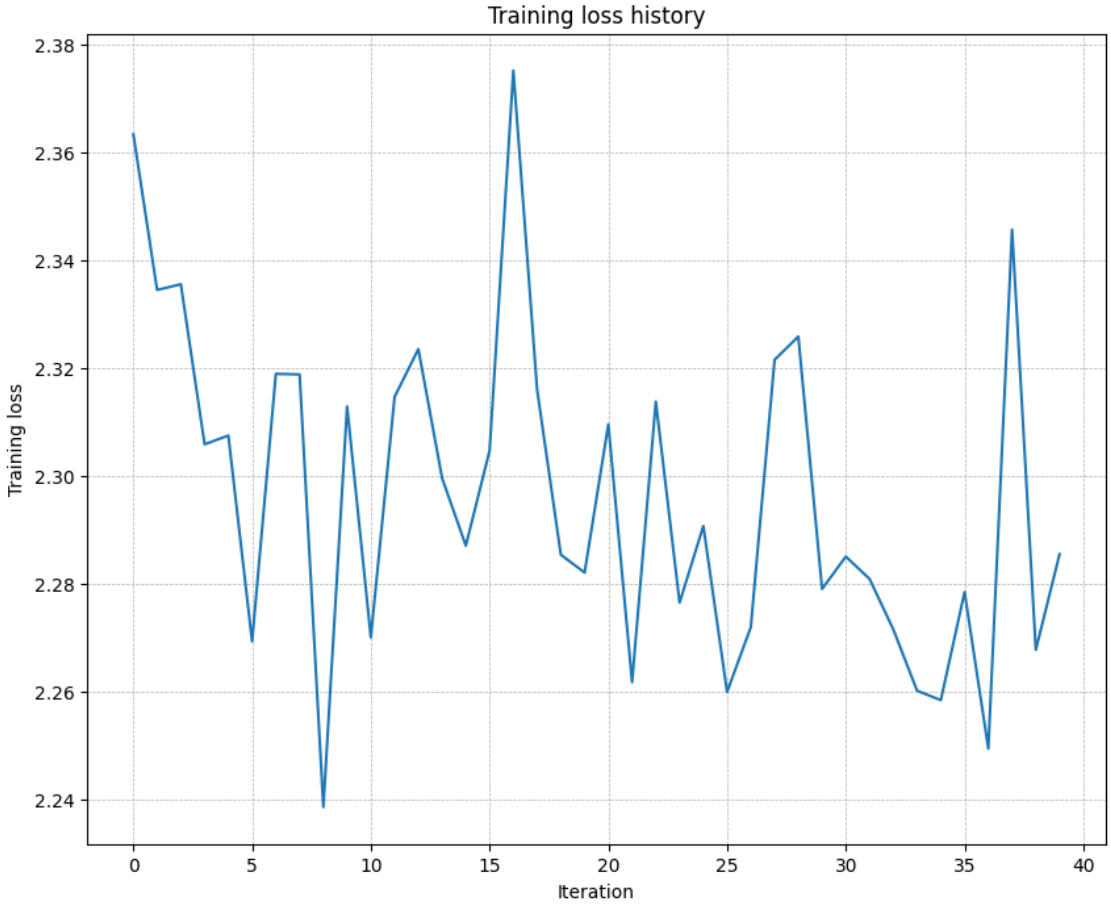
As another sanity check, make sure your network can overfit on a small dataset of 50 images. First, we will try a three-layer network with 100 units in each hidden layer. In the following cell, tweak the **learning rate** and **weight initialization scale** to overfit and achieve 100% training accuracy

within 20 epochs.

[16]:
```
# TODO: Use a three-layer Net to overfit 50 training examples by
# tweaking just the learning rate and initialization scale.

num_train = 50
small_data = {
  "X_train": data["X_train"][:num_train],
  "y_train": data["y_train"][:num_train],
  "X_val": data["X_val"],
  "y_val": data["y_val"],
}

weight_scale = 1e-2   # Experiment with this!
learning_rate = 1e-4  # Experiment with this!


model = FullyConnectedNet(
    [100, 100],
    weight_scale=weight_scale,
    dtype=np.float64
)
solver = Solver(
    model,
    small_data,
    print_every=10,
    num_epochs=20,
    batch_size=25,
    update_rule="sgd",
    optim_config={"learning_rate": learning_rate},
)
solver.train()

plt.plot(solver.loss_history)
plt.title("Training loss history")
plt.xlabel("Iteration")
plt.ylabel("Training loss")
plt.grid(linestyle='--', linewidth=0.5)
plt.show()
```

```
(Iteration 1 / 40) loss: 2.363364
(Epoch 0 / 20) train acc: 0.020000; val_acc: 0.105000
(Epoch 1 / 20) train acc: 0.020000; val_acc: 0.106000
(Epoch 2 / 20) train acc: 0.020000; val_acc: 0.110000
(Epoch 3 / 20) train acc: 0.020000; val_acc: 0.110000
(Epoch 4 / 20) train acc: 0.040000; val_acc: 0.109000
(Epoch 5 / 20) train acc: 0.040000; val_acc: 0.111000
(Iteration 11 / 40) loss: 2.270022
```

```
(Epoch 6 / 20) train acc: 0.040000; val_acc: 0.111000
(Epoch 7 / 20) train acc: 0.060000; val_acc: 0.112000
(Epoch 8 / 20) train acc: 0.060000; val_acc: 0.111000
(Epoch 9 / 20) train acc: 0.040000; val_acc: 0.110000
(Epoch 10 / 20) train acc: 0.040000; val_acc: 0.109000
(Iteration 21 / 40) loss: 2.309562
(Epoch 11 / 20) train acc: 0.060000; val_acc: 0.110000
(Epoch 12 / 20) train acc: 0.060000; val_acc: 0.110000
(Epoch 13 / 20) train acc: 0.060000; val_acc: 0.110000
(Epoch 14 / 20) train acc: 0.060000; val_acc: 0.110000
(Epoch 15 / 20) train acc: 0.060000; val_acc: 0.113000
(Iteration 31 / 40) loss: 2.285026
(Epoch 16 / 20) train acc: 0.060000; val_acc: 0.117000
(Epoch 17 / 20) train acc: 0.080000; val_acc: 0.113000
(Epoch 18 / 20) train acc: 0.080000; val_acc: 0.118000
(Epoch 19 / 20) train acc: 0.100000; val_acc: 0.118000
(Epoch 20 / 20) train acc: 0.100000; val_acc: 0.120000
```



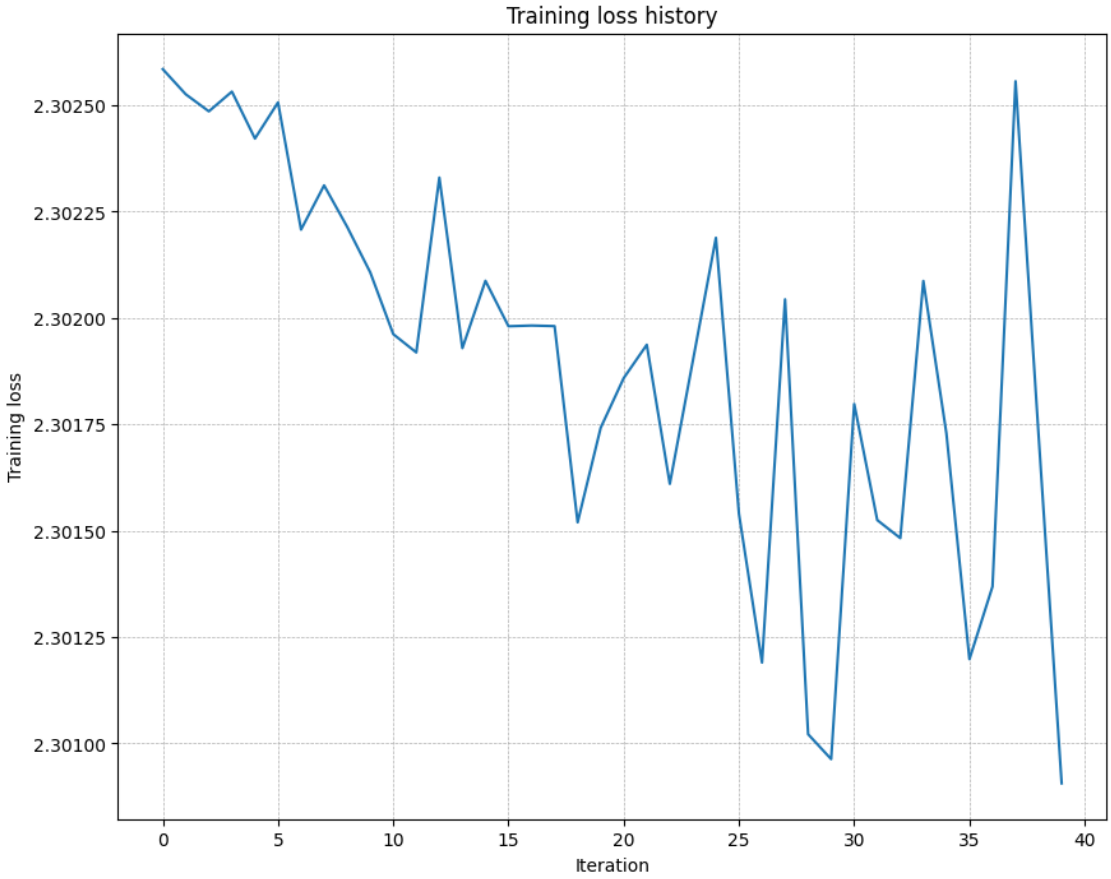
Now, try to use a five-layer network with 100 units on each layer to overfit on 50 training examples.

Again, you will have to adjust the learning rate and weight initialization scale, but you should be able to achieve 100% training accuracy within 20 epochs.

[17]:
```
# TODO: Use a five-layer Net to overfit 50 training examples by
# tweaking just the learning rate and initialization scale.

num_train = 50
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

learning_rate = 2e-3  # Experiment with this!
weight_scale = 1e-5   # Experiment with this!


model = FullyConnectedNet(
    [100, 100, 100, 100],
    weight_scale=weight_scale,
    dtype=np.float64
)
solver = Solver(
    model,
    small_data,
    print_every=10,
    num_epochs=20,
    batch_size=25,
    update_rule='sgd',
    optim_config={'learning_rate': learning_rate},
)
solver.train()

plt.plot(solver.loss_history)
plt.title('Training loss history')
plt.xlabel('Iteration')
plt.ylabel('Training loss')
plt.grid(linestyle='--', linewidth=0.5)
plt.show()
```

```
(Iteration 1 / 40) loss: 2.302585
(Epoch 0 / 20) train acc: 0.160000; val_acc: 0.079000
(Epoch 1 / 20) train acc: 0.160000; val_acc: 0.079000
(Epoch 2 / 20) train acc: 0.160000; val_acc: 0.079000
(Epoch 3 / 20) train acc: 0.160000; val_acc: 0.079000
(Epoch 4 / 20) train acc: 0.160000; val_acc: 0.079000
(Epoch 5 / 20) train acc: 0.160000; val_acc: 0.079000
```

```
(Iteration 11 / 40) loss: 2.301962
(Epoch 6 / 20) train acc: 0.160000; val_acc: 0.079000
(Epoch 7 / 20) train acc: 0.160000; val_acc: 0.079000
(Epoch 8 / 20) train acc: 0.160000; val_acc: 0.079000
(Epoch 9 / 20) train acc: 0.160000; val_acc: 0.079000
(Epoch 10 / 20) train acc: 0.160000; val_acc: 0.079000
(Iteration 21 / 40) loss: 2.301859
(Epoch 11 / 20) train acc: 0.160000; val_acc: 0.079000
(Epoch 12 / 20) train acc: 0.160000; val_acc: 0.079000
(Epoch 13 / 20) train acc: 0.160000; val_acc: 0.079000
(Epoch 14 / 20) train acc: 0.160000; val_acc: 0.079000
(Epoch 15 / 20) train acc: 0.160000; val_acc: 0.079000
(Iteration 31 / 40) loss: 2.301798
(Epoch 16 / 20) train acc: 0.160000; val_acc: 0.079000
(Epoch 17 / 20) train acc: 0.160000; val_acc: 0.079000
(Epoch 18 / 20) train acc: 0.160000; val_acc: 0.079000
(Epoch 19 / 20) train acc: 0.160000; val_acc: 0.079000
(Epoch 20 / 20) train acc: 0.160000; val_acc: 0.079000
```



1.2 Inline Question 1:

Did you notice anything about the comparative difficulty of training the three-layer network vs. training the five-layer network? In particular, based on your experience, which network seemed more sensitive to the initialization scale? Why do you think that is the case?

