knn

February 3, 2026

[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 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.

[2]:
```
!pip -q install -U ipython
```

```
                        0.0/621.4 kB
? eta -:--:--
                      621.4/621.4 kB
36.3 MB/s eta 0:00:00
                        0.0/1.6
MB ? eta -:--:--
                      1.6/1.6 MB 79.2
MB/s eta 0:00:00
                        0.0/85.4
kB ? eta -:--:--
                      85.4/85.4 kB 8.6
MB/s eta 0:00:00
ERROR: pip's dependency resolver does not currently take into account
all the packages that are installed. This behaviour is the source of the
following dependency conflicts.
google-colab 1.0.0 requires ipython==7.34.0, but you have ipython 9.9.0 which is
incompatible.
```

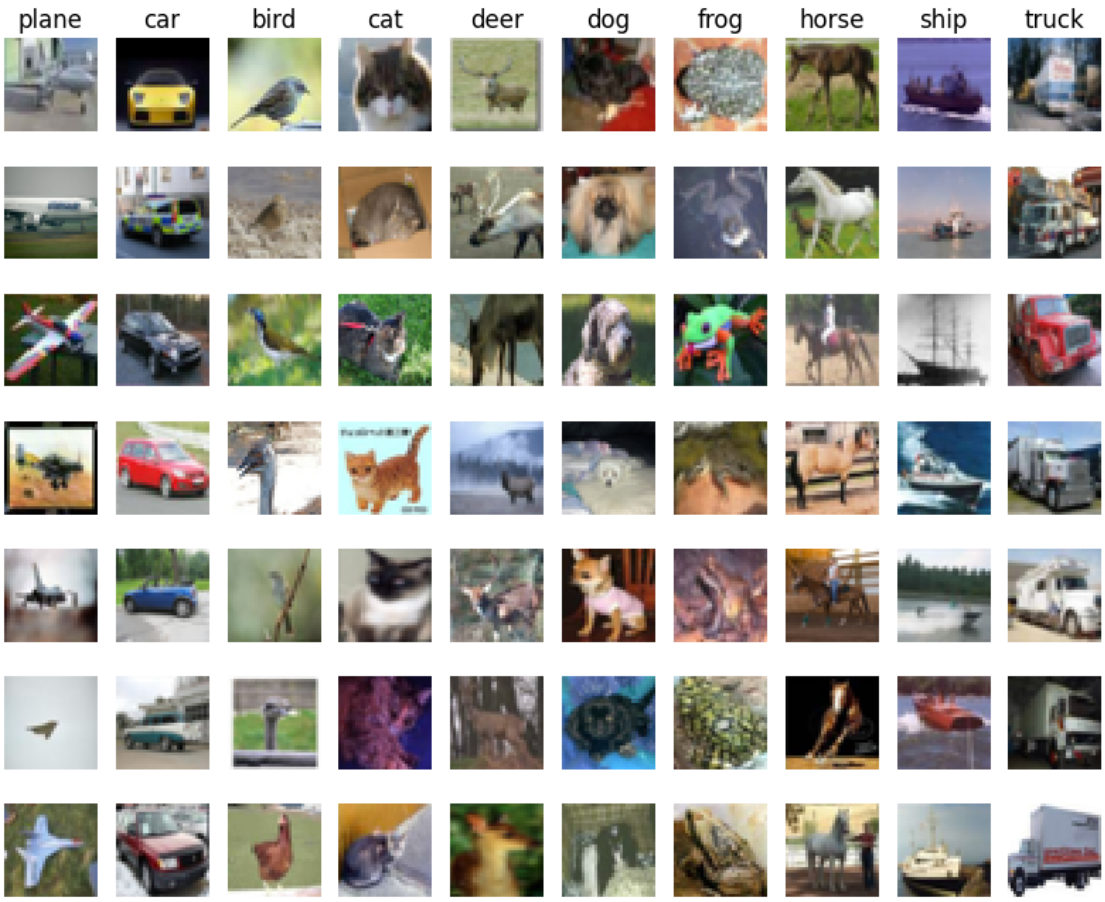
[3]:
```
# Run some setup code for this notebook.

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
```

[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]:
```
# 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)
```

[7]:
```
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.

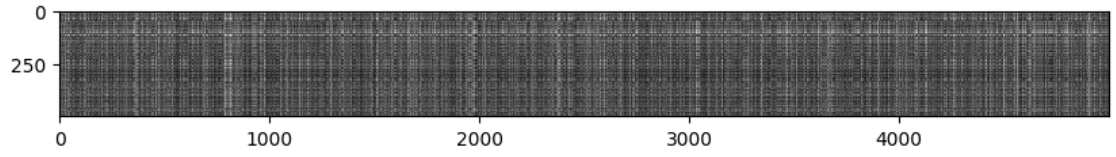
Note: For the three distance computations that we require you to implement in this notebook, you may not use the np.linalg.norm() function that numpy provides.

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.

[8]:
```
# Open cs231n/classifiers/k_nearest_neighbor.py and implement
# compute_distances_two_loops.

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

```
(500, 5000)
```

[9]:
```
# 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 : *If a single test image is different from many train images, you get light row. If a single train image is very different from all of the test images, you get light column*

[10]:
```
# 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`:

[11]:
```
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 :

YourExplanation :

[12]:
```
# Now lets speed up distance matrix computation by using partial vectorization
# with one loop. Implement the function compute_distances_one_loop and run the
# code below:
dists_one = classifier.compute_distances_one_loop(X_test)

# To ensure that our vectorized implementation is correct, we make sure that it
# agrees with the naive implementation. There are many ways to decide whether
# two matrices are similar; one of the simplest is the Frobenius norm. In case
# you haven't seen it before, the Frobenius norm of two matrices is the square
# root of the squared sum of differences of all elements; in other words, reshape
# the matrices into vectors and compute the Euclidean distance between them.
difference = np.linalg.norm(dists - dists_one, ord='fro')
print('One loop difference was: %f' % (difference, ))
if difference < 0.001:
    print('Good! The distance matrices are the same')
else:
    print('Uh-oh! The distance matrices are different')
```

```
One loop difference was: 0.000000
Good! The distance matrices are the same
```

[13]:
```
# Now implement the fully vectorized version inside compute_distances_no_loops
# and run the code
dists_two = classifier.compute_distances_no_loops(X_test)

# check that the distance matrix agrees with the one we computed before:
difference = np.linalg.norm(dists - dists_two, ord='fro')
print('No loop difference was: %f' % (difference, ))
if difference < 0.001:
    print('Good! The distance matrices are the same')
else:
    print('Uh-oh! The distance matrices are different')
```

```
No loop difference was: 0.000000
Good! The distance matrices are the same
```

[14]:
```
# Let's compare how fast the implementations are
def time_function(f, *args):
    """
    Call a function f with args and return the time (in seconds) that it took␣
 ↪to execute.
    """
    import time
    tic = time.time()
    f(*args)
    toc = time.time()
    return toc - tic

two_loop_time = time_function(classifier.compute_distances_two_loops, X_test)
print('Two loop version took %f seconds' % two_loop_time)

one_loop_time = time_function(classifier.compute_distances_one_loop, X_test)
print('One loop version took %f seconds' % one_loop_time)

no_loop_time = time_function(classifier.compute_distances_no_loops, X_test)
print('No loop version took %f seconds' % no_loop_time)

# You should see significantly faster performance with the fully vectorized␣
 ↪implementation!

# NOTE: depending on what machine you're using,
# you might not see a speedup when you go from two loops to one loop,
# and might even see a slow-down.
```

```
Two loop version took 253.383609 seconds
One loop version took 250.155631 seconds
No loop version took 0.626031 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.