1.3 Answer:

Five-layer *RELU*

2 Update rules

So far we have used vanilla stochastic gradient descent (SGD) as our update rule. More sophisticated update rules can make it easier to train deep networks. We will implement a few of the most commonly used update rules and compare them to vanilla SGD.

2.1 SGD+Momentum

Stochastic gradient descent with momentum is a widely used update rule that tends to make deep networks converge faster than vanilla stochastic gradient descent. See the Momentum Update section at http://cs231n.github.io/neural-networks-3/#sgd for more information.

Open the file `cs231n/optim.py` and read the documentation at the top of the file to make sure you understand the API. Implement the SGD+momentum update rule in the function `sgd_momentum` and run the following to check your implementation. You should see errors less than e-8.

[19]:
```
from cs231n.optim import sgd_momentum

N, D = 4, 5
w = np.linspace(-0.4, 0.6, num=N*D).reshape(N, D)
dw = np.linspace(-0.6, 0.4, num=N*D).reshape(N, D)
v = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)

config = {"learning_rate": 1e-3, "velocity": v}
next_w, _ = sgd_momentum(w, dw, config=config)

expected_next_w = np.asarray([
  [ 0.1406,      0.20738947,  0.27417895,  0.34096842,  0.40775789],
  [ 0.47454737,  0.54133684,  0.60812632,  0.67491579,  0.74170526],
  [ 0.80849474,  0.87528421,  0.94207368,  1.00886316,  1.07565263],
  [ 1.14244211,  1.20923158,  1.27602105,  1.34281053,  1.4096    ]])
expected_velocity = np.asarray([
  [ 0.5406,      0.55475789,  0.56891579, 0.58307368,  0.59723158],
  [ 0.61138947,  0.62554737,  0.63970526,  0.65386316,  0.66802105],
  [ 0.68217895,  0.69633684,  0.71049474,  0.72465263,  0.73881053],
  [ 0.75296842,  0.76712632,  0.78128421,  0.79544211,  0.8096    ]])

# Should see relative errors around e-8 or less
print("next_w error: ", rel_error(next_w, expected_next_w))
```

```
print("velocity error: ", rel_error(expected_velocity, config["velocity"]))
```

```
next_w error:  8.882347033505819e-09
velocity error:  4.269287743278663e-09
```

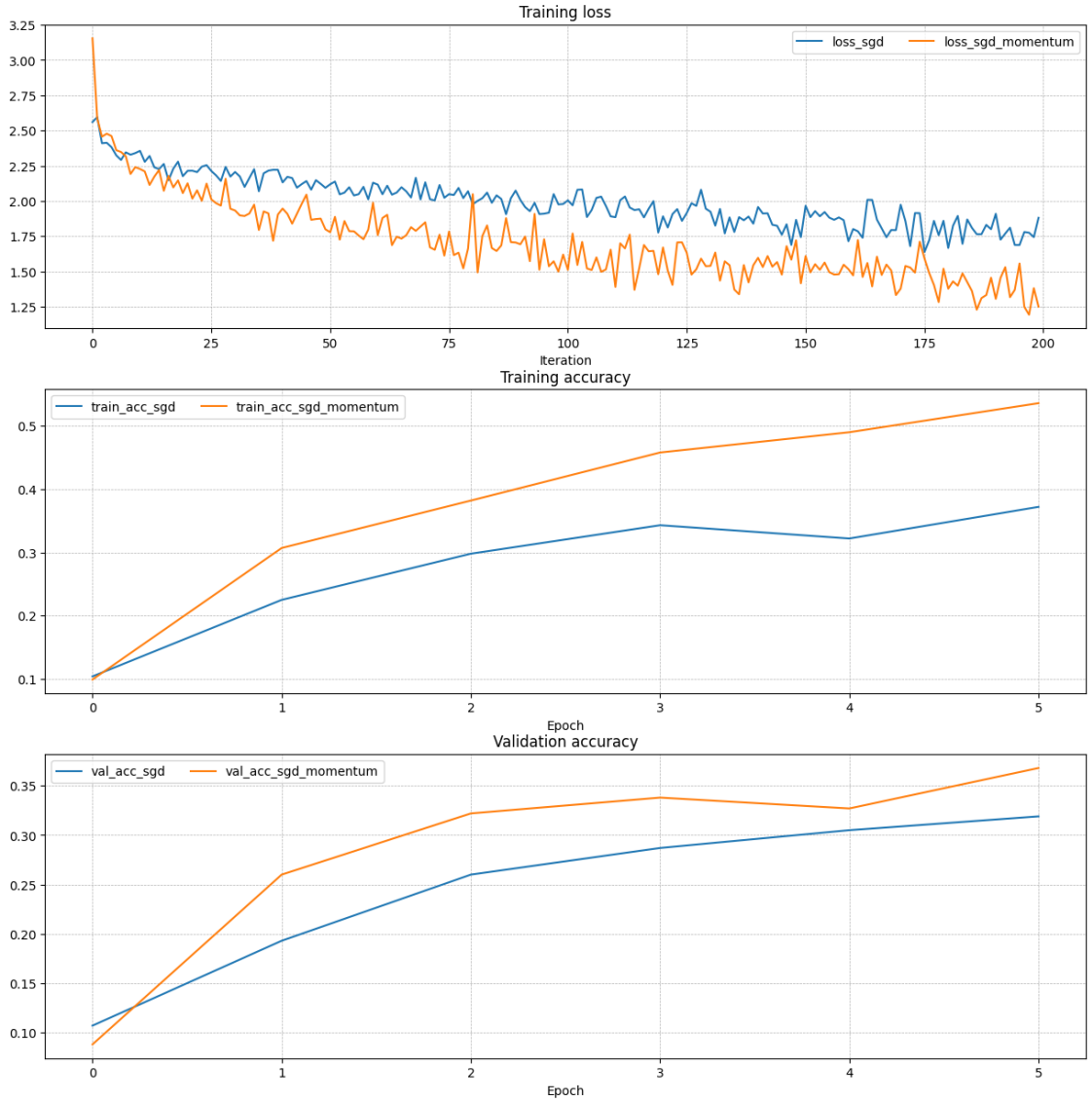
Once you have done so, run the following to train a six-layer network with both SGD and SGD+momentum. You should see the SGD+momentum update rule converge faster.

[20]:
```
num_train = 4000
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

solvers = {}

for update_rule in ['sgd', 'sgd_momentum']:
    print('Running with ', update_rule)
    model = FullyConnectedNet(
        [100, 100, 100, 100, 100],
        weight_scale=5e-2
    )

    solver = Solver(
        model,
        small_data,
        num_epochs=5,
        batch_size=100,
        update_rule=update_rule,
        optim_config={'learning_rate': 5e-3},
        verbose=True,
    )
    solvers[update_rule] = solver
    solver.train()

fig, axes = plt.subplots(3, 1, figsize=(15, 15))

axes[0].set_title('Training loss')
axes[0].set_xlabel('Iteration')
axes[1].set_title('Training accuracy')
axes[1].set_xlabel('Epoch')
axes[2].set_title('Validation accuracy')
axes[2].set_xlabel('Epoch')

for update_rule, solver in solvers.items():
    axes[0].plot(solver.loss_history, label=f"loss_{update_rule}")
```

```
    axes[1].plot(solver.train_acc_history, label=f"train_acc_{update_rule}")
    axes[2].plot(solver.val_acc_history, label=f"val_acc_{update_rule}")

for ax in axes:
    ax.legend(loc="best", ncol=4)
    ax.grid(linestyle='--', linewidth=0.5)

plt.show()
```

```
Running with  sgd
(Iteration 1 / 200) loss: 2.559978
(Epoch 0 / 5) train acc: 0.104000; val_acc: 0.107000
(Iteration 11 / 200) loss: 2.356070
(Iteration 21 / 200) loss: 2.214091
(Iteration 31 / 200) loss: 2.205928
(Epoch 1 / 5) train acc: 0.225000; val_acc: 0.193000
(Iteration 41 / 200) loss: 2.132095
(Iteration 51 / 200) loss: 2.118950
(Iteration 61 / 200) loss: 2.116443
(Iteration 71 / 200) loss: 2.132549
(Epoch 2 / 5) train acc: 0.298000; val_acc: 0.260000
(Iteration 81 / 200) loss: 1.977227
(Iteration 91 / 200) loss: 2.007528
(Iteration 101 / 200) loss: 2.004762
(Iteration 111 / 200) loss: 1.885342
(Epoch 3 / 5) train acc: 0.343000; val_acc: 0.287000
(Iteration 121 / 200) loss: 1.891516
(Iteration 131 / 200) loss: 1.923677
(Iteration 141 / 200) loss: 1.957743
(Iteration 151 / 200) loss: 1.966736
(Epoch 4 / 5) train acc: 0.322000; val_acc: 0.305000
(Iteration 161 / 200) loss: 1.801483
(Iteration 171 / 200) loss: 1.973780
(Iteration 181 / 200) loss: 1.666573
(Iteration 191 / 200) loss: 1.909494
(Epoch 5 / 5) train acc: 0.372000; val_acc: 0.319000
Running with  sgd_momentum
(Iteration 1 / 200) loss: 3.153778
(Epoch 0 / 5) train acc: 0.099000; val_acc: 0.088000
(Iteration 11 / 200) loss: 2.227203
(Iteration 21 / 200) loss: 2.125706
(Iteration 31 / 200) loss: 1.932695
(Epoch 1 / 5) train acc: 0.307000; val_acc: 0.260000
(Iteration 41 / 200) loss: 1.946488
(Iteration 51 / 200) loss: 1.778583
(Iteration 61 / 200) loss: 1.758119
(Iteration 71 / 200) loss: 1.849137
```

```
(Epoch 2 / 5) train acc: 0.382000; val_acc: 0.322000
(Iteration 81 / 200) loss: 2.048671
(Iteration 91 / 200) loss: 1.693223
(Iteration 101 / 200) loss: 1.511693
(Iteration 111 / 200) loss: 1.390754
(Epoch 3 / 5) train acc: 0.458000; val_acc: 0.338000
(Iteration 121 / 200) loss: 1.670614
(Iteration 131 / 200) loss: 1.540271
(Iteration 141 / 200) loss: 1.597365
(Iteration 151 / 200) loss: 1.609851
(Epoch 4 / 5) train acc: 0.490000; val_acc: 0.327000
(Iteration 161 / 200) loss: 1.472687
(Iteration 171 / 200) loss: 1.378620
(Iteration 181 / 200) loss: 1.378175
(Iteration 191 / 200) loss: 1.305934
(Epoch 5 / 5) train acc: 0.536000; val_acc: 0.368000
```



2.2 RMSProp and Adam

RMSProp [1] and Adam [2] are update rules that set per-parameter learning rates by using a running average of the second moments of gradients.

In the file `cs231n/optim.py`, implement the RMSProp update rule in the `rmsprop` function and implement the Adam update rule in the `adam` function, and check your implementations using the tests below.

NOTE: Please implement the *complete* Adam update rule (with the bias correction mechanism), not the first simplified version mentioned in the course notes.

[1] Tijmen Tieleman and Geoffrey Hinton. "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude." COURSERA: Neural Networks for Machine Learning 4 (2012).

[2] Diederik Kingma and Jimmy Ba, "Adam: A Method for Stochastic Optimization", ICLR 2015.

[21]:
```
# Test RMSProp implementation
from cs231n.optim import rmsprop

N, D = 4, 5
w = np.linspace(-0.4, 0.6, num=N*D).reshape(N, D)
dw = np.linspace(-0.6, 0.4, num=N*D).reshape(N, D)
cache = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)

config = {'learning_rate': 1e-2, 'cache': cache}
next_w, _ = rmsprop(w, dw, config=config)

expected_next_w = np.asarray([
  [-0.39223849, -0.34037513, -0.28849239, -0.23659121, -0.18467247],
  [-0.132737,   -0.08078555, -0.02881884,  0.02316247,  0.07515774],
  [ 0.12716641,  0.17918792,  0.23122175,  0.28326742,  0.33532447],
  [ 0.38739248,  0.43947102,  0.49155973,  0.54365823,  0.59576619]])
expected_cache = np.asarray([
  [ 0.5976,      0.6126277,   0.6277108,   0.64284931,  0.65804321],
  [ 0.67329252,  0.68859723,  0.70395734,  0.71937285,  0.73484377],
  [ 0.75037008,  0.7659518,   0.78158892,  0.79728144,  0.81302936],
  [ 0.82883269,  0.84469141,  0.86060554,  0.87657507,  0.8926    ]])

# You should see relative errors around e-7 or less
print('next_w error: ', rel_error(expected_next_w, next_w))
print('cache error: ', rel_error(expected_cache, config['cache']))
```

```
next_w error:  9.524687511038133e-08
cache error:  2.6477955807156126e-09
```

[22]:
```
# Test Adam implementation
from cs231n.optim import adam

N, D = 4, 5
w = np.linspace(-0.4, 0.6, num=N*D).reshape(N, D)
dw = np.linspace(-0.6, 0.4, num=N*D).reshape(N, D)
m = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)
v = np.linspace(0.7, 0.5, num=N*D).reshape(N, D)

config = {'learning_rate': 1e-2, 'm': m, 'v': v, 't': 5}
next_w, _ = adam(w, dw, config=config)

expected_next_w = np.asarray([
  [-0.40094747, -0.34836187, -0.29577703, -0.24319299, -0.19060977],
  [-0.1380274,  -0.08544591, -0.03286534,  0.01971428,  0.0722929],
  [ 0.1248705,   0.17744702,  0.23002243,  0.28259667,  0.33516969],
  [ 0.38774145,  0.44031188,  0.49288093,  0.54544852,  0.59801459]])
```

```
expected_v = np.asarray([
  [ 0.69966,     0.68908382,  0.67851319,  0.66794809,  0.65738853,],
  [ 0.64683452,  0.63628604,  0.6257431,   0.61520571,  0.60467385,],
  [ 0.59414753,  0.58362676,  0.57311152,  0.56260183,  0.55209767,],
  [ 0.54159906,  0.53110598,  0.52061845,  0.51013645,  0.49966,   ]])
expected_m = np.asarray([
  [ 0.48,        0.49947368,  0.51894737,  0.53842105,  0.55789474],
  [ 0.57736842,  0.59684211,  0.61631579,  0.63578947,  0.65526316],
  [ 0.67473684,  0.69421053,  0.71368421,  0.73315789,  0.75263158],
  [ 0.77210526,  0.79157895,  0.81105263,  0.83052632,  0.85      ]])

# You should see relative errors around e-7 or less
print('next_w error: ', rel_error(expected_next_w, next_w))
print('v error: ', rel_error(expected_v, config['v']))
print('m error: ', rel_error(expected_m, config['m']))
```

```
next_w error:  1.139887467333134e-07
v error:  4.208314038113071e-09
m error:  4.214963193114416e-09
```

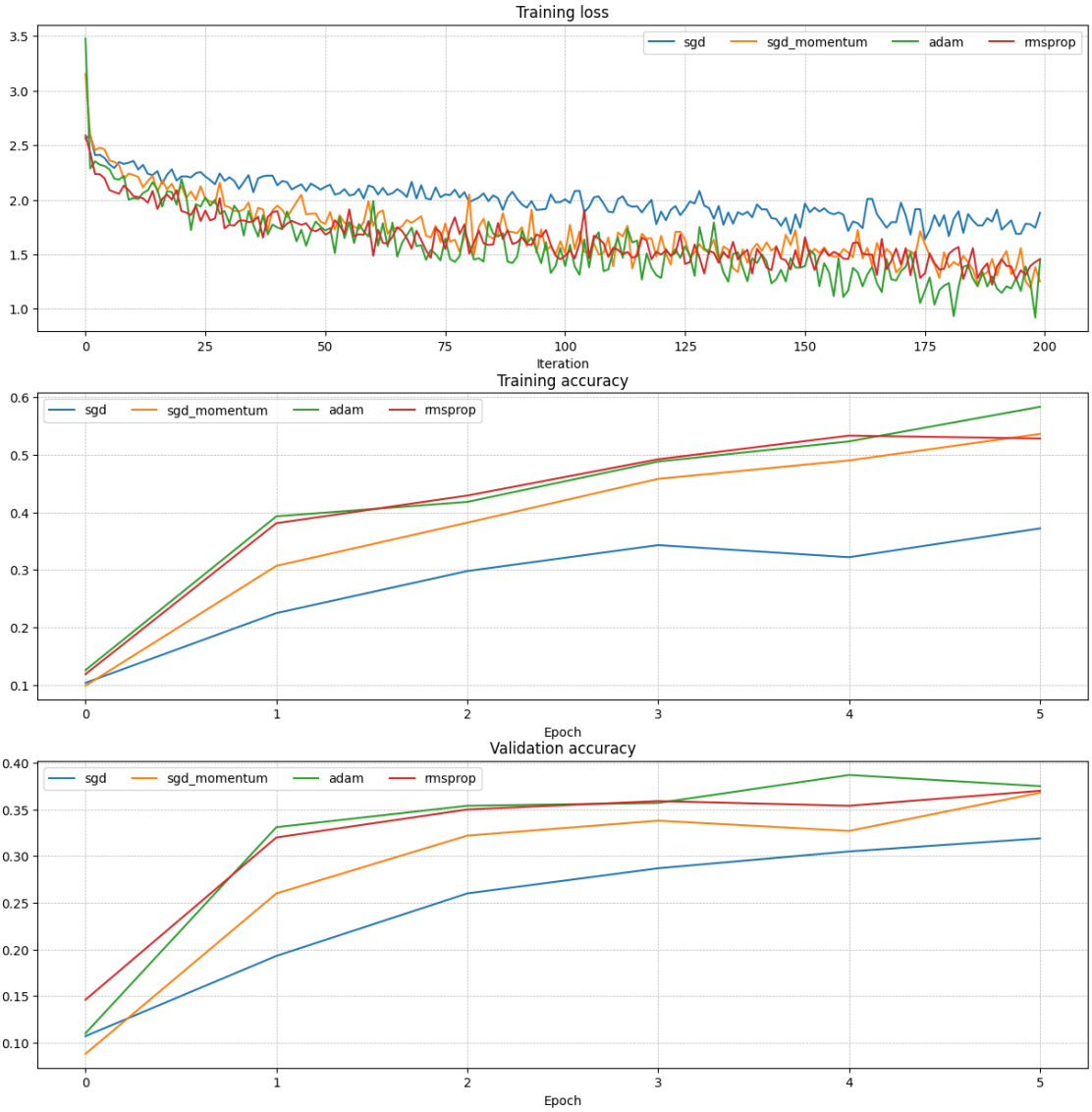
Once you have debugged your RMSProp and Adam implementations, run the following to train a pair of deep networks using these new update rules:

[23]:
```
learning_rates = {'rmsprop': 1e-4, 'adam': 1e-3}
for update_rule in ['adam', 'rmsprop']:
    print('Running with ', update_rule)
    model = FullyConnectedNet(
        [100, 100, 100, 100, 100],
        weight_scale=5e-2
    )
    solver = Solver(
        model,
        small_data,
        num_epochs=5,
        batch_size=100,
        update_rule=update_rule,
        optim_config={'learning_rate': learning_rates[update_rule]},
        verbose=True
    )
    solvers[update_rule] = solver
    solver.train()
    print()

fig, axes = plt.subplots(3, 1, figsize=(15, 15))

axes[0].set_title('Training loss')
axes[0].set_xlabel('Iteration')
axes[1].set_title('Training accuracy')
```

```
axes[1].set_xlabel('Epoch')
axes[2].set_title('Validation accuracy')
axes[2].set_xlabel('Epoch')

for update_rule, solver in solvers.items():
    axes[0].plot(solver.loss_history, label=f"{update_rule}")
    axes[1].plot(solver.train_acc_history, label=f"{update_rule}")
    axes[2].plot(solver.val_acc_history, label=f"{update_rule}")

for ax in axes:
    ax.legend(loc='best', ncol=4)
    ax.grid(linestyle='--', linewidth=0.5)

plt.show()
```

```
Running with  adam
(Iteration 1 / 200) loss: 3.476928
(Epoch 0 / 5) train acc: 0.126000; val_acc: 0.110000
(Iteration 11 / 200) loss: 2.015214
(Iteration 21 / 200) loss: 2.190157
(Iteration 31 / 200) loss: 1.785010
(Epoch 1 / 5) train acc: 0.393000; val_acc: 0.331000
(Iteration 41 / 200) loss: 1.746502
(Iteration 51 / 200) loss: 1.719389
(Iteration 61 / 200) loss: 1.988491
(Iteration 71 / 200) loss: 1.582892
(Epoch 2 / 5) train acc: 0.418000; val_acc: 0.354000
(Iteration 81 / 200) loss: 1.593839
(Iteration 91 / 200) loss: 1.479027
(Iteration 101 / 200) loss: 1.393083
(Iteration 111 / 200) loss: 1.527003
(Epoch 3 / 5) train acc: 0.488000; val_acc: 0.357000
(Iteration 121 / 200) loss: 1.282885
(Iteration 131 / 200) loss: 1.506014
(Iteration 141 / 200) loss: 1.420248
(Iteration 151 / 200) loss: 1.392918
(Epoch 4 / 5) train acc: 0.523000; val_acc: 0.387000
(Iteration 161 / 200) loss: 1.375995
(Iteration 171 / 200) loss: 1.344928
(Iteration 181 / 200) loss: 1.236344
(Iteration 191 / 200) loss: 1.186700
(Epoch 5 / 5) train acc: 0.583000; val_acc: 0.375000

Running with  rmsprop
(Iteration 1 / 200) loss: 2.589166
(Epoch 0 / 5) train acc: 0.119000; val_acc: 0.146000
(Iteration 11 / 200) loss: 2.032921
```

```
(Iteration 21 / 200) loss: 1.897278
(Iteration 31 / 200) loss: 1.770793
(Epoch 1 / 5) train acc: 0.381000; val_acc: 0.320000
(Iteration 41 / 200) loss: 1.895731
(Iteration 51 / 200) loss: 1.681091
(Iteration 61 / 200) loss: 1.487204
(Iteration 71 / 200) loss: 1.629973
(Epoch 2 / 5) train acc: 0.429000; val_acc: 0.350000
(Iteration 81 / 200) loss: 1.506686
(Iteration 91 / 200) loss: 1.610742
(Iteration 101 / 200) loss: 1.486124
(Iteration 111 / 200) loss: 1.559454
(Epoch 3 / 5) train acc: 0.492000; val_acc: 0.359000
(Iteration 121 / 200) loss: 1.496860
(Iteration 131 / 200) loss: 1.531552
(Iteration 141 / 200) loss: 1.550195
(Iteration 151 / 200) loss: 1.657838
(Epoch 4 / 5) train acc: 0.533000; val_acc: 0.354000
(Iteration 161 / 200) loss: 1.603105
(Iteration 171 / 200) loss: 1.408064
(Iteration 181 / 200) loss: 1.504707
(Iteration 191 / 200) loss: 1.385212
(Epoch 5 / 5) train acc: 0.528000; val_acc: 0.370000
```



2.3 Inline Question 2:

AdaGrad, like Adam, is a per-parameter optimization method that uses the following update rule:

```
cache += dw**2
w += - learning_rate * dw / (np.sqrt(cache) + eps)
```

John notices that when he was training a network with AdaGrad that the updates became very small, and that his network was learning slowly. Using your knowledge of the AdaGrad update rule, why do you think the updates would become very small? Would Adam have the same issue?