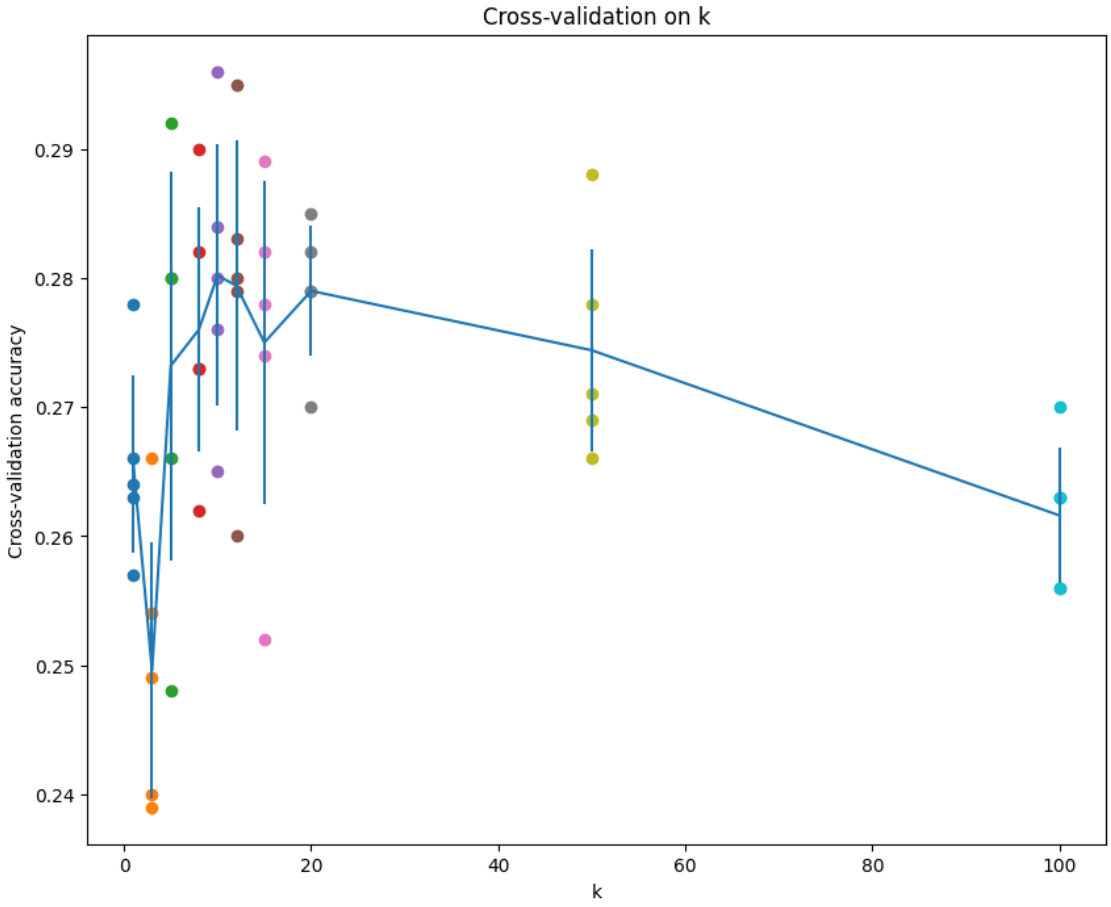
[15]:
```
num_folds = 5
k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50, 100]

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)



# 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_validation = X_train_folds[i]
      y_validation = y_train_folds[i]

      X_train2 = np.concatenate(X_train_folds[:i] + X_train_folds[i+1:])
      y_train2 = np.concatenate(y_train_folds[:i] + y_train_folds[i+1:])

      classifier = KNearestNeighbor()
      classifier.train(X_train2, y_train2)

      y_prediction = classifier.predict(X_validation, k=k)
      accuracy = (np.sum(y_prediction == y_validation)) / len(y_prediction)
      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
```

[16]:
```
# 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()
```



[17]:
```
# 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 = 7

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

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 2, 4

YourExplanation :

The training error of a 1-NN will always be lower because it will always find the correct original system, which results in the error being 0. As for the time to classify, the time will increase because each test image needs to be checked against each of the train set images. Thus it is O(n) time where n depends on size of train set.

[17]:

softmax

February 3, 2026

[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

[2]:
```
!pip -q install -U ipython
```

```
                        621.4/621.4 kB
11.7 MB/s eta 0:00:00
                        1.6/1.6 MB
29.1 MB/s eta 0:00:00
                        85.4/85.4 kB
3.4 MB/s eta 0:00:00
ERROR: pip's dependency resolver does not currently take into account
all the packages that are installed. This behaviour is the source of the
following dependency conflicts.
google-colab 1.0.0 requires ipython==7.34.0, but you have ipython 9.9.0 which is
incompatible.
```

[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
```

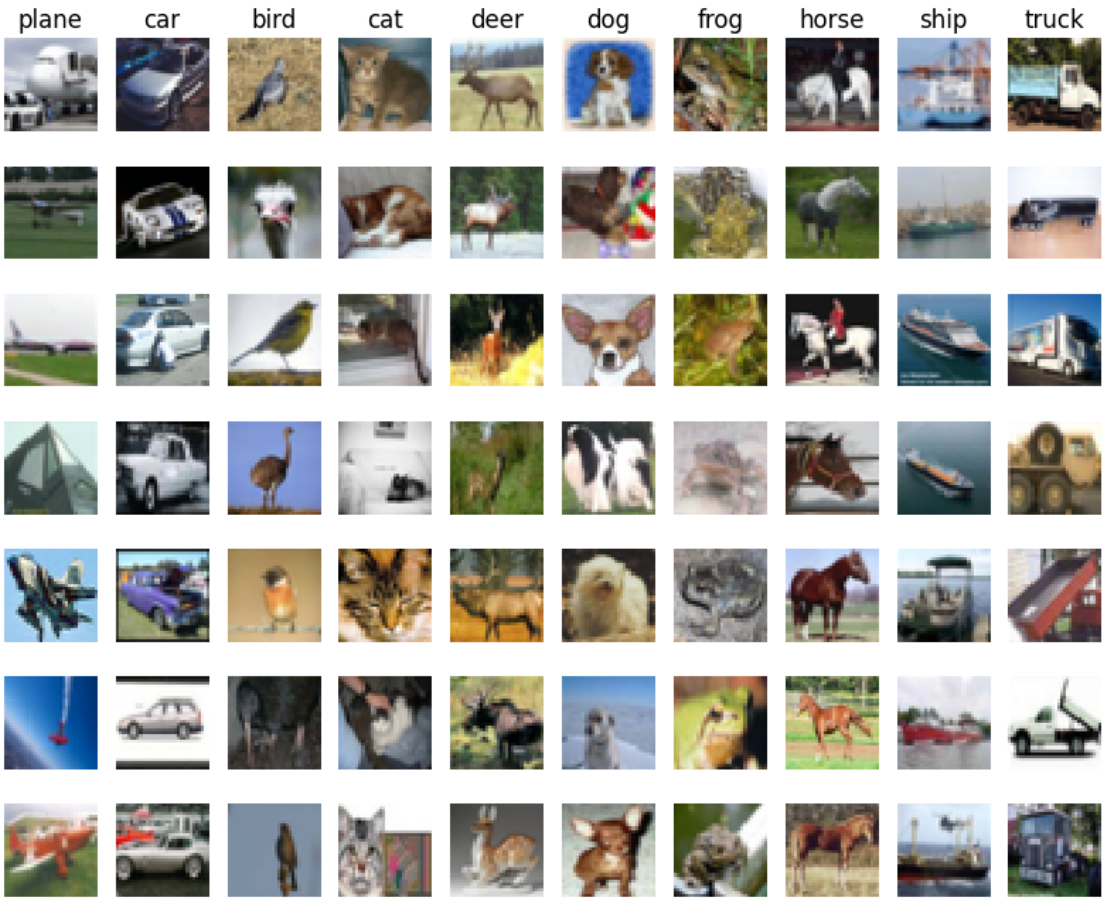
1.1 CIFAR-10 Data Loading and Preprocessing

[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)
print('Development data shape: ', X_dev.shape)
print('Development labels shape: ', y_dev.shape)
```

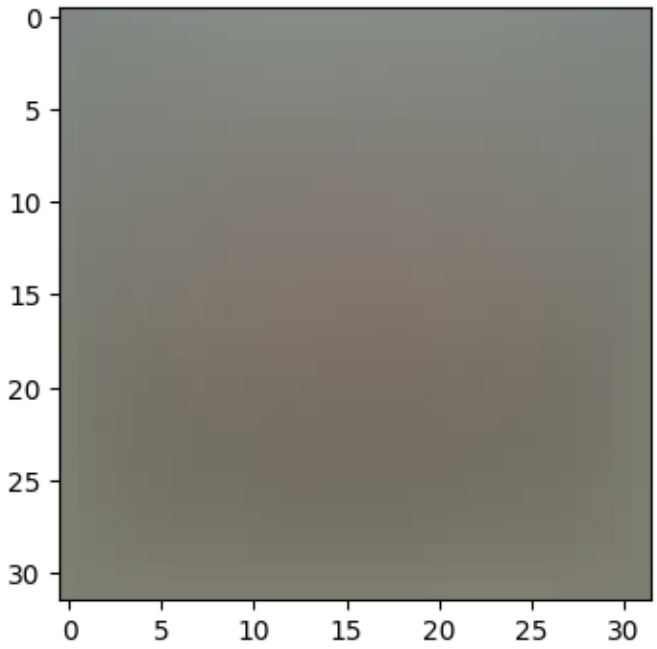
```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
Development data shape:  (500, 32, 32, 3)
Development labels shape:  (500,)
```

[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: 2.438894
loss: 2.438894
sanity check: 2.302585
```

Inline Question 1

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

Your Answer : CIFAR-10 has 10 classes. As such, given a fully random weights matrix, it will have approximately equal probability that an image is each of the 10 classes (so 10% each). Thus, for the correct class, it will have an expected probability of 10%, or 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: -0.607719 analytic: -0.607719, relative error: 7.544297e-08
numerical: -0.963731 analytic: -0.963731, relative error: 1.211442e-09
numerical: 2.357429 analytic: 2.357429, relative error: 2.982946e-08
numerical: 5.452634 analytic: 5.452634, relative error: 2.104067e-08
numerical: 1.709411 analytic: 1.709411, relative error: 1.285308e-08
numerical: 1.528882 analytic: 1.528882, relative error: 1.215897e-08
numerical: 0.208949 analytic: 0.208949, relative error: 1.955965e-07
numerical: -7.113117 analytic: -7.113117, relative error: 1.363815e-09
numerical: -2.429928 analytic: -2.429928, relative error: 3.209432e-08
numerical: 1.071678 analytic: 1.071678, relative error: 9.173443e-08
numerical: 1.670374 analytic: 1.670374, relative error: 2.154278e-08
numerical: -1.457070 analytic: -1.457070, relative error: 1.019196e-08
numerical: -0.191723 analytic: -0.191723, relative error: 2.355984e-07
numerical: -0.956268 analytic: -0.956269, relative error: 1.395899e-08
numerical: -0.160799 analytic: -0.160800, relative error: 2.818824e-07
numerical: 0.372659 analytic: 0.372659, relative error: 2.882204e-07
numerical: 1.859332 analytic: 1.859332, relative error: 4.385225e-08
numerical: 1.702587 analytic: 1.702586, relative error: 1.581221e-08
numerical: -2.747199 analytic: -2.747199, relative error: 2.202032e-09
numerical: -3.125688 analytic: -3.125688, relative error: 8.983242e-09
```

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 : fill this in.

[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: 2.438894e+00 computed in 0.073885s
Vectorized loss: 2.438894e+00 computed in 0.013832s
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 0.076666s
Vectorized loss and gradient: computed in 0.010486s
difference: 0.000000
```

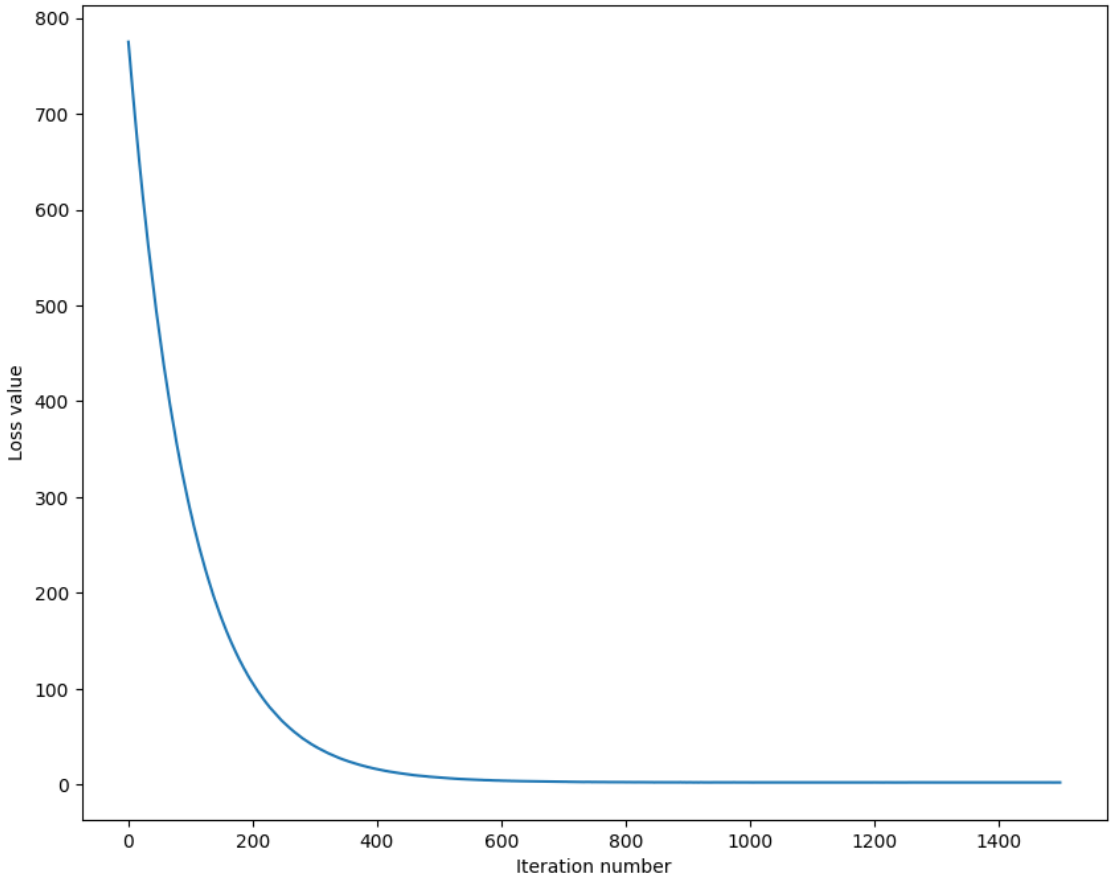
1.2.1 Stochastic Gradient Descent

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`.

[15]:
```
# 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 775.141391
iteration 100 / 1500: loss 284.904406
iteration 200 / 1500: loss 105.402891
iteration 300 / 1500: loss 39.874386
iteration 400 / 1500: loss 15.938130
iteration 500 / 1500: loss 7.144530
iteration 600 / 1500: loss 3.925246
iteration 700 / 1500: loss 2.791096
iteration 800 / 1500: loss 2.332104
iteration 900 / 1500: loss 2.180487
iteration 1000 / 1500: loss 2.160736
iteration 1100 / 1500: loss 2.113146
iteration 1200 / 1500: loss 2.104491
iteration 1300 / 1500: loss 2.091774
iteration 1400 / 1500: loss 2.077699
That took 11.990355s
```

[16]:
```
# 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()
```



[17]:
```
# 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: 0.330673
validation accuracy: 0.346000
```

[18]:
```
# Save the trained model for autograder.
softmax.save("softmax.npy")
```

```
softmax.npy saved.
```

[22]:
```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
```

```
# get a classification accuracy of about 0.365 (> 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, 1e-5, 1e-8, 1e-9]
regularization_strengths = [2.5e4, 1e4, 1e5, 5e4, 2e4]
best_val = 0
best_softmax = None

for lr in learning_rates:
    for reg in regularization_strengths:
      results[(lr, reg)] = []
      softmax = Softmax()
      loss_hist = softmax.train(X_train, y_train, learning_rate=lr, reg=reg,␣
 ↪num_iters=2000, verbose=True)
      y_train_pred = softmax.predict(X_train)
      y_val_pred = softmax.predict(X_val)
      results[(lr, reg)] = (np.mean(y_train == y_train_pred), np.mean(y_val ==␣
 ↪y_val_pred)) #cuz its number of samples correct, so just mean total
```

```
      if results[(lr, reg)][1] > best_val:
        best_val = results[(lr, reg)][1]
        best_softmax = softmax




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

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

```
iteration 0 / 2000: loss 773.860058
iteration 100 / 2000: loss 284.089373
iteration 200 / 2000: loss 105.245530
iteration 300 / 2000: loss 39.784664
iteration 400 / 2000: loss 15.812378
iteration 500 / 2000: loss 7.185407
iteration 600 / 2000: loss 3.959175
iteration 700 / 2000: loss 2.773858
iteration 800 / 2000: loss 2.344574
iteration 900 / 2000: loss 2.160718
iteration 1000 / 2000: loss 2.075880
iteration 1100 / 2000: loss 2.154104
iteration 1200 / 2000: loss 2.099103
iteration 1300 / 2000: loss 2.007851
iteration 1400 / 2000: loss 2.077097
iteration 1500 / 2000: loss 2.051218
iteration 1600 / 2000: loss 2.096206
iteration 1700 / 2000: loss 2.039264
iteration 1800 / 2000: loss 2.078826
iteration 1900 / 2000: loss 2.105300
iteration 0 / 2000: loss 314.034053
iteration 100 / 2000: loss 209.668082
iteration 200 / 2000: loss 140.836276
iteration 300 / 2000: loss 94.571126
iteration 400 / 2000: loss 63.976016
iteration 500 / 2000: loss 43.379349
iteration 600 / 2000: loss 29.696000
iteration 700 / 2000: loss 20.609397
```