2.4 Answer:

AdaGrad *dw* *Adam* *AdaGrad*

3 Train a Good Model!

Train the best fully connected model that you can on CIFAR-10, storing your best model in the `best_model` variable. We require you to get at least 50% accuracy on the validation set using a fully connected network.

If you are careful it should be possible to get accuracies above 55%, but we don't require it for this part and won't assign extra credit for doing so. Later in the next assignment, we will ask you to train the best convolutional network that you can on CIFAR-10, and we would prefer that you spend your effort working on convolutional networks rather than fully connected networks.

Note: In the next assignment, you will learn techniques like BatchNormalization and Dropout which can help you train powerful models.

[26]:
```
best_model = None

################################################################################
# TODO: Train the best FullyConnectedNet that you can on CIFAR-10. You might   #
# find batch/layer normalization and dropout useful. Store your best model in  #
# the best_model variable.                                                     #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
from cs231n.classifiers.fc_net import FullyConnectedNet
from cs231n.solver import Solver
best_val_acc = -1
model = FullyConnectedNet(
      [100, 100, 100, 100, 100],
      weight_scale = 1e-2
)

solver = Solver(
    model,
    data,
    num_epochs =  20,
    batch_size = 100,
    update_rule = 'adam',
    optim_config = {'learning_rate': 1e-3},
    verbose = True,
    lr_decay = 0.95
)

solver.train()

best_val_acc = solver.val_acc_history[-1]
best_model = model


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

```
#                                 END OF YOUR CODE                             #
################################################################################
```

```
(Iteration 1 / 9800) loss: 2.302578
(Epoch 0 / 20) train acc: 0.101000; val_acc: 0.090000
(Iteration 11 / 9800) loss: 2.303174
(Iteration 21 / 9800) loss: 2.272730
(Iteration 31 / 9800) loss: 2.178746
(Iteration 41 / 9800) loss: 2.141502
(Iteration 51 / 9800) loss: 2.216677
(Iteration 61 / 9800) loss: 1.983872
(Iteration 71 / 9800) loss: 2.088522
(Iteration 81 / 9800) loss: 2.167604
(Iteration 91 / 9800) loss: 2.104162
(Iteration 101 / 9800) loss: 1.919975
(Iteration 111 / 9800) loss: 2.094265
(Iteration 121 / 9800) loss: 1.940261
(Iteration 131 / 9800) loss: 2.058716
(Iteration 141 / 9800) loss: 2.022792
(Iteration 151 / 9800) loss: 1.970152
(Iteration 161 / 9800) loss: 2.023720
(Iteration 171 / 9800) loss: 2.038835
(Iteration 181 / 9800) loss: 2.004002
(Iteration 191 / 9800) loss: 2.086852
(Iteration 201 / 9800) loss: 1.965657
(Iteration 211 / 9800) loss: 1.969418
(Iteration 221 / 9800) loss: 1.920283
(Iteration 231 / 9800) loss: 1.970267
(Iteration 241 / 9800) loss: 2.109726
(Iteration 251 / 9800) loss: 1.995159
(Iteration 261 / 9800) loss: 2.062294
(Iteration 271 / 9800) loss: 1.939872
(Iteration 281 / 9800) loss: 1.972446
(Iteration 291 / 9800) loss: 1.801423
(Iteration 301 / 9800) loss: 1.923327
(Iteration 311 / 9800) loss: 1.786081
(Iteration 321 / 9800) loss: 2.026001
(Iteration 331 / 9800) loss: 1.874522
(Iteration 341 / 9800) loss: 1.977579
(Iteration 351 / 9800) loss: 1.921376
(Iteration 361 / 9800) loss: 1.877561
(Iteration 371 / 9800) loss: 1.763454
(Iteration 381 / 9800) loss: 1.924350
(Iteration 391 / 9800) loss: 1.777274
(Iteration 401 / 9800) loss: 1.837448
(Iteration 411 / 9800) loss: 1.949123
(Iteration 421 / 9800) loss: 1.795889
```

```
(Iteration 431 / 9800) loss: 1.793548
(Iteration 441 / 9800) loss: 1.806672
(Iteration 451 / 9800) loss: 1.969171
(Iteration 461 / 9800) loss: 1.674999
(Iteration 471 / 9800) loss: 1.741067
(Iteration 481 / 9800) loss: 1.873013
(Epoch 1 / 20) train acc: 0.322000; val_acc: 0.328000
(Iteration 491 / 9800) loss: 1.946753
(Iteration 501 / 9800) loss: 1.802623
(Iteration 511 / 9800) loss: 1.990905
(Iteration 521 / 9800) loss: 1.852602
(Iteration 531 / 9800) loss: 1.852172
(Iteration 541 / 9800) loss: 1.868615
(Iteration 551 / 9800) loss: 1.672970
(Iteration 561 / 9800) loss: 1.810874
(Iteration 571 / 9800) loss: 1.811065
(Iteration 581 / 9800) loss: 1.672266
(Iteration 591 / 9800) loss: 1.937537
(Iteration 601 / 9800) loss: 1.714483
(Iteration 611 / 9800) loss: 1.573656
(Iteration 621 / 9800) loss: 1.629332
(Iteration 631 / 9800) loss: 1.740932
(Iteration 641 / 9800) loss: 1.823287
(Iteration 651 / 9800) loss: 1.716662
(Iteration 661 / 9800) loss: 1.760932
(Iteration 671 / 9800) loss: 1.860602
(Iteration 681 / 9800) loss: 1.819198
(Iteration 691 / 9800) loss: 1.642353
(Iteration 701 / 9800) loss: 1.637041
(Iteration 711 / 9800) loss: 1.690911
(Iteration 721 / 9800) loss: 1.600859
(Iteration 731 / 9800) loss: 1.616280
(Iteration 741 / 9800) loss: 1.435730
(Iteration 751 / 9800) loss: 1.701768
(Iteration 761 / 9800) loss: 1.665217
(Iteration 771 / 9800) loss: 1.837285
(Iteration 781 / 9800) loss: 1.676383
(Iteration 791 / 9800) loss: 1.724985
(Iteration 801 / 9800) loss: 1.601056
(Iteration 811 / 9800) loss: 1.442675
(Iteration 821 / 9800) loss: 1.513286
(Iteration 831 / 9800) loss: 1.680019
(Iteration 841 / 9800) loss: 1.490679
(Iteration 851 / 9800) loss: 1.603760
(Iteration 861 / 9800) loss: 1.657655
(Iteration 871 / 9800) loss: 1.665616
(Iteration 881 / 9800) loss: 1.712839
(Iteration 891 / 9800) loss: 1.613931
```

```
(Iteration 901 / 9800) loss: 1.558941
(Iteration 911 / 9800) loss: 1.605365
(Iteration 921 / 9800) loss: 1.632145
(Iteration 931 / 9800) loss: 1.704846
(Iteration 941 / 9800) loss: 1.592737
(Iteration 951 / 9800) loss: 1.610151
(Iteration 961 / 9800) loss: 1.374480
(Iteration 971 / 9800) loss: 1.787455
(Epoch 2 / 20) train acc: 0.442000; val_acc: 0.425000
(Iteration 981 / 9800) loss: 1.615311
(Iteration 991 / 9800) loss: 1.498027
(Iteration 1001 / 9800) loss: 1.426605
(Iteration 1011 / 9800) loss: 1.556226
(Iteration 1021 / 9800) loss: 1.504574
(Iteration 1031 / 9800) loss: 1.562550
(Iteration 1041 / 9800) loss: 1.460419
(Iteration 1051 / 9800) loss: 1.651307
(Iteration 1061 / 9800) loss: 1.535204
(Iteration 1071 / 9800) loss: 1.370677
(Iteration 1081 / 9800) loss: 1.605669
(Iteration 1091 / 9800) loss: 1.389009
(Iteration 1101 / 9800) loss: 1.550484
(Iteration 1111 / 9800) loss: 1.563158
(Iteration 1121 / 9800) loss: 1.437367
(Iteration 1131 / 9800) loss: 1.472833
(Iteration 1141 / 9800) loss: 1.443579
(Iteration 1151 / 9800) loss: 1.558596
(Iteration 1161 / 9800) loss: 1.440940
(Iteration 1171 / 9800) loss: 1.411787
(Iteration 1181 / 9800) loss: 1.509360
(Iteration 1191 / 9800) loss: 1.403898
(Iteration 1201 / 9800) loss: 1.604250
(Iteration 1211 / 9800) loss: 1.498884
(Iteration 1221 / 9800) loss: 1.449226
(Iteration 1231 / 9800) loss: 1.450262
(Iteration 1241 / 9800) loss: 1.546362
(Iteration 1251 / 9800) loss: 1.331257
(Iteration 1261 / 9800) loss: 1.597733
(Iteration 1271 / 9800) loss: 1.436199
(Iteration 1281 / 9800) loss: 1.158712
(Iteration 1291 / 9800) loss: 1.428906
(Iteration 1301 / 9800) loss: 1.502051
(Iteration 1311 / 9800) loss: 1.642509
(Iteration 1321 / 9800) loss: 1.341505
(Iteration 1331 / 9800) loss: 1.374521
(Iteration 1341 / 9800) loss: 1.506235
(Iteration 1351 / 9800) loss: 1.538957
(Iteration 1361 / 9800) loss: 1.496529
```

```
(Iteration 1371 / 9800) loss: 1.488742
(Iteration 1381 / 9800) loss: 1.327814
(Iteration 1391 / 9800) loss: 1.352726
(Iteration 1401 / 9800) loss: 1.705031
(Iteration 1411 / 9800) loss: 1.318936
(Iteration 1421 / 9800) loss: 1.340073
(Iteration 1431 / 9800) loss: 1.440398
(Iteration 1441 / 9800) loss: 1.308234
(Iteration 1451 / 9800) loss: 1.495650
(Iteration 1461 / 9800) loss: 1.454402
(Epoch 3 / 20) train acc: 0.461000; val_acc: 0.468000
(Iteration 1471 / 9800) loss: 1.439972
(Iteration 1481 / 9800) loss: 1.423229
(Iteration 1491 / 9800) loss: 1.547255
(Iteration 1501 / 9800) loss: 1.617513
(Iteration 1511 / 9800) loss: 1.344420
(Iteration 1521 / 9800) loss: 1.413199
(Iteration 1531 / 9800) loss: 1.530280
(Iteration 1541 / 9800) loss: 1.434449
(Iteration 1551 / 9800) loss: 1.349748
(Iteration 1561 / 9800) loss: 1.530790
(Iteration 1571 / 9800) loss: 1.566660
(Iteration 1581 / 9800) loss: 1.449620
(Iteration 1591 / 9800) loss: 1.740659
(Iteration 1601 / 9800) loss: 1.315003
(Iteration 1611 / 9800) loss: 1.382945
(Iteration 1621 / 9800) loss: 1.486838
(Iteration 1631 / 9800) loss: 1.240986
(Iteration 1641 / 9800) loss: 1.333017
(Iteration 1651 / 9800) loss: 1.492741
(Iteration 1661 / 9800) loss: 1.592025
(Iteration 1671 / 9800) loss: 1.267500
(Iteration 1681 / 9800) loss: 1.341483
(Iteration 1691 / 9800) loss: 1.550567
(Iteration 1701 / 9800) loss: 1.534408
(Iteration 1711 / 9800) loss: 1.443660
(Iteration 1721 / 9800) loss: 1.296558
(Iteration 1731 / 9800) loss: 1.268237
(Iteration 1741 / 9800) loss: 1.367527
(Iteration 1751 / 9800) loss: 1.543437
(Iteration 1761 / 9800) loss: 1.473308
(Iteration 1771 / 9800) loss: 1.279937
(Iteration 1781 / 9800) loss: 1.447187
(Iteration 1791 / 9800) loss: 1.471014
(Iteration 1801 / 9800) loss: 1.372244
(Iteration 1811 / 9800) loss: 1.306223
(Iteration 1821 / 9800) loss: 1.244234
(Iteration 1831 / 9800) loss: 1.524481
```