```
iteration 800 / 2000: loss 14.361890
iteration 900 / 2000: loss 10.294571
iteration 1000 / 2000: loss 7.531748
iteration 1100 / 2000: loss 5.679744
iteration 1200 / 2000: loss 4.473326
iteration 1300 / 2000: loss 3.698369
iteration 1400 / 2000: loss 3.101059
iteration 1500 / 2000: loss 2.811788
iteration 1600 / 2000: loss 2.405102
iteration 1700 / 2000: loss 2.297056
iteration 1800 / 2000: loss 2.192218
iteration 1900 / 2000: loss 2.098240
iteration 0 / 2000: loss 3114.373469
iteration 100 / 2000: loss 56.756494
iteration 200 / 2000: loss 3.155146
iteration 300 / 2000: loss 2.176664
iteration 400 / 2000: loss 2.209786
iteration 500 / 2000: loss 2.209469
iteration 600 / 2000: loss 2.173674
iteration 700 / 2000: loss 2.193283
iteration 800 / 2000: loss 2.211232
iteration 900 / 2000: loss 2.153244
iteration 1000 / 2000: loss 2.222171
iteration 1100 / 2000: loss 2.183503
iteration 1200 / 2000: loss 2.230476
iteration 1300 / 2000: loss 2.195704
iteration 1400 / 2000: loss 2.203370
iteration 1500 / 2000: loss 2.227119
iteration 1600 / 2000: loss 2.183650
iteration 1700 / 2000: loss 2.166366
iteration 1800 / 2000: loss 2.230550
iteration 1900 / 2000: loss 2.194127
iteration 0 / 2000: loss 1516.281180
iteration 100 / 2000: loss 204.387071
iteration 200 / 2000: loss 29.090883
iteration 300 / 2000: loss 5.743999
iteration 400 / 2000: loss 2.617286
iteration 500 / 2000: loss 2.216886
iteration 600 / 2000: loss 2.165445
iteration 700 / 2000: loss 2.184068
iteration 800 / 2000: loss 2.074803
iteration 900 / 2000: loss 2.162137
iteration 1000 / 2000: loss 2.121434
iteration 1100 / 2000: loss 2.188956
iteration 1200 / 2000: loss 2.131676
iteration 1300 / 2000: loss 2.153548
iteration 1400 / 2000: loss 2.117342
iteration 1500 / 2000: loss 2.156037
```

```
iteration 1600 / 2000: loss 2.084186
iteration 1700 / 2000: loss 2.140500
iteration 1800 / 2000: loss 2.122291
iteration 1900 / 2000: loss 2.130224
iteration 0 / 2000: loss 609.246963
iteration 100 / 2000: loss 272.865002
iteration 200 / 2000: loss 123.192563
iteration 300 / 2000: loss 56.184211
iteration 400 / 2000: loss 26.393845
iteration 500 / 2000: loss 12.965286
iteration 600 / 2000: loss 6.911579
iteration 700 / 2000: loss 4.287842
iteration 800 / 2000: loss 3.047641
iteration 900 / 2000: loss 2.499415
iteration 1000 / 2000: loss 2.247541
iteration 1100 / 2000: loss 2.168297
iteration 1200 / 2000: loss 2.090155
iteration 1300 / 2000: loss 2.084578
iteration 1400 / 2000: loss 2.072779
iteration 1500 / 2000: loss 2.018628
iteration 1600 / 2000: loss 2.033742
iteration 1700 / 2000: loss 2.032286
iteration 1800 / 2000: loss 2.054533
iteration 1900 / 2000: loss 2.035691
iteration 0 / 2000: loss 767.668339
iteration 100 / 2000: loss 2.146283
iteration 200 / 2000: loss 2.079942
iteration 300 / 2000: loss 2.148486
iteration 400 / 2000: loss 2.121626
iteration 500 / 2000: loss 2.104362
iteration 600 / 2000: loss 2.037648
iteration 700 / 2000: loss 2.129105
iteration 800 / 2000: loss 2.097862
iteration 900 / 2000: loss 2.071540
iteration 1000 / 2000: loss 2.121357
iteration 1100 / 2000: loss 2.116656
iteration 1200 / 2000: loss 2.101435
iteration 1300 / 2000: loss 2.063935
iteration 1400 / 2000: loss 2.100124
iteration 1500 / 2000: loss 2.108111
iteration 1600 / 2000: loss 2.028468
iteration 1700 / 2000: loss 2.101073
iteration 1800 / 2000: loss 2.125178
iteration 1900 / 2000: loss 2.110056
iteration 0 / 2000: loss 317.433271
iteration 100 / 2000: loss 7.392277
iteration 200 / 2000: loss 2.102851
iteration 300 / 2000: loss 2.020741
```

```
iteration 400 / 2000: loss 2.061295
iteration 500 / 2000: loss 2.072838
iteration 600 / 2000: loss 2.024821
iteration 700 / 2000: loss 1.924324
iteration 800 / 2000: loss 1.910913
iteration 900 / 2000: loss 1.992765
iteration 1000 / 2000: loss 2.006124
iteration 1100 / 2000: loss 2.029742
iteration 1200 / 2000: loss 1.995583
iteration 1300 / 2000: loss 1.939098
iteration 1400 / 2000: loss 2.100580
iteration 1500 / 2000: loss 1.989563
iteration 1600 / 2000: loss 1.970179
iteration 1700 / 2000: loss 2.059839
iteration 1800 / 2000: loss 2.085870
iteration 1900 / 2000: loss 2.007583
iteration 0 / 2000: loss 3062.411750
iteration 100 / 2000: loss 2.167190
iteration 200 / 2000: loss 2.201668
iteration 300 / 2000: loss 2.222707
iteration 400 / 2000: loss 2.199126
iteration 500 / 2000: loss 2.224243
iteration 600 / 2000: loss 2.191648
iteration 700 / 2000: loss 2.216292
iteration 800 / 2000: loss 2.215135
iteration 900 / 2000: loss 2.193087
iteration 1000 / 2000: loss 2.219787
iteration 1100 / 2000: loss 2.185059
iteration 1200 / 2000: loss 2.182145
iteration 1300 / 2000: loss 2.224181
iteration 1400 / 2000: loss 2.203852
iteration 1500 / 2000: loss 2.218446
iteration 1600 / 2000: loss 2.202184
iteration 1700 / 2000: loss 2.262620
iteration 1800 / 2000: loss 2.206187
iteration 1900 / 2000: loss 2.216176
iteration 0 / 2000: loss 1553.984239
iteration 100 / 2000: loss 2.152099
iteration 200 / 2000: loss 2.219796
iteration 300 / 2000: loss 2.198998
iteration 400 / 2000: loss 2.134961
iteration 500 / 2000: loss 2.141926
iteration 600 / 2000: loss 2.162200
iteration 700 / 2000: loss 2.135865
iteration 800 / 2000: loss 2.122476
iteration 900 / 2000: loss 2.142825
iteration 1000 / 2000: loss 2.163530
iteration 1100 / 2000: loss 2.151403
```

```
iteration 1200 / 2000: loss 2.151367
iteration 1300 / 2000: loss 2.124408
iteration 1400 / 2000: loss 2.139904
iteration 1500 / 2000: loss 2.127859
iteration 1600 / 2000: loss 2.153639
iteration 1700 / 2000: loss 2.098877
iteration 1800 / 2000: loss 2.162105
iteration 1900 / 2000: loss 2.158314
iteration 0 / 2000: loss 609.459593
iteration 100 / 2000: loss 2.237641
iteration 200 / 2000: loss 2.079269
iteration 300 / 2000: loss 2.024663
iteration 400 / 2000: loss 2.085404
iteration 500 / 2000: loss 2.121909
iteration 600 / 2000: loss 2.001103
iteration 700 / 2000: loss 2.056583
iteration 800 / 2000: loss 2.038227
iteration 900 / 2000: loss 2.081232
iteration 1000 / 2000: loss 2.075952
iteration 1100 / 2000: loss 2.163221
iteration 1200 / 2000: loss 2.093546
iteration 1300 / 2000: loss 2.104708
iteration 1400 / 2000: loss 2.095095
iteration 1500 / 2000: loss 2.071706
iteration 1600 / 2000: loss 2.063920
iteration 1700 / 2000: loss 2.008081
iteration 1800 / 2000: loss 2.079163
iteration 1900 / 2000: loss 2.024103
iteration 0 / 2000: loss 775.978236
iteration 100 / 2000: loss 6.822712
iteration 200 / 2000: loss 8.791060
iteration 300 / 2000: loss 5.719057
iteration 400 / 2000: loss 8.761065
iteration 500 / 2000: loss 5.767170
iteration 600 / 2000: loss 9.080091
iteration 700 / 2000: loss 8.191508
iteration 800 / 2000: loss 7.375419
iteration 900 / 2000: loss 6.807149
iteration 1000 / 2000: loss 7.420195
iteration 1100 / 2000: loss 9.045980
iteration 1200 / 2000: loss 6.691352
iteration 1300 / 2000: loss 6.676311
iteration 1400 / 2000: loss 8.307029
iteration 1500 / 2000: loss 9.976482
iteration 1600 / 2000: loss 9.270798
iteration 1700 / 2000: loss 8.793627
iteration 1800 / 2000: loss 6.382784
iteration 1900 / 2000: loss 9.523693
```

```
iteration 0 / 2000: loss 317.000503
iteration 100 / 2000: loss 5.995550
iteration 200 / 2000: loss 5.185945
iteration 300 / 2000: loss 4.807756
iteration 400 / 2000: loss 5.964433
iteration 500 / 2000: loss 3.851257
iteration 600 / 2000: loss 4.196610
iteration 700 / 2000: loss 4.475509
iteration 800 / 2000: loss 6.220701
iteration 900 / 2000: loss 5.438259
iteration 1000 / 2000: loss 5.936206
iteration 1100 / 2000: loss 5.315333
iteration 1200 / 2000: loss 5.721054
iteration 1300 / 2000: loss 5.154116
iteration 1400 / 2000: loss 4.968878
iteration 1500 / 2000: loss 3.850163
iteration 1600 / 2000: loss 3.095071
iteration 1700 / 2000: loss 5.388440
iteration 1800 / 2000: loss 4.471393
iteration 1900 / 2000: loss 4.867879
iteration 0 / 2000: loss 3044.598068
```

```
/content/drive/MyDrive/cs231n/assignments/assignment1/cs231n/classifiers/softmax
.py:80: RuntimeWarning: divide by zero encountered in log
  loss = -np.sum(np.log(probability_correct))
```

```
iteration 100 / 2000: loss inf
iteration 200 / 2000: loss inf
iteration 300 / 2000: loss inf
iteration 400 / 2000: loss inf
iteration 500 / 2000: loss inf
iteration 600 / 2000: loss inf
iteration 700 / 2000: loss inf
iteration 800 / 2000: loss inf
iteration 900 / 2000: loss inf
iteration 1000 / 2000: loss inf
iteration 1100 / 2000: loss inf
iteration 1200 / 2000: loss inf
iteration 1300 / 2000: loss inf
iteration 1400 / 2000: loss inf
iteration 1500 / 2000: loss inf
iteration 1600 / 2000: loss inf
iteration 1700 / 2000: loss inf
iteration 1800 / 2000: loss inf
iteration 1900 / 2000: loss inf
iteration 0 / 2000: loss 1535.605908
iteration 100 / 2000: loss 14.593007
iteration 200 / 2000: loss 14.829779
iteration 300 / 2000: loss 15.513755
```

```
iteration 400 / 2000: loss 15.557831
iteration 500 / 2000: loss 16.520890
iteration 600 / 2000: loss 20.108196
iteration 700 / 2000: loss 18.451642
iteration 800 / 2000: loss 14.158580
iteration 900 / 2000: loss 18.756424
iteration 1000 / 2000: loss 13.519297
iteration 1100 / 2000: loss 16.428768
iteration 1200 / 2000: loss 17.552947
iteration 1300 / 2000: loss 15.868181
iteration 1400 / 2000: loss 12.550853
iteration 1500 / 2000: loss 20.553423
iteration 1600 / 2000: loss 17.823394
iteration 1700 / 2000: loss 14.668853
iteration 1800 / 2000: loss 16.069592
iteration 1900 / 2000: loss 14.560742
iteration 0 / 2000: loss 617.958352
iteration 100 / 2000: loss 8.009940
iteration 200 / 2000: loss 8.758676
iteration 300 / 2000: loss 5.957853
iteration 400 / 2000: loss 7.875189
iteration 500 / 2000: loss 5.700415
iteration 600 / 2000: loss 7.752054
iteration 700 / 2000: loss 6.171818
iteration 800 / 2000: loss 8.853443
iteration 900 / 2000: loss 6.630304
iteration 1000 / 2000: loss 8.262494
iteration 1100 / 2000: loss 6.384973
iteration 1200 / 2000: loss 6.526137
iteration 1300 / 2000: loss 6.123632
iteration 1400 / 2000: loss 7.991898
iteration 1500 / 2000: loss 6.267841
iteration 1600 / 2000: loss 6.313243
iteration 1700 / 2000: loss 7.377915
iteration 1800 / 2000: loss 7.429677
iteration 1900 / 2000: loss 6.225775
iteration 0 / 2000: loss 768.637600
iteration 100 / 2000: loss 695.627316
iteration 200 / 2000: loss 628.629395
iteration 300 / 2000: loss 568.903055
iteration 400 / 2000: loss 514.855482
iteration 500 / 2000: loss 465.990695
iteration 600 / 2000: loss 421.327401
iteration 700 / 2000: loss 381.237451
iteration 800 / 2000: loss 344.970096
iteration 900 / 2000: loss 312.236223
iteration 1000 / 2000: loss 282.818205
iteration 1100 / 2000: loss 255.968971
```

```
iteration 1200 / 2000: loss 231.685330
iteration 1300 / 2000: loss 209.784227
iteration 1400 / 2000: loss 189.911503
iteration 1500 / 2000: loss 171.968070
iteration 1600 / 2000: loss 155.712746
iteration 1700 / 2000: loss 140.983610
iteration 1800 / 2000: loss 127.677531
iteration 1900 / 2000: loss 115.903203
iteration 0 / 2000: loss 312.569637
iteration 100 / 2000: loss 299.438875
iteration 200 / 2000: loss 288.298727
iteration 300 / 2000: loss 276.167681
iteration 400 / 2000: loss 265.407975
iteration 500 / 2000: loss 254.455683
iteration 600 / 2000: loss 244.475871
iteration 700 / 2000: loss 235.206320
iteration 800 / 2000: loss 225.671200
iteration 900 / 2000: loss 216.636607
iteration 1000 / 2000: loss 208.063550
iteration 1100 / 2000: loss 200.077394
iteration 1200 / 2000: loss 191.995759
iteration 1300 / 2000: loss 184.756992
iteration 1400 / 2000: loss 177.562103
iteration 1500 / 2000: loss 170.560182
iteration 1600 / 2000: loss 163.687243
iteration 1700 / 2000: loss 157.422107
iteration 1800 / 2000: loss 151.145497
iteration 1900 / 2000: loss 145.431136
iteration 0 / 2000: loss 3058.701438
iteration 100 / 2000: loss 2049.589321
iteration 200 / 2000: loss 1373.589069
iteration 300 / 2000: loss 920.614864
iteration 400 / 2000: loss 617.279285
iteration 500 / 2000: loss 414.151955
iteration 600 / 2000: loss 278.104059
iteration 700 / 2000: loss 186.948548
iteration 800 / 2000: loss 125.935105
iteration 900 / 2000: loss 85.066226
iteration 1000 / 2000: loss 57.714632
iteration 1100 / 2000: loss 39.386286
iteration 1200 / 2000: loss 27.049832
iteration 1300 / 2000: loss 18.828309
iteration 1400 / 2000: loss 13.389339
iteration 1500 / 2000: loss 9.678942
iteration 1600 / 2000: loss 7.201401
iteration 1700 / 2000: loss 5.528673
iteration 1800 / 2000: loss 4.411291
iteration 1900 / 2000: loss 3.689568
```