```
(Iteration 1841 / 9800) loss: 1.464421
(Iteration 1851 / 9800) loss: 1.402268
(Iteration 1861 / 9800) loss: 1.139554
(Iteration 1871 / 9800) loss: 1.341252
(Iteration 1881 / 9800) loss: 1.471263
(Iteration 1891 / 9800) loss: 1.211309
(Iteration 1901 / 9800) loss: 1.242686
(Iteration 1911 / 9800) loss: 1.202983
(Iteration 1921 / 9800) loss: 1.493550
(Iteration 1931 / 9800) loss: 1.276823
(Iteration 1941 / 9800) loss: 1.253020
(Iteration 1951 / 9800) loss: 1.412980
(Epoch 4 / 20) train acc: 0.526000; val_acc: 0.478000
(Iteration 1961 / 9800) loss: 1.318442
(Iteration 1971 / 9800) loss: 1.490313
(Iteration 1981 / 9800) loss: 1.369841
(Iteration 1991 / 9800) loss: 1.346716
(Iteration 2001 / 9800) loss: 1.494769
(Iteration 2011 / 9800) loss: 1.425250
(Iteration 2021 / 9800) loss: 1.359716
(Iteration 2031 / 9800) loss: 1.450192
(Iteration 2041 / 9800) loss: 1.339178
(Iteration 2051 / 9800) loss: 1.255157
(Iteration 2061 / 9800) loss: 1.360415
(Iteration 2071 / 9800) loss: 1.362423
(Iteration 2081 / 9800) loss: 1.340479
(Iteration 2091 / 9800) loss: 1.400368
(Iteration 2101 / 9800) loss: 1.427034
(Iteration 2111 / 9800) loss: 1.313881
(Iteration 2121 / 9800) loss: 1.251047
(Iteration 2131 / 9800) loss: 1.483206
(Iteration 2141 / 9800) loss: 1.408018
(Iteration 2151 / 9800) loss: 1.391494
(Iteration 2161 / 9800) loss: 1.125423
(Iteration 2171 / 9800) loss: 1.265213
(Iteration 2181 / 9800) loss: 1.525398
(Iteration 2191 / 9800) loss: 1.558294
(Iteration 2201 / 9800) loss: 1.289876
(Iteration 2211 / 9800) loss: 1.262568
(Iteration 2221 / 9800) loss: 1.425381
(Iteration 2231 / 9800) loss: 1.337252
(Iteration 2241 / 9800) loss: 1.545367
(Iteration 2251 / 9800) loss: 1.374045
(Iteration 2261 / 9800) loss: 1.281114
(Iteration 2271 / 9800) loss: 1.156690
(Iteration 2281 / 9800) loss: 1.252744
(Iteration 2291 / 9800) loss: 1.491081
(Iteration 2301 / 9800) loss: 1.487404
```

```
(Iteration 2311 / 9800) loss: 1.328186
(Iteration 2321 / 9800) loss: 1.302702
(Iteration 2331 / 9800) loss: 1.327841
(Iteration 2341 / 9800) loss: 1.233847
(Iteration 2351 / 9800) loss: 1.141752
(Iteration 2361 / 9800) loss: 1.443309
(Iteration 2371 / 9800) loss: 1.452064
(Iteration 2381 / 9800) loss: 1.512503
(Iteration 2391 / 9800) loss: 1.476138
(Iteration 2401 / 9800) loss: 1.236173
(Iteration 2411 / 9800) loss: 1.276066
(Iteration 2421 / 9800) loss: 1.233626
(Iteration 2431 / 9800) loss: 1.293082
(Iteration 2441 / 9800) loss: 1.195045
(Epoch 5 / 20) train acc: 0.510000; val_acc: 0.507000
(Iteration 2451 / 9800) loss: 1.489424
(Iteration 2461 / 9800) loss: 1.246508
(Iteration 2471 / 9800) loss: 1.301075
(Iteration 2481 / 9800) loss: 1.293582
(Iteration 2491 / 9800) loss: 1.224363
(Iteration 2501 / 9800) loss: 1.187834
(Iteration 2511 / 9800) loss: 1.191505
(Iteration 2521 / 9800) loss: 1.242158
(Iteration 2531 / 9800) loss: 1.320474
(Iteration 2541 / 9800) loss: 1.347812
(Iteration 2551 / 9800) loss: 1.160996
(Iteration 2561 / 9800) loss: 1.456521
(Iteration 2571 / 9800) loss: 1.152271
(Iteration 2581 / 9800) loss: 1.332251
(Iteration 2591 / 9800) loss: 1.554487
(Iteration 2601 / 9800) loss: 1.273241
(Iteration 2611 / 9800) loss: 1.457928
(Iteration 2621 / 9800) loss: 1.287100
(Iteration 2631 / 9800) loss: 1.234720
(Iteration 2641 / 9800) loss: 1.245242
(Iteration 2651 / 9800) loss: 1.384680
(Iteration 2661 / 9800) loss: 1.087025
(Iteration 2671 / 9800) loss: 1.132394
(Iteration 2681 / 9800) loss: 1.291262
(Iteration 2691 / 9800) loss: 1.254378
(Iteration 2701 / 9800) loss: 1.250450
(Iteration 2711 / 9800) loss: 1.090676
(Iteration 2721 / 9800) loss: 1.319811
(Iteration 2731 / 9800) loss: 1.123473
(Iteration 2741 / 9800) loss: 1.282171
(Iteration 2751 / 9800) loss: 1.164757
(Iteration 2761 / 9800) loss: 1.545158
(Iteration 2771 / 9800) loss: 1.156333
```

```
(Iteration 2781 / 9800) loss: 1.294921
(Iteration 2791 / 9800) loss: 1.280071
(Iteration 2801 / 9800) loss: 0.973284
(Iteration 2811 / 9800) loss: 1.298186
(Iteration 2821 / 9800) loss: 1.381347
(Iteration 2831 / 9800) loss: 1.274295
(Iteration 2841 / 9800) loss: 1.228968
(Iteration 2851 / 9800) loss: 1.298344
(Iteration 2861 / 9800) loss: 1.283421
(Iteration 2871 / 9800) loss: 1.239379
(Iteration 2881 / 9800) loss: 1.218998
(Iteration 2891 / 9800) loss: 1.314072
(Iteration 2901 / 9800) loss: 1.100167
(Iteration 2911 / 9800) loss: 1.110141
(Iteration 2921 / 9800) loss: 0.998821
(Iteration 2931 / 9800) loss: 1.364686
(Epoch 6 / 20) train acc: 0.566000; val_acc: 0.533000
(Iteration 2941 / 9800) loss: 1.337112
(Iteration 2951 / 9800) loss: 1.118084
(Iteration 2961 / 9800) loss: 1.262627
(Iteration 2971 / 9800) loss: 1.170338
(Iteration 2981 / 9800) loss: 1.147738
(Iteration 2991 / 9800) loss: 1.288369
(Iteration 3001 / 9800) loss: 1.257685
(Iteration 3011 / 9800) loss: 1.168786
(Iteration 3021 / 9800) loss: 1.219139
(Iteration 3031 / 9800) loss: 1.197054
(Iteration 3041 / 9800) loss: 1.209927
(Iteration 3051 / 9800) loss: 1.201144
(Iteration 3061 / 9800) loss: 1.292817
(Iteration 3071 / 9800) loss: 1.243021
(Iteration 3081 / 9800) loss: 1.122446
(Iteration 3091 / 9800) loss: 1.265249
(Iteration 3101 / 9800) loss: 1.128589
(Iteration 3111 / 9800) loss: 1.459281
(Iteration 3121 / 9800) loss: 1.119847
(Iteration 3131 / 9800) loss: 1.312438
(Iteration 3141 / 9800) loss: 1.267777
(Iteration 3151 / 9800) loss: 1.230261
(Iteration 3161 / 9800) loss: 1.328239
(Iteration 3171 / 9800) loss: 1.331351
(Iteration 3181 / 9800) loss: 1.397325
(Iteration 3191 / 9800) loss: 1.246014
(Iteration 3201 / 9800) loss: 1.277505
(Iteration 3211 / 9800) loss: 1.221671
(Iteration 3221 / 9800) loss: 1.159225
(Iteration 3231 / 9800) loss: 1.311206
(Iteration 3241 / 9800) loss: 1.309711
```

```
(Iteration 3251 / 9800) loss: 1.280238
(Iteration 3261 / 9800) loss: 1.296598
(Iteration 3271 / 9800) loss: 1.263716
(Iteration 3281 / 9800) loss: 1.209364
(Iteration 3291 / 9800) loss: 1.184050
(Iteration 3301 / 9800) loss: 1.132421
(Iteration 3311 / 9800) loss: 1.295286
(Iteration 3321 / 9800) loss: 1.359394
(Iteration 3331 / 9800) loss: 1.277517
(Iteration 3341 / 9800) loss: 1.210039
(Iteration 3351 / 9800) loss: 1.355757
(Iteration 3361 / 9800) loss: 1.372822
(Iteration 3371 / 9800) loss: 1.396012
(Iteration 3381 / 9800) loss: 1.385399
(Iteration 3391 / 9800) loss: 1.324363
(Iteration 3401 / 9800) loss: 1.155722
(Iteration 3411 / 9800) loss: 1.332826
(Iteration 3421 / 9800) loss: 1.112181
(Epoch 7 / 20) train acc: 0.552000; val_acc: 0.511000
(Iteration 3431 / 9800) loss: 1.118578
(Iteration 3441 / 9800) loss: 1.331461
(Iteration 3451 / 9800) loss: 1.046354
(Iteration 3461 / 9800) loss: 1.095912
(Iteration 3471 / 9800) loss: 1.019803
(Iteration 3481 / 9800) loss: 1.398973
(Iteration 3491 / 9800) loss: 1.247510
(Iteration 3501 / 9800) loss: 1.218081
(Iteration 3511 / 9800) loss: 1.328279
(Iteration 3521 / 9800) loss: 1.196227
(Iteration 3531 / 9800) loss: 1.106223
(Iteration 3541 / 9800) loss: 1.144245
(Iteration 3551 / 9800) loss: 1.321079
(Iteration 3561 / 9800) loss: 1.079354
(Iteration 3571 / 9800) loss: 1.122888
(Iteration 3581 / 9800) loss: 1.260656
(Iteration 3591 / 9800) loss: 1.146812
(Iteration 3601 / 9800) loss: 1.240543
(Iteration 3611 / 9800) loss: 1.215692
(Iteration 3621 / 9800) loss: 0.918462
(Iteration 3631 / 9800) loss: 1.180711
(Iteration 3641 / 9800) loss: 1.382483
(Iteration 3651 / 9800) loss: 0.936387
(Iteration 3661 / 9800) loss: 1.094818
(Iteration 3671 / 9800) loss: 1.178838
(Iteration 3681 / 9800) loss: 1.328679
(Iteration 3691 / 9800) loss: 1.122398
(Iteration 3701 / 9800) loss: 1.112707
(Iteration 3711 / 9800) loss: 1.401948
```