```
iteration 0 / 2000: loss 1552.453182
iteration 100 / 2000: loss 1270.989794
iteration 200 / 2000: loss 1040.326972
iteration 300 / 2000: loss 851.616403
iteration 400 / 2000: loss 697.697268
iteration 500 / 2000: loss 571.099234
iteration 600 / 2000: loss 467.713473
iteration 700 / 2000: loss 383.098914
iteration 800 / 2000: loss 313.872530
iteration 900 / 2000: loss 257.366181
iteration 1000 / 2000: loss 211.142157
iteration 1100 / 2000: loss 173.041893
iteration 1200 / 2000: loss 142.058283
iteration 1300 / 2000: loss 116.700414
iteration 1400 / 2000: loss 95.871032
iteration 1500 / 2000: loss 78.775422
iteration 1600 / 2000: loss 64.874721
iteration 1700 / 2000: loss 53.474931
iteration 1800 / 2000: loss 44.124665
iteration 1900 / 2000: loss 36.480654
iteration 0 / 2000: loss 617.336677
iteration 100 / 2000: loss 569.471460
iteration 200 / 2000: loss 525.710773
iteration 300 / 2000: loss 485.432062
iteration 400 / 2000: loss 448.269862
iteration 500 / 2000: loss 413.491764
iteration 600 / 2000: loss 381.842283
iteration 700 / 2000: loss 352.555365
iteration 800 / 2000: loss 325.563356
iteration 900 / 2000: loss 300.741293
iteration 1000 / 2000: loss 277.466374
iteration 1100 / 2000: loss 256.290175
iteration 1200 / 2000: loss 236.755953
iteration 1300 / 2000: loss 218.623987
iteration 1400 / 2000: loss 201.868439
iteration 1500 / 2000: loss 186.469212
iteration 1600 / 2000: loss 172.313679
iteration 1700 / 2000: loss 159.055107
iteration 1800 / 2000: loss 146.919167
iteration 1900 / 2000: loss 135.685366
iteration 0 / 2000: loss 769.252280
iteration 100 / 2000: loss 761.651905
iteration 200 / 2000: loss 754.157998
iteration 300 / 2000: loss 746.533124
iteration 400 / 2000: loss 739.227590
iteration 500 / 2000: loss 731.976602
iteration 600 / 2000: loss 724.604357
iteration 700 / 2000: loss 717.242168
```

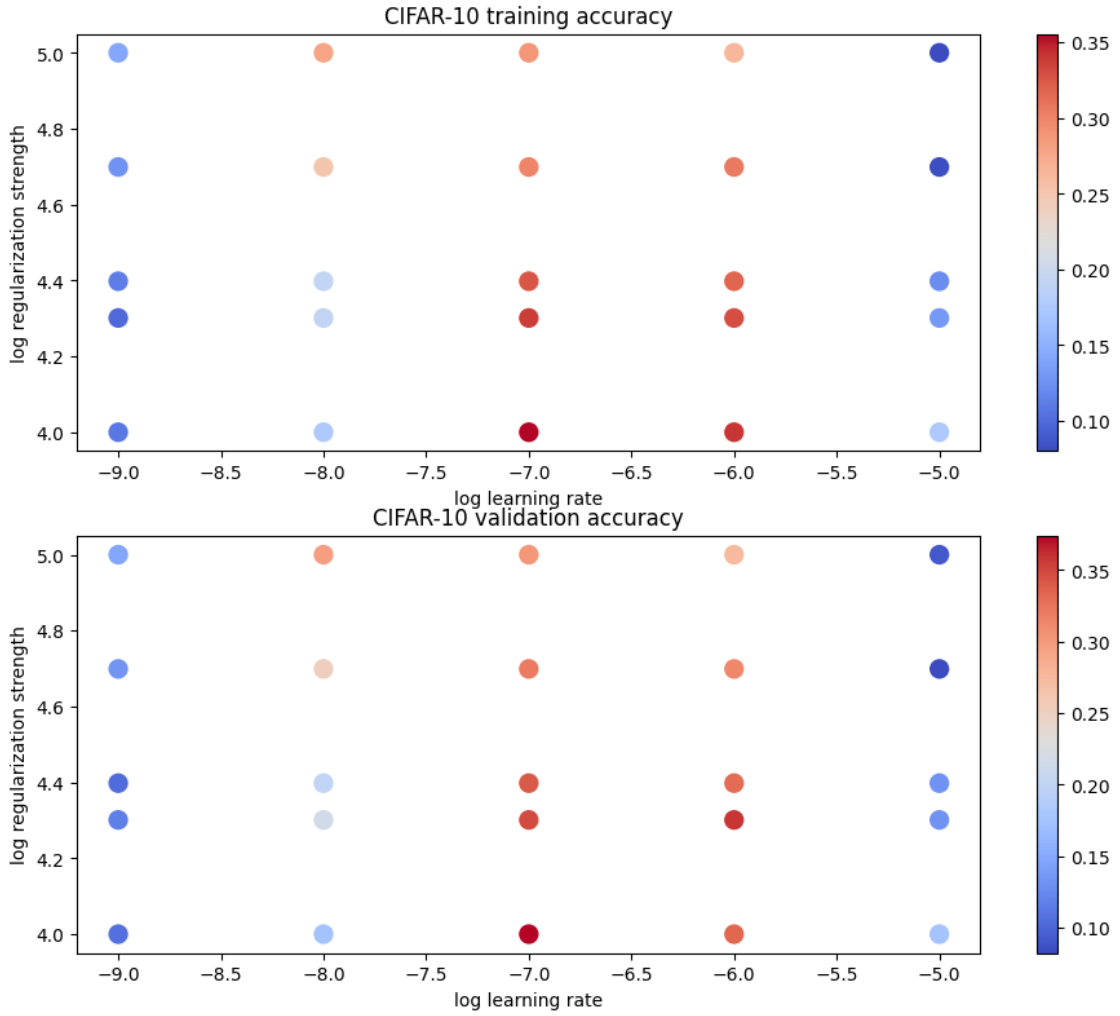
```
iteration 800 / 2000: loss 710.076456
iteration 900 / 2000: loss 702.904184
iteration 1000 / 2000: loss 695.945016
iteration 1100 / 2000: loss 689.266611
iteration 1200 / 2000: loss 681.953154
iteration 1300 / 2000: loss 675.286270
iteration 1400 / 2000: loss 668.367909
iteration 1500 / 2000: loss 662.106200
iteration 1600 / 2000: loss 655.373116
iteration 1700 / 2000: loss 648.769365
iteration 1800 / 2000: loss 642.484363
iteration 1900 / 2000: loss 635.916669
iteration 0 / 2000: loss 316.988146
iteration 100 / 2000: loss 315.584759
iteration 200 / 2000: loss 314.388333
iteration 300 / 2000: loss 312.758631
iteration 400 / 2000: loss 311.698455
iteration 500 / 2000: loss 310.436076
iteration 600 / 2000: loss 309.153933
iteration 700 / 2000: loss 307.866300
iteration 800 / 2000: loss 306.486757
iteration 900 / 2000: loss 305.399077
iteration 1000 / 2000: loss 304.538544
iteration 1100 / 2000: loss 303.052667
iteration 1200 / 2000: loss 301.635545
iteration 1300 / 2000: loss 300.202993
iteration 1400 / 2000: loss 299.129693
iteration 1500 / 2000: loss 298.095276
iteration 1600 / 2000: loss 297.486441
iteration 1700 / 2000: loss 295.144064
iteration 1800 / 2000: loss 294.363327
iteration 1900 / 2000: loss 292.981950
iteration 0 / 2000: loss 3138.319306
iteration 100 / 2000: loss 3015.123158
iteration 200 / 2000: loss 2897.657569
iteration 300 / 2000: loss 2783.411067
iteration 400 / 2000: loss 2674.471466
iteration 500 / 2000: loss 2569.455181
iteration 600 / 2000: loss 2468.859020
iteration 700 / 2000: loss 2371.843720
iteration 800 / 2000: loss 2279.210703
iteration 900 / 2000: loss 2190.109052
iteration 1000 / 2000: loss 2104.039493
iteration 1100 / 2000: loss 2021.405530
iteration 1200 / 2000: loss 1942.652393
iteration 1300 / 2000: loss 1865.886783
iteration 1400 / 2000: loss 1792.977051
iteration 1500 / 2000: loss 1722.974466
```

```
iteration 1600 / 2000: loss 1654.972491
iteration 1700 / 2000: loss 1590.059908
iteration 1800 / 2000: loss 1527.969515
iteration 1900 / 2000: loss 1467.894177
iteration 0 / 2000: loss 1547.525202
iteration 100 / 2000: loss 1517.322988
iteration 200 / 2000: loss 1487.114792
iteration 300 / 2000: loss 1457.247193
iteration 400 / 2000: loss 1428.857545
iteration 500 / 2000: loss 1400.607157
iteration 600 / 2000: loss 1372.474829
iteration 700 / 2000: loss 1345.827465
iteration 800 / 2000: loss 1318.879349
iteration 900 / 2000: loss 1292.804739
iteration 1000 / 2000: loss 1267.010569
iteration 1100 / 2000: loss 1241.721196
iteration 1200 / 2000: loss 1217.183223
iteration 1300 / 2000: loss 1193.019375
iteration 1400 / 2000: loss 1169.819394
iteration 1500 / 2000: loss 1146.177814
iteration 1600 / 2000: loss 1123.457290
iteration 1700 / 2000: loss 1101.295590
iteration 1800 / 2000: loss 1079.923377
iteration 1900 / 2000: loss 1058.285111
iteration 0 / 2000: loss 618.965043
iteration 100 / 2000: loss 614.285585
iteration 200 / 2000: loss 609.460182
iteration 300 / 2000: loss 604.445723
iteration 400 / 2000: loss 599.483593
iteration 500 / 2000: loss 594.607859
iteration 600 / 2000: loss 589.787590
iteration 700 / 2000: loss 585.529238
iteration 800 / 2000: loss 580.741524
iteration 900 / 2000: loss 575.855263
iteration 1000 / 2000: loss 571.420111
iteration 1100 / 2000: loss 566.877404
iteration 1200 / 2000: loss 562.015132
iteration 1300 / 2000: loss 557.685598
iteration 1400 / 2000: loss 553.216004
iteration 1500 / 2000: loss 548.595214
iteration 1600 / 2000: loss 544.363280
iteration 1700 / 2000: loss 540.009223
iteration 1800 / 2000: loss 535.633350
iteration 1900 / 2000: loss 531.267078
lr 1.000000e-09 reg 1.000000e+04 train accuracy: 0.108673 val accuracy: 0.107000
lr 1.000000e-09 reg 2.000000e+04 train accuracy: 0.098673 val accuracy: 0.117000
lr 1.000000e-09 reg 2.500000e+04 train accuracy: 0.112204 val accuracy: 0.103000
lr 1.000000e-09 reg 5.000000e+04 train accuracy: 0.127306 val accuracy: 0.133000
```