```
(Iteration 3721 / 9800) loss: 1.186876
(Iteration 3731 / 9800) loss: 1.270350
(Iteration 3741 / 9800) loss: 1.138164
(Iteration 3751 / 9800) loss: 1.113749
(Iteration 3761 / 9800) loss: 1.231010
(Iteration 3771 / 9800) loss: 1.227295
(Iteration 3781 / 9800) loss: 1.243993
(Iteration 3791 / 9800) loss: 1.021825
(Iteration 3801 / 9800) loss: 1.235441
(Iteration 3811 / 9800) loss: 1.171919
(Iteration 3821 / 9800) loss: 1.191023
(Iteration 3831 / 9800) loss: 1.208112
(Iteration 3841 / 9800) loss: 1.138801
(Iteration 3851 / 9800) loss: 1.226439
(Iteration 3861 / 9800) loss: 0.945473
(Iteration 3871 / 9800) loss: 1.080480
(Iteration 3881 / 9800) loss: 1.017295
(Iteration 3891 / 9800) loss: 0.910930
(Iteration 3901 / 9800) loss: 1.280407
(Iteration 3911 / 9800) loss: 1.161253
(Epoch 8 / 20) train acc: 0.573000; val_acc: 0.518000
(Iteration 3921 / 9800) loss: 1.153326
(Iteration 3931 / 9800) loss: 1.121190
(Iteration 3941 / 9800) loss: 1.038038
(Iteration 3951 / 9800) loss: 1.144616
(Iteration 3961 / 9800) loss: 1.023494
(Iteration 3971 / 9800) loss: 1.086527
(Iteration 3981 / 9800) loss: 1.125083
(Iteration 3991 / 9800) loss: 0.970808
(Iteration 4001 / 9800) loss: 1.081762
(Iteration 4011 / 9800) loss: 1.179879
(Iteration 4021 / 9800) loss: 1.350290
(Iteration 4031 / 9800) loss: 1.182838
(Iteration 4041 / 9800) loss: 1.314243
(Iteration 4051 / 9800) loss: 1.108137
(Iteration 4061 / 9800) loss: 1.126737
(Iteration 4071 / 9800) loss: 1.173319
(Iteration 4081 / 9800) loss: 1.122147
(Iteration 4091 / 9800) loss: 1.064607
(Iteration 4101 / 9800) loss: 1.183027
(Iteration 4111 / 9800) loss: 1.066429
(Iteration 4121 / 9800) loss: 1.001158
(Iteration 4131 / 9800) loss: 1.037044
(Iteration 4141 / 9800) loss: 1.233360
(Iteration 4151 / 9800) loss: 1.117031
(Iteration 4161 / 9800) loss: 1.046628
(Iteration 4171 / 9800) loss: 0.862356
(Iteration 4181 / 9800) loss: 1.218032
```

```
(Iteration 4191 / 9800) loss: 1.226179
(Iteration 4201 / 9800) loss: 1.042122
(Iteration 4211 / 9800) loss: 0.981708
(Iteration 4221 / 9800) loss: 0.925946
(Iteration 4231 / 9800) loss: 0.973963
(Iteration 4241 / 9800) loss: 1.168240
(Iteration 4251 / 9800) loss: 1.118591
(Iteration 4261 / 9800) loss: 1.185578
(Iteration 4271 / 9800) loss: 1.276777
(Iteration 4281 / 9800) loss: 1.073931
(Iteration 4291 / 9800) loss: 1.172062
(Iteration 4301 / 9800) loss: 0.945988
(Iteration 4311 / 9800) loss: 1.033134
(Iteration 4321 / 9800) loss: 1.283601
(Iteration 4331 / 9800) loss: 0.949655
(Iteration 4341 / 9800) loss: 1.027474
(Iteration 4351 / 9800) loss: 1.015917
(Iteration 4361 / 9800) loss: 1.049250
(Iteration 4371 / 9800) loss: 1.044707
(Iteration 4381 / 9800) loss: 1.380167
(Iteration 4391 / 9800) loss: 1.186792
(Iteration 4401 / 9800) loss: 1.255721
(Epoch 9 / 20) train acc: 0.611000; val_acc: 0.534000
(Iteration 4411 / 9800) loss: 1.127020
(Iteration 4421 / 9800) loss: 1.080289
(Iteration 4431 / 9800) loss: 1.077148
(Iteration 4441 / 9800) loss: 1.041952
(Iteration 4451 / 9800) loss: 1.224917
(Iteration 4461 / 9800) loss: 1.268964
(Iteration 4471 / 9800) loss: 1.014393
(Iteration 4481 / 9800) loss: 1.159475
(Iteration 4491 / 9800) loss: 1.084852
(Iteration 4501 / 9800) loss: 1.079988
(Iteration 4511 / 9800) loss: 1.269733
(Iteration 4521 / 9800) loss: 0.983857
(Iteration 4531 / 9800) loss: 1.332211
(Iteration 4541 / 9800) loss: 0.949416
(Iteration 4551 / 9800) loss: 0.946387
(Iteration 4561 / 9800) loss: 1.223227
(Iteration 4571 / 9800) loss: 1.063018
(Iteration 4581 / 9800) loss: 1.327975
(Iteration 4591 / 9800) loss: 0.932150
(Iteration 4601 / 9800) loss: 1.159035
(Iteration 4611 / 9800) loss: 1.079382
(Iteration 4621 / 9800) loss: 1.341186
(Iteration 4631 / 9800) loss: 1.138658
(Iteration 4641 / 9800) loss: 0.892237
(Iteration 4651 / 9800) loss: 1.048923
```

```
(Iteration 4661 / 9800) loss: 0.874531
(Iteration 4671 / 9800) loss: 1.011631
(Iteration 4681 / 9800) loss: 1.150078
(Iteration 4691 / 9800) loss: 1.254593
(Iteration 4701 / 9800) loss: 1.062380
(Iteration 4711 / 9800) loss: 0.947590
(Iteration 4721 / 9800) loss: 0.987454
(Iteration 4731 / 9800) loss: 0.866712
(Iteration 4741 / 9800) loss: 1.124800
(Iteration 4751 / 9800) loss: 1.140803
(Iteration 4761 / 9800) loss: 1.050772
(Iteration 4771 / 9800) loss: 0.990043
(Iteration 4781 / 9800) loss: 1.201434
(Iteration 4791 / 9800) loss: 1.227974
(Iteration 4801 / 9800) loss: 1.020558
(Iteration 4811 / 9800) loss: 1.062469
(Iteration 4821 / 9800) loss: 1.019361
(Iteration 4831 / 9800) loss: 1.094151
(Iteration 4841 / 9800) loss: 1.370797
(Iteration 4851 / 9800) loss: 1.101888
(Iteration 4861 / 9800) loss: 1.133698
(Iteration 4871 / 9800) loss: 0.981018
(Iteration 4881 / 9800) loss: 1.019603
(Iteration 4891 / 9800) loss: 1.284391
(Epoch 10 / 20) train acc: 0.626000; val_acc: 0.517000
(Iteration 4901 / 9800) loss: 1.188625
(Iteration 4911 / 9800) loss: 1.091592
(Iteration 4921 / 9800) loss: 1.112942
(Iteration 4931 / 9800) loss: 1.096250
(Iteration 4941 / 9800) loss: 1.096898
(Iteration 4951 / 9800) loss: 1.082035
(Iteration 4961 / 9800) loss: 0.964660
(Iteration 4971 / 9800) loss: 1.058817
(Iteration 4981 / 9800) loss: 1.110739
(Iteration 4991 / 9800) loss: 0.911412
(Iteration 5001 / 9800) loss: 1.095555
(Iteration 5011 / 9800) loss: 1.169128
(Iteration 5021 / 9800) loss: 1.101317
(Iteration 5031 / 9800) loss: 1.058420
(Iteration 5041 / 9800) loss: 1.055261
(Iteration 5051 / 9800) loss: 1.309088
(Iteration 5061 / 9800) loss: 1.190792
(Iteration 5071 / 9800) loss: 0.708592
(Iteration 5081 / 9800) loss: 1.071181
(Iteration 5091 / 9800) loss: 1.128087
(Iteration 5101 / 9800) loss: 1.339637
(Iteration 5111 / 9800) loss: 1.046909
(Iteration 5121 / 9800) loss: 1.083694
```

```
(Iteration 5131 / 9800) loss: 1.107082
(Iteration 5141 / 9800) loss: 1.066948
(Iteration 5151 / 9800) loss: 1.069394
(Iteration 5161 / 9800) loss: 1.135431
(Iteration 5171 / 9800) loss: 1.193852
(Iteration 5181 / 9800) loss: 1.092334
(Iteration 5191 / 9800) loss: 1.038933
(Iteration 5201 / 9800) loss: 1.113057
(Iteration 5211 / 9800) loss: 1.183416
(Iteration 5221 / 9800) loss: 1.084712
(Iteration 5231 / 9800) loss: 0.831129
(Iteration 5241 / 9800) loss: 1.083115
(Iteration 5251 / 9800) loss: 1.267537
(Iteration 5261 / 9800) loss: 1.027642
(Iteration 5271 / 9800) loss: 0.924352
(Iteration 5281 / 9800) loss: 1.108893
(Iteration 5291 / 9800) loss: 1.087536
(Iteration 5301 / 9800) loss: 1.215943
(Iteration 5311 / 9800) loss: 0.889484
(Iteration 5321 / 9800) loss: 1.060887
(Iteration 5331 / 9800) loss: 0.967133
(Iteration 5341 / 9800) loss: 1.023608
(Iteration 5351 / 9800) loss: 1.087028
(Iteration 5361 / 9800) loss: 0.997253
(Iteration 5371 / 9800) loss: 1.049170
(Iteration 5381 / 9800) loss: 1.204213
(Epoch 11 / 20) train acc: 0.648000; val_acc: 0.520000
(Iteration 5391 / 9800) loss: 1.135659
(Iteration 5401 / 9800) loss: 0.898257
(Iteration 5411 / 9800) loss: 0.996621
(Iteration 5421 / 9800) loss: 0.935083
(Iteration 5431 / 9800) loss: 1.068976
(Iteration 5441 / 9800) loss: 0.901754
(Iteration 5451 / 9800) loss: 1.005247
(Iteration 5461 / 9800) loss: 1.197226
(Iteration 5471 / 9800) loss: 1.077413
(Iteration 5481 / 9800) loss: 0.907656
(Iteration 5491 / 9800) loss: 1.147336
(Iteration 5501 / 9800) loss: 1.034409
(Iteration 5511 / 9800) loss: 1.039639
(Iteration 5521 / 9800) loss: 1.273795
(Iteration 5531 / 9800) loss: 1.077780
(Iteration 5541 / 9800) loss: 1.071949
(Iteration 5551 / 9800) loss: 1.174131
(Iteration 5561 / 9800) loss: 1.178692
(Iteration 5571 / 9800) loss: 1.170486
(Iteration 5581 / 9800) loss: 1.012258
(Iteration 5591 / 9800) loss: 1.112422
```

```
(Iteration 5601 / 9800) loss: 1.250752
(Iteration 5611 / 9800) loss: 0.912797
(Iteration 5621 / 9800) loss: 0.930369
(Iteration 5631 / 9800) loss: 0.976217
(Iteration 5641 / 9800) loss: 0.982177
(Iteration 5651 / 9800) loss: 0.846776
(Iteration 5661 / 9800) loss: 0.873548
(Iteration 5671 / 9800) loss: 0.842876
(Iteration 5681 / 9800) loss: 1.107317
(Iteration 5691 / 9800) loss: 1.004805
(Iteration 5701 / 9800) loss: 1.039153
(Iteration 5711 / 9800) loss: 0.863422
(Iteration 5721 / 9800) loss: 0.891038
(Iteration 5731 / 9800) loss: 0.987502
(Iteration 5741 / 9800) loss: 1.130416
(Iteration 5751 / 9800) loss: 1.039212
(Iteration 5761 / 9800) loss: 1.154184
(Iteration 5771 / 9800) loss: 0.999728
(Iteration 5781 / 9800) loss: 1.065273
(Iteration 5791 / 9800) loss: 0.865679
(Iteration 5801 / 9800) loss: 0.961054
(Iteration 5811 / 9800) loss: 0.977553
(Iteration 5821 / 9800) loss: 1.141958
(Iteration 5831 / 9800) loss: 0.997707
(Iteration 5841 / 9800) loss: 1.065747
(Iteration 5851 / 9800) loss: 0.912564
(Iteration 5861 / 9800) loss: 1.069294
(Iteration 5871 / 9800) loss: 0.952018
(Epoch 12 / 20) train acc: 0.636000; val_acc: 0.518000
(Iteration 5881 / 9800) loss: 1.013761
(Iteration 5891 / 9800) loss: 1.067495
(Iteration 5901 / 9800) loss: 0.984542
(Iteration 5911 / 9800) loss: 0.844915
(Iteration 5921 / 9800) loss: 0.971419
(Iteration 5931 / 9800) loss: 1.035798
(Iteration 5941 / 9800) loss: 1.181568
(Iteration 5951 / 9800) loss: 0.884959
(Iteration 5961 / 9800) loss: 1.165940
(Iteration 5971 / 9800) loss: 0.859811
(Iteration 5981 / 9800) loss: 0.987029
(Iteration 5991 / 9800) loss: 1.196318
(Iteration 6001 / 9800) loss: 1.030073
(Iteration 6011 / 9800) loss: 1.056895
(Iteration 6021 / 9800) loss: 0.821735
(Iteration 6031 / 9800) loss: 0.960598
(Iteration 6041 / 9800) loss: 1.016693
(Iteration 6051 / 9800) loss: 0.874344
(Iteration 6061 / 9800) loss: 1.030008
```

```
(Iteration 6071 / 9800) loss: 0.954666
(Iteration 6081 / 9800) loss: 1.065682
(Iteration 6091 / 9800) loss: 0.956356
(Iteration 6101 / 9800) loss: 0.956301
(Iteration 6111 / 9800) loss: 0.911443
(Iteration 6121 / 9800) loss: 0.991858
(Iteration 6131 / 9800) loss: 1.125363
(Iteration 6141 / 9800) loss: 1.128127
(Iteration 6151 / 9800) loss: 0.967769
(Iteration 6161 / 9800) loss: 1.161247
(Iteration 6171 / 9800) loss: 0.960738
(Iteration 6181 / 9800) loss: 0.990226
(Iteration 6191 / 9800) loss: 1.051834
(Iteration 6201 / 9800) loss: 1.089068
(Iteration 6211 / 9800) loss: 1.100545
(Iteration 6221 / 9800) loss: 0.724106
(Iteration 6231 / 9800) loss: 1.126266
(Iteration 6241 / 9800) loss: 1.022084
(Iteration 6251 / 9800) loss: 0.942832
(Iteration 6261 / 9800) loss: 0.898455
(Iteration 6271 / 9800) loss: 1.095858
(Iteration 6281 / 9800) loss: 0.903117
(Iteration 6291 / 9800) loss: 0.897215
(Iteration 6301 / 9800) loss: 1.007187
(Iteration 6311 / 9800) loss: 0.908238
(Iteration 6321 / 9800) loss: 0.891556
(Iteration 6331 / 9800) loss: 1.073987
(Iteration 6341 / 9800) loss: 1.006055
(Iteration 6351 / 9800) loss: 0.815674
(Iteration 6361 / 9800) loss: 1.001645
(Epoch 13 / 20) train acc: 0.628000; val_acc: 0.518000
(Iteration 6371 / 9800) loss: 0.845412
(Iteration 6381 / 9800) loss: 0.795464
(Iteration 6391 / 9800) loss: 0.878389
(Iteration 6401 / 9800) loss: 0.985682
(Iteration 6411 / 9800) loss: 0.885332
(Iteration 6421 / 9800) loss: 0.916014
(Iteration 6431 / 9800) loss: 0.899444
(Iteration 6441 / 9800) loss: 1.183935
(Iteration 6451 / 9800) loss: 0.828684
(Iteration 6461 / 9800) loss: 0.930846
(Iteration 6471 / 9800) loss: 0.881735
(Iteration 6481 / 9800) loss: 1.047614
(Iteration 6491 / 9800) loss: 0.970009
(Iteration 6501 / 9800) loss: 1.036150
(Iteration 6511 / 9800) loss: 0.909600
(Iteration 6521 / 9800) loss: 0.841511
(Iteration 6531 / 9800) loss: 0.850330
```

```
(Iteration 6541 / 9800) loss: 1.042193
(Iteration 6551 / 9800) loss: 0.816288
(Iteration 6561 / 9800) loss: 1.069428
(Iteration 6571 / 9800) loss: 1.126455
(Iteration 6581 / 9800) loss: 1.026672
(Iteration 6591 / 9800) loss: 0.997831
(Iteration 6601 / 9800) loss: 0.804457
(Iteration 6611 / 9800) loss: 1.067271
(Iteration 6621 / 9800) loss: 1.070413
(Iteration 6631 / 9800) loss: 1.070877
(Iteration 6641 / 9800) loss: 0.919664
(Iteration 6651 / 9800) loss: 1.085661
(Iteration 6661 / 9800) loss: 0.864873
(Iteration 6671 / 9800) loss: 0.858197
(Iteration 6681 / 9800) loss: 0.946800
(Iteration 6691 / 9800) loss: 1.131519
(Iteration 6701 / 9800) loss: 0.880601
(Iteration 6711 / 9800) loss: 0.784056
(Iteration 6721 / 9800) loss: 0.862205
(Iteration 6731 / 9800) loss: 0.885755
(Iteration 6741 / 9800) loss: 1.048640
(Iteration 6751 / 9800) loss: 1.040962
(Iteration 6761 / 9800) loss: 1.222907
(Iteration 6771 / 9800) loss: 0.849375
(Iteration 6781 / 9800) loss: 0.950682
(Iteration 6791 / 9800) loss: 0.864420
(Iteration 6801 / 9800) loss: 0.901531
(Iteration 6811 / 9800) loss: 1.086635
(Iteration 6821 / 9800) loss: 0.981399
(Iteration 6831 / 9800) loss: 0.904654
(Iteration 6841 / 9800) loss: 0.951133
(Iteration 6851 / 9800) loss: 1.060417
(Epoch 14 / 20) train acc: 0.645000; val_acc: 0.525000
(Iteration 6861 / 9800) loss: 1.064772
(Iteration 6871 / 9800) loss: 1.065067
(Iteration 6881 / 9800) loss: 1.070217
(Iteration 6891 / 9800) loss: 0.952181
(Iteration 6901 / 9800) loss: 0.981764
(Iteration 6911 / 9800) loss: 1.029769
(Iteration 6921 / 9800) loss: 0.869385
(Iteration 6931 / 9800) loss: 1.159323
(Iteration 6941 / 9800) loss: 1.035403
(Iteration 6951 / 9800) loss: 0.917941
(Iteration 6961 / 9800) loss: 1.002229
(Iteration 6971 / 9800) loss: 0.856722
(Iteration 6981 / 9800) loss: 1.056186
(Iteration 6991 / 9800) loss: 1.008300
(Iteration 7001 / 9800) loss: 0.900705
```

```
(Iteration 7011 / 9800) loss: 1.010154
(Iteration 7021 / 9800) loss: 0.906093
(Iteration 7031 / 9800) loss: 0.883141
(Iteration 7041 / 9800) loss: 0.906875
(Iteration 7051 / 9800) loss: 1.024527
(Iteration 7061 / 9800) loss: 1.043609
(Iteration 7071 / 9800) loss: 0.877379
(Iteration 7081 / 9800) loss: 1.094159
(Iteration 7091 / 9800) loss: 0.895221
(Iteration 7101 / 9800) loss: 0.968972
(Iteration 7111 / 9800) loss: 1.063877
(Iteration 7121 / 9800) loss: 0.871775
(Iteration 7131 / 9800) loss: 0.831872
(Iteration 7141 / 9800) loss: 1.008043
(Iteration 7151 / 9800) loss: 1.037123
(Iteration 7161 / 9800) loss: 1.086924
(Iteration 7171 / 9800) loss: 1.101796
(Iteration 7181 / 9800) loss: 0.932224
(Iteration 7191 / 9800) loss: 0.787993
(Iteration 7201 / 9800) loss: 1.083438
(Iteration 7211 / 9800) loss: 0.840780
(Iteration 7221 / 9800) loss: 0.762775
(Iteration 7231 / 9800) loss: 0.864941
(Iteration 7241 / 9800) loss: 1.079083
(Iteration 7251 / 9800) loss: 0.953677
(Iteration 7261 / 9800) loss: 0.911758
(Iteration 7271 / 9800) loss: 0.912187
(Iteration 7281 / 9800) loss: 0.981930
(Iteration 7291 / 9800) loss: 1.107623
(Iteration 7301 / 9800) loss: 0.958162
(Iteration 7311 / 9800) loss: 0.868780
(Iteration 7321 / 9800) loss: 0.723474
(Iteration 7331 / 9800) loss: 0.961279
(Iteration 7341 / 9800) loss: 1.172271
(Epoch 15 / 20) train acc: 0.685000; val_acc: 0.517000
(Iteration 7351 / 9800) loss: 0.969623
(Iteration 7361 / 9800) loss: 0.810808
(Iteration 7371 / 9800) loss: 1.015609
(Iteration 7381 / 9800) loss: 0.987597
(Iteration 7391 / 9800) loss: 0.863103
(Iteration 7401 / 9800) loss: 0.885632
(Iteration 7411 / 9800) loss: 0.945629
(Iteration 7421 / 9800) loss: 0.936141
(Iteration 7431 / 9800) loss: 0.739192
(Iteration 7441 / 9800) loss: 0.940440
(Iteration 7451 / 9800) loss: 1.063978
(Iteration 7461 / 9800) loss: 0.998808
(Iteration 7471 / 9800) loss: 0.890908
```

```
(Iteration 7481 / 9800) loss: 0.944019
(Iteration 7491 / 9800) loss: 0.961148
(Iteration 7501 / 9800) loss: 0.886474
(Iteration 7511 / 9800) loss: 1.006493
(Iteration 7521 / 9800) loss: 1.175131
(Iteration 7531 / 9800) loss: 0.993612
(Iteration 7541 / 9800) loss: 0.837885
(Iteration 7551 / 9800) loss: 0.901936
(Iteration 7561 / 9800) loss: 1.098074
(Iteration 7571 / 9800) loss: 0.873281
(Iteration 7581 / 9800) loss: 1.005986
(Iteration 7591 / 9800) loss: 1.001273
(Iteration 7601 / 9800) loss: 0.835331
(Iteration 7611 / 9800) loss: 0.826985
(Iteration 7621 / 9800) loss: 0.911050
(Iteration 7631 / 9800) loss: 1.007158
(Iteration 7641 / 9800) loss: 0.886212
(Iteration 7651 / 9800) loss: 1.023516
(Iteration 7661 / 9800) loss: 0.864032
(Iteration 7671 / 9800) loss: 0.789589
(Iteration 7681 / 9800) loss: 0.975658
(Iteration 7691 / 9800) loss: 0.994820
(Iteration 7701 / 9800) loss: 0.956491
(Iteration 7711 / 9800) loss: 0.951916
(Iteration 7721 / 9800) loss: 0.846584
(Iteration 7731 / 9800) loss: 0.878182
(Iteration 7741 / 9800) loss: 0.974274
(Iteration 7751 / 9800) loss: 1.131864
(Iteration 7761 / 9800) loss: 0.995856
(Iteration 7771 / 9800) loss: 0.840681
(Iteration 7781 / 9800) loss: 0.885289
(Iteration 7791 / 9800) loss: 0.810774
(Iteration 7801 / 9800) loss: 1.114780
(Iteration 7811 / 9800) loss: 0.904211
(Iteration 7821 / 9800) loss: 0.866448
(Iteration 7831 / 9800) loss: 0.887118
(Epoch 16 / 20) train acc: 0.672000; val_acc: 0.526000
(Iteration 7841 / 9800) loss: 0.815967
(Iteration 7851 / 9800) loss: 1.019293
(Iteration 7861 / 9800) loss: 0.663726
(Iteration 7871 / 9800) loss: 0.860755
(Iteration 7881 / 9800) loss: 0.886405
(Iteration 7891 / 9800) loss: 0.928275
(Iteration 7901 / 9800) loss: 1.147353
(Iteration 7911 / 9800) loss: 0.968054
(Iteration 7921 / 9800) loss: 0.990619
(Iteration 7931 / 9800) loss: 0.725273
(Iteration 7941 / 9800) loss: 0.988839
```