```
lr 1.000000e-09 reg 1.000000e+05 train accuracy: 0.142347 val accuracy: 0.147000
lr 1.000000e-08 reg 1.000000e+04 train accuracy: 0.175306 val accuracy: 0.174000
lr 1.000000e-08 reg 2.000000e+04 train accuracy: 0.191551 val accuracy: 0.215000
lr 1.000000e-08 reg 2.500000e+04 train accuracy: 0.192061 val accuracy: 0.199000
lr 1.000000e-08 reg 5.000000e+04 train accuracy: 0.251714 val accuracy: 0.252000
lr 1.000000e-08 reg 1.000000e+05 train accuracy: 0.279061 val accuracy: 0.297000
lr 1.000000e-07 reg 1.000000e+04 train accuracy: 0.355612 val accuracy: 0.374000
lr 1.000000e-07 reg 2.000000e+04 train accuracy: 0.337510 val accuracy: 0.348000
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.325612 val accuracy: 0.340000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.300367 val accuracy: 0.321000
lr 1.000000e-07 reg 1.000000e+05 train accuracy: 0.288551 val accuracy: 0.303000
lr 1.000000e-06 reg 1.000000e+04 train accuracy: 0.340694 val accuracy: 0.334000
lr 1.000000e-06 reg 2.000000e+04 train accuracy: 0.330408 val accuracy: 0.359000
lr 1.000000e-06 reg 2.500000e+04 train accuracy: 0.318122 val accuracy: 0.330000
lr 1.000000e-06 reg 5.000000e+04 train accuracy: 0.306857 val accuracy: 0.313000
lr 1.000000e-06 reg 1.000000e+05 train accuracy: 0.264612 val accuracy: 0.274000
lr 1.000000e-05 reg 1.000000e+04 train accuracy: 0.173857 val accuracy: 0.175000
lr 1.000000e-05 reg 2.000000e+04 train accuracy: 0.131143 val accuracy: 0.132000
lr 1.000000e-05 reg 2.500000e+04 train accuracy: 0.122816 val accuracy: 0.130000
lr 1.000000e-05 reg 5.000000e+04 train accuracy: 0.082041 val accuracy: 0.082000
lr 1.000000e-05 reg 1.000000e+05 train accuracy: 0.079980 val accuracy: 0.090000
best validation accuracy achieved during cross-validation: 0.374000
```

[23]:
```python
# Visualize the cross-validation results
import math
import pdb

# pdb.set_trace()

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

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

# plot validation accuracy
colors = [results[x][1] for x in results] # default size of markers is 20
plt.subplot(2, 1, 2)
plt.scatter(x_scatter, y_scatter, marker_size, c=colors, cmap=plt.cm.coolwarm)
```

```
plt.colorbar()
plt.xlabel('log learning rate')
plt.ylabel('log regularization strength')
plt.title('CIFAR-10 validation accuracy')
plt.show()
```



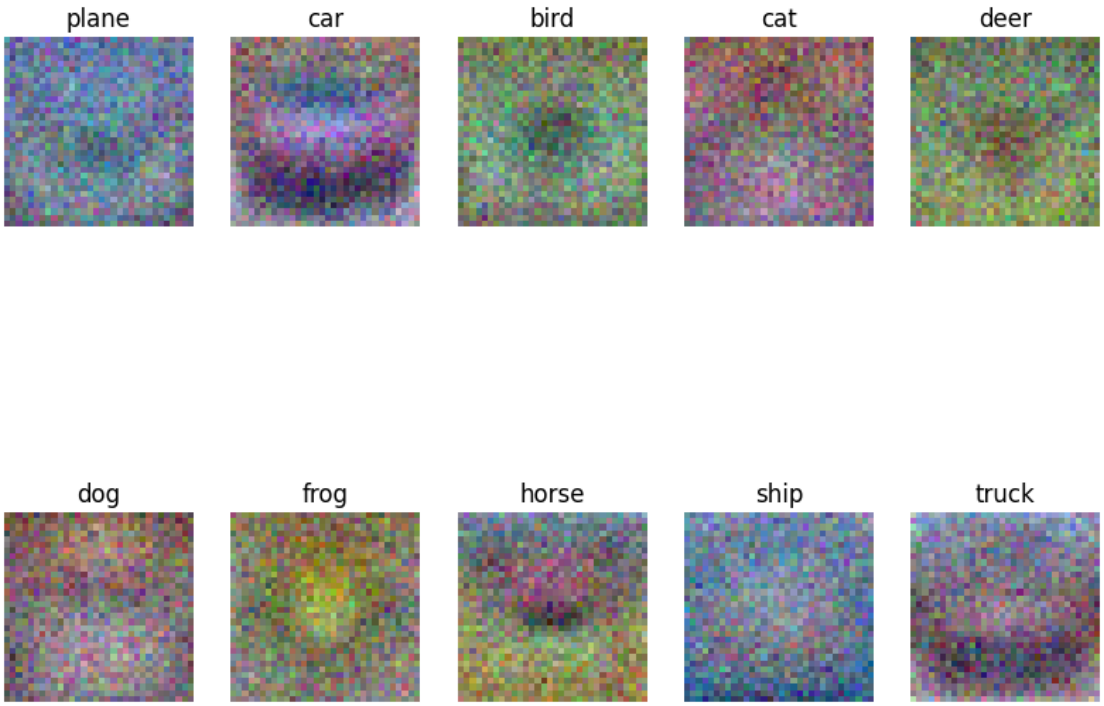
[24]:
```
# Evaluate the best softmax on test set
y_test_pred = best_softmax.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('Softmax classifier on raw pixels final test set accuracy: %f' %␣
  ↪test_accuracy)
```

```
Softmax classifier on raw pixels final test set accuracy: 0.360000
```

[25]:
```
# Save best softmax model
best_softmax.save("best_softmax.npy")
```

```
best_softmax.npy saved.
```

[26]:
```
# 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 : The prediction for the class is given by the dot product of the weights vector and the image, we want the prediction for the class to be the highest, so there is some image where if the weights is multiplied by it, it gives the highest class score. As W has the same dimensions as the images, we can reshape it into an image where multiplying them gives the highest dot product (as a vector's dot product is highest with itself).

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 : Yes, as softmax simply cares that the difference of the class score between correct class and all other classes is greater than some number. Softmax always changes with new datapoints, whilst svm is piecewise so the new point could contribute exactly 0 or 1 (0 wouldnt change svm loss).

[ ]:

two_layer_net

February 3, 2026

[ ]:
```
# 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.

[ ]:
```
!pip -q install -U ipython
```

```
                        0.0/621.4 kB
? eta -:--:--
                      614.4/621.4
kB 19.6 MB/s eta 0:00:01
                      621.4/621.4 kB
12.5 MB/s eta 0:00:00
                        1.6/1.6 MB
43.0 MB/s eta 0:00:00
                        85.4/85.4 kB
4.6 MB/s eta 0:00:00
ERROR: pip's dependency resolver does not currently take into account
all the packages that are installed. This behaviour is the source of the
following dependency conflicts.
google-colab 1.0.0 requires ipython==7.34.0, but you have ipython 9.9.0 which is
incompatible.
```

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

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

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

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

2 Affine layer: forward

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

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

[ ]:
```
# Test the affine_forward function

num_inputs = 2
input_shape = (4, 5, 6)
```

```
output_dim = 3

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

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

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

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

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

3 Affine layer: backward

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

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

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

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

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

```
print('db error: ', rel_error(db_num, db))
```

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

4 ReLU activation: forward

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

[ ]:
```
# Test the relu_forward function

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

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

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

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

5 ReLU activation: backward

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

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

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

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

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

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

5.1 Inline Question 1:

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

YourAnswer : Mainly sigmoid and 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:

[ ]:
```
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:

[ ]:
```
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.

[26]:
```
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.

[41]:
```
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
model = TwoLayerNet(input_size, hidden_size, num_classes, weight_scale=0.001,␣
 ↪reg=1e-4)
solver = Solver(model, data,
                    update_rule='sgd',
                    optim_config={
                      'learning_rate': 3e-4,
                    },
                    num_epochs=15, batch_size=200,
                    print_every=100)

################################################################################
# TODO: Use a Solver instance to train a TwoLayerNet that achieves about 36% #
# accuracy on the validation set.                                              #
################################################################################
solver.train()
################################################################################
#                              END OF YOUR CODE                                #
################################################################################
```

```
(Iteration 1 / 3675) loss: 2.301191
(Epoch 0 / 15) train acc: 0.097000; val_acc: 0.106000
(Iteration 101 / 3675) loss: 2.071692
(Iteration 201 / 3675) loss: 1.938718
(Epoch 1 / 15) train acc: 0.321000; val_acc: 0.334000
(Iteration 301 / 3675) loss: 1.882014
```

```
(Iteration 401 / 3675) loss: 1.902843
(Epoch 2 / 15) train acc: 0.378000; val_acc: 0.402000
(Iteration 501 / 3675) loss: 1.728009
(Iteration 601 / 3675) loss: 1.682963
(Iteration 701 / 3675) loss: 1.675548
(Epoch 3 / 15) train acc: 0.440000; val_acc: 0.443000
(Iteration 801 / 3675) loss: 1.556401
(Iteration 901 / 3675) loss: 1.445823
(Epoch 4 / 15) train acc: 0.444000; val_acc: 0.464000
(Iteration 1001 / 3675) loss: 1.481548
(Iteration 1101 / 3675) loss: 1.654791
(Iteration 1201 / 3675) loss: 1.504394
(Epoch 5 / 15) train acc: 0.458000; val_acc: 0.470000
(Iteration 1301 / 3675) loss: 1.483563
(Iteration 1401 / 3675) loss: 1.457532
(Epoch 6 / 15) train acc: 0.458000; val_acc: 0.474000
(Iteration 1501 / 3675) loss: 1.491160
(Iteration 1601 / 3675) loss: 1.453714
(Iteration 1701 / 3675) loss: 1.573820
(Epoch 7 / 15) train acc: 0.476000; val_acc: 0.469000
(Iteration 1801 / 3675) loss: 1.451893
(Iteration 1901 / 3675) loss: 1.404205
(Epoch 8 / 15) train acc: 0.507000; val_acc: 0.465000
(Iteration 2001 / 3675) loss: 1.381133
(Iteration 2101 / 3675) loss: 1.459649
(Iteration 2201 / 3675) loss: 1.343543
(Epoch 9 / 15) train acc: 0.486000; val_acc: 0.478000
(Iteration 2301 / 3675) loss: 1.376479
(Iteration 2401 / 3675) loss: 1.421056
(Epoch 10 / 15) train acc: 0.497000; val_acc: 0.479000
(Iteration 2501 / 3675) loss: 1.289264
(Iteration 2601 / 3675) loss: 1.341333
(Epoch 11 / 15) train acc: 0.482000; val_acc: 0.483000
(Iteration 2701 / 3675) loss: 1.380545
(Iteration 2801 / 3675) loss: 1.297705
(Iteration 2901 / 3675) loss: 1.235475
(Epoch 12 / 15) train acc: 0.510000; val_acc: 0.492000
(Iteration 3001 / 3675) loss: 1.374557
(Iteration 3101 / 3675) loss: 1.306007
(Epoch 13 / 15) train acc: 0.522000; val_acc: 0.489000
(Iteration 3201 / 3675) loss: 1.335387
(Iteration 3301 / 3675) loss: 1.327401
(Iteration 3401 / 3675) loss: 1.336443
(Epoch 14 / 15) train acc: 0.506000; val_acc: 0.491000
(Iteration 3501 / 3675) loss: 1.145222
(Iteration 3601 / 3675) loss: 1.419452
(Epoch 15 / 15) train acc: 0.565000; val_acc: 0.494000
```