```
(Iteration 7951 / 9800) loss: 1.049674
(Iteration 7961 / 9800) loss: 0.890969
(Iteration 7971 / 9800) loss: 0.987163
(Iteration 7981 / 9800) loss: 0.933962
(Iteration 7991 / 9800) loss: 1.007062
(Iteration 8001 / 9800) loss: 0.685327
(Iteration 8011 / 9800) loss: 0.737587
(Iteration 8021 / 9800) loss: 0.803753
(Iteration 8031 / 9800) loss: 0.923768
(Iteration 8041 / 9800) loss: 0.861503
(Iteration 8051 / 9800) loss: 0.993350
(Iteration 8061 / 9800) loss: 0.861173
(Iteration 8071 / 9800) loss: 0.935553
(Iteration 8081 / 9800) loss: 0.891060
(Iteration 8091 / 9800) loss: 0.990896
(Iteration 8101 / 9800) loss: 0.956255
(Iteration 8111 / 9800) loss: 0.839583
(Iteration 8121 / 9800) loss: 0.944421
(Iteration 8131 / 9800) loss: 0.850135
(Iteration 8141 / 9800) loss: 0.595094
(Iteration 8151 / 9800) loss: 0.753122
(Iteration 8161 / 9800) loss: 0.874382
(Iteration 8171 / 9800) loss: 0.801081
(Iteration 8181 / 9800) loss: 0.851749
(Iteration 8191 / 9800) loss: 0.710268
(Iteration 8201 / 9800) loss: 0.853636
(Iteration 8211 / 9800) loss: 0.996140
(Iteration 8221 / 9800) loss: 0.714340
(Iteration 8231 / 9800) loss: 0.754940
(Iteration 8241 / 9800) loss: 0.879468
(Iteration 8251 / 9800) loss: 1.136368
(Iteration 8261 / 9800) loss: 0.900523
(Iteration 8271 / 9800) loss: 0.955195
(Iteration 8281 / 9800) loss: 0.909802
(Iteration 8291 / 9800) loss: 0.740476
(Iteration 8301 / 9800) loss: 0.769636
(Iteration 8311 / 9800) loss: 0.925698
(Iteration 8321 / 9800) loss: 0.912752
(Epoch 17 / 20) train acc: 0.683000; val_acc: 0.516000
(Iteration 8331 / 9800) loss: 0.909885
(Iteration 8341 / 9800) loss: 0.811230
(Iteration 8351 / 9800) loss: 0.963282
(Iteration 8361 / 9800) loss: 0.954895
(Iteration 8371 / 9800) loss: 0.925818
(Iteration 8381 / 9800) loss: 0.861275
(Iteration 8391 / 9800) loss: 0.772062
(Iteration 8401 / 9800) loss: 0.942126
(Iteration 8411 / 9800) loss: 1.003148
```

```
(Iteration 8421 / 9800) loss: 0.875010
(Iteration 8431 / 9800) loss: 0.754219
(Iteration 8441 / 9800) loss: 0.803872
(Iteration 8451 / 9800) loss: 0.701993
(Iteration 8461 / 9800) loss: 0.882117
(Iteration 8471 / 9800) loss: 0.616695
(Iteration 8481 / 9800) loss: 0.856700
(Iteration 8491 / 9800) loss: 0.951814
(Iteration 8501 / 9800) loss: 0.793793
(Iteration 8511 / 9800) loss: 0.845435
(Iteration 8521 / 9800) loss: 0.784053
(Iteration 8531 / 9800) loss: 0.908357
(Iteration 8541 / 9800) loss: 1.041355
(Iteration 8551 / 9800) loss: 0.972951
(Iteration 8561 / 9800) loss: 0.703093
(Iteration 8571 / 9800) loss: 0.938347
(Iteration 8581 / 9800) loss: 0.906541
(Iteration 8591 / 9800) loss: 0.898057
(Iteration 8601 / 9800) loss: 0.709743
(Iteration 8611 / 9800) loss: 0.770581
(Iteration 8621 / 9800) loss: 0.855765
(Iteration 8631 / 9800) loss: 0.647435
(Iteration 8641 / 9800) loss: 1.032576
(Iteration 8651 / 9800) loss: 0.674735
(Iteration 8661 / 9800) loss: 1.022206
(Iteration 8671 / 9800) loss: 0.853940
(Iteration 8681 / 9800) loss: 0.903558
(Iteration 8691 / 9800) loss: 0.858337
(Iteration 8701 / 9800) loss: 0.777650
(Iteration 8711 / 9800) loss: 0.877702
(Iteration 8721 / 9800) loss: 0.855661
(Iteration 8731 / 9800) loss: 0.811539
(Iteration 8741 / 9800) loss: 0.849902
(Iteration 8751 / 9800) loss: 0.850658
(Iteration 8761 / 9800) loss: 0.801508
(Iteration 8771 / 9800) loss: 1.298341
(Iteration 8781 / 9800) loss: 0.849197
(Iteration 8791 / 9800) loss: 0.798197
(Iteration 8801 / 9800) loss: 0.862540
(Iteration 8811 / 9800) loss: 1.085672
(Epoch 18 / 20) train acc: 0.708000; val_acc: 0.530000
(Iteration 8821 / 9800) loss: 1.020558
(Iteration 8831 / 9800) loss: 0.886155
(Iteration 8841 / 9800) loss: 1.051110
(Iteration 8851 / 9800) loss: 0.750110
(Iteration 8861 / 9800) loss: 0.906350
(Iteration 8871 / 9800) loss: 0.904244
(Iteration 8881 / 9800) loss: 0.739601
```

```
(Iteration 8891 / 9800) loss: 0.874017
(Iteration 8901 / 9800) loss: 1.093933
(Iteration 8911 / 9800) loss: 0.906913
(Iteration 8921 / 9800) loss: 0.762364
(Iteration 8931 / 9800) loss: 1.024936
(Iteration 8941 / 9800) loss: 1.172865
(Iteration 8951 / 9800) loss: 0.770005
(Iteration 8961 / 9800) loss: 0.865477
(Iteration 8971 / 9800) loss: 0.859413
(Iteration 8981 / 9800) loss: 0.777268
(Iteration 8991 / 9800) loss: 0.633265
(Iteration 9001 / 9800) loss: 0.804309
(Iteration 9011 / 9800) loss: 0.721395
(Iteration 9021 / 9800) loss: 1.023170
(Iteration 9031 / 9800) loss: 1.024501
(Iteration 9041 / 9800) loss: 0.807202
(Iteration 9051 / 9800) loss: 1.041315
(Iteration 9061 / 9800) loss: 0.981575
(Iteration 9071 / 9800) loss: 1.140269
(Iteration 9081 / 9800) loss: 0.780102
(Iteration 9091 / 9800) loss: 0.799490
(Iteration 9101 / 9800) loss: 0.806266
(Iteration 9111 / 9800) loss: 0.914295
(Iteration 9121 / 9800) loss: 0.813747
(Iteration 9131 / 9800) loss: 0.874102
(Iteration 9141 / 9800) loss: 0.843852
(Iteration 9151 / 9800) loss: 0.996565
(Iteration 9161 / 9800) loss: 0.605466
(Iteration 9171 / 9800) loss: 0.921259
(Iteration 9181 / 9800) loss: 0.944141
(Iteration 9191 / 9800) loss: 0.639947
(Iteration 9201 / 9800) loss: 0.933537
(Iteration 9211 / 9800) loss: 0.862356
(Iteration 9221 / 9800) loss: 0.778324
(Iteration 9231 / 9800) loss: 0.813803
(Iteration 9241 / 9800) loss: 0.764433
(Iteration 9251 / 9800) loss: 0.746550
(Iteration 9261 / 9800) loss: 0.768448
(Iteration 9271 / 9800) loss: 0.756168
(Iteration 9281 / 9800) loss: 0.950110
(Iteration 9291 / 9800) loss: 0.946594
(Iteration 9301 / 9800) loss: 0.927208
(Epoch 19 / 20) train acc: 0.707000; val_acc: 0.529000
(Iteration 9311 / 9800) loss: 0.864230
(Iteration 9321 / 9800) loss: 0.982270
(Iteration 9331 / 9800) loss: 0.767899
(Iteration 9341 / 9800) loss: 0.912425
(Iteration 9351 / 9800) loss: 0.828114
```

```
(Iteration 9361 / 9800) loss: 0.919414
(Iteration 9371 / 9800) loss: 0.892448
(Iteration 9381 / 9800) loss: 0.810629
(Iteration 9391 / 9800) loss: 1.054790
(Iteration 9401 / 9800) loss: 0.788118
(Iteration 9411 / 9800) loss: 0.794686
(Iteration 9421 / 9800) loss: 0.811743
(Iteration 9431 / 9800) loss: 0.853904
(Iteration 9441 / 9800) loss: 0.679216
(Iteration 9451 / 9800) loss: 0.989763
(Iteration 9461 / 9800) loss: 0.849078
(Iteration 9471 / 9800) loss: 0.886863
(Iteration 9481 / 9800) loss: 0.686574
(Iteration 9491 / 9800) loss: 0.873632
(Iteration 9501 / 9800) loss: 0.834207
(Iteration 9511 / 9800) loss: 0.841562
(Iteration 9521 / 9800) loss: 0.778831
(Iteration 9531 / 9800) loss: 0.782715
(Iteration 9541 / 9800) loss: 0.715783
(Iteration 9551 / 9800) loss: 0.788982
(Iteration 9561 / 9800) loss: 0.879675
(Iteration 9571 / 9800) loss: 0.733631
(Iteration 9581 / 9800) loss: 0.927848
(Iteration 9591 / 9800) loss: 0.941123
(Iteration 9601 / 9800) loss: 0.844681
(Iteration 9611 / 9800) loss: 0.828752
(Iteration 9621 / 9800) loss: 0.762874
(Iteration 9631 / 9800) loss: 0.772157
(Iteration 9641 / 9800) loss: 0.820690
(Iteration 9651 / 9800) loss: 0.633941
(Iteration 9661 / 9800) loss: 1.080957
(Iteration 9671 / 9800) loss: 0.712495
(Iteration 9681 / 9800) loss: 0.742119
(Iteration 9691 / 9800) loss: 0.707796
(Iteration 9701 / 9800) loss: 0.875387
(Iteration 9711 / 9800) loss: 0.851491
(Iteration 9721 / 9800) loss: 0.994538
(Iteration 9731 / 9800) loss: 0.892846
(Iteration 9741 / 9800) loss: 0.954318
(Iteration 9751 / 9800) loss: 0.933578
(Iteration 9761 / 9800) loss: 0.845813
(Iteration 9771 / 9800) loss: 0.737536
(Iteration 9781 / 9800) loss: 0.712632
(Iteration 9791 / 9800) loss: 0.739725
(Epoch 20 / 20) train acc: 0.730000; val_acc: 0.521000
```

4 Test Your Model!

Run your best model on the validation and test sets. You should achieve at least 50% accuracy on the validation set and the test set.

[27]:
```
y_test_pred = np.argmax(best_model.loss(data['X_test']), axis=1)
y_val_pred = np.argmax(best_model.loss(data['X_val']), axis=1)
print('Validation set accuracy: ', (y_val_pred == data['y_val']).mean())
print('Test set accuracy: ', (y_test_pred == data['y_test']).mean())
```

```
Validation set accuracy:  0.534
Test set accuracy:  0.525
```