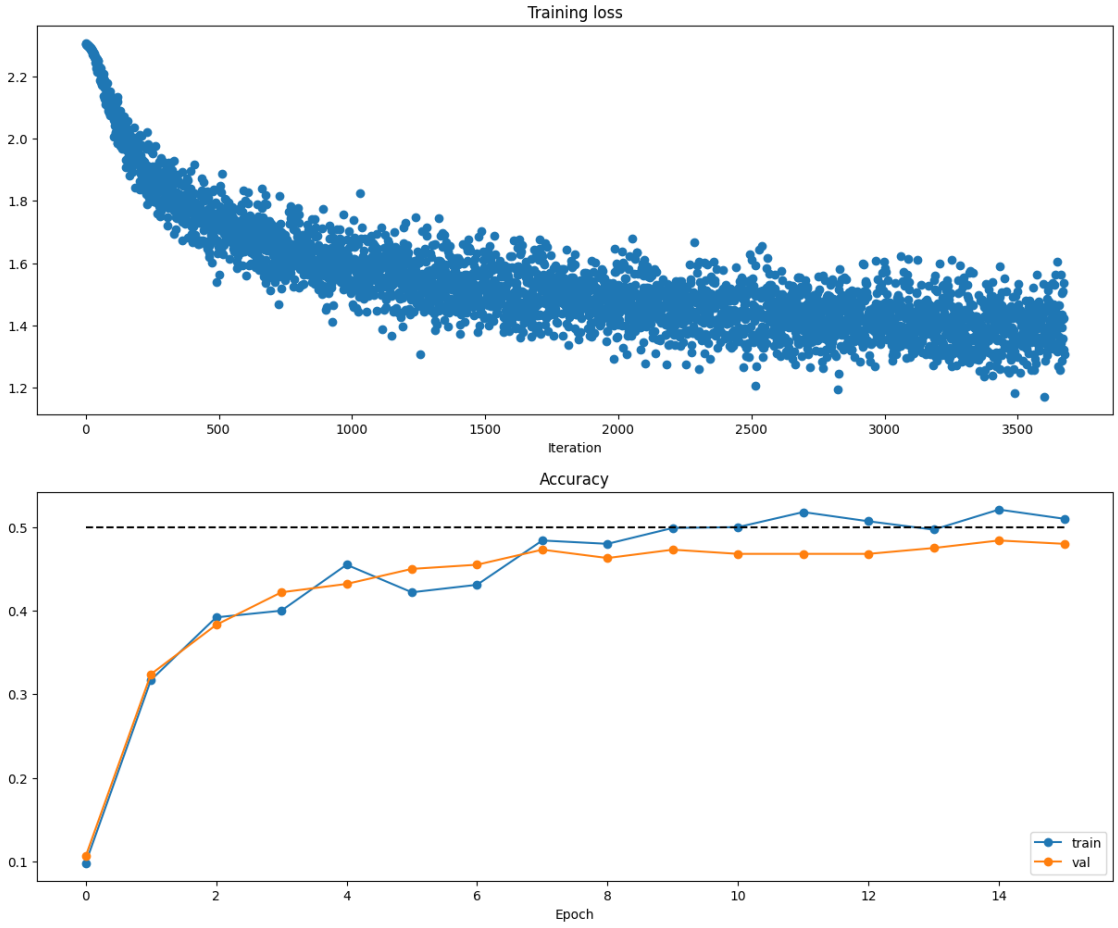
10 Debug the training

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.

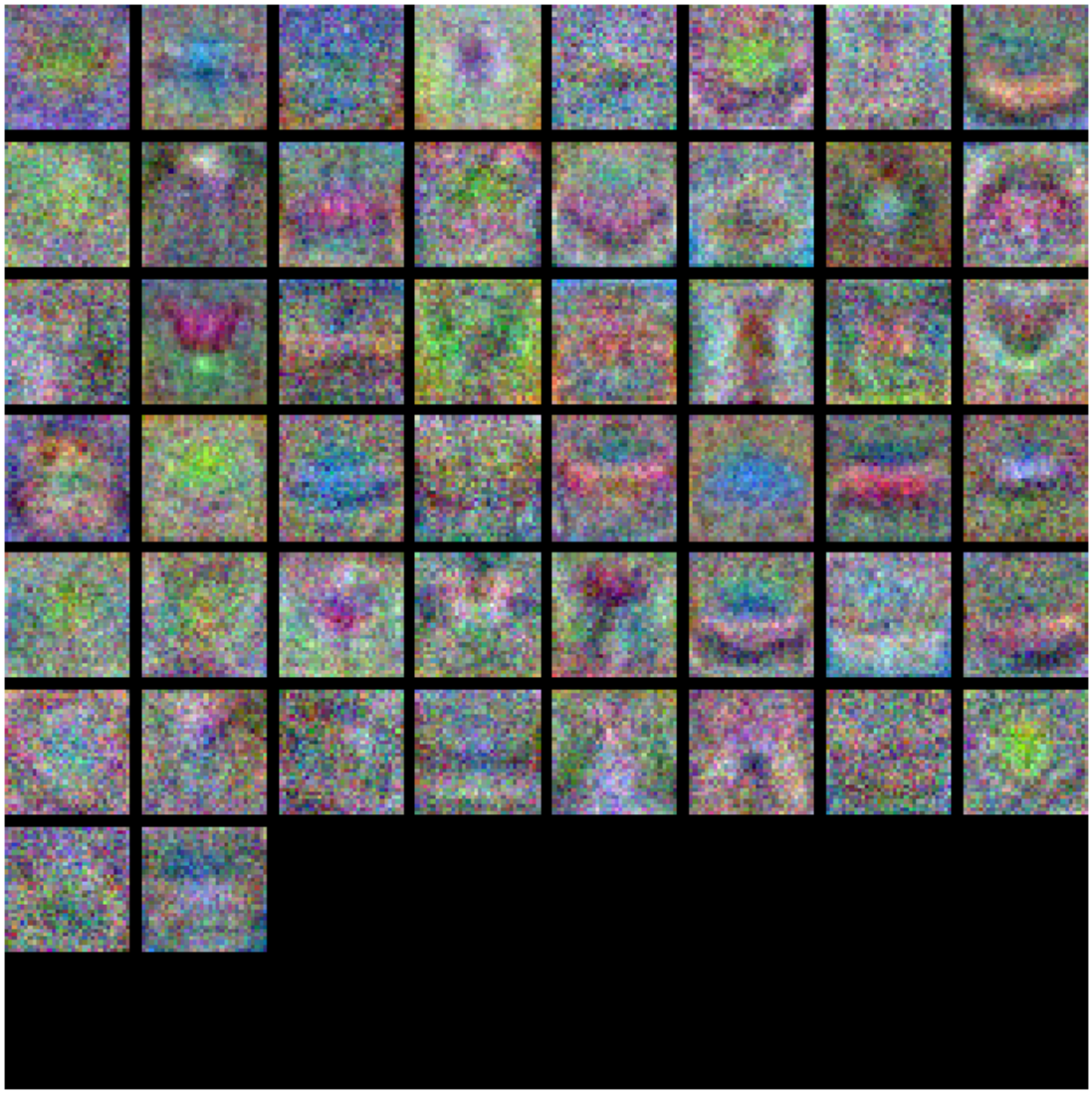
One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

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.

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



[36]:
```
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.).

[49]:
```
best_model = model



################################################################################
# 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.                       ␣
↪  #
################################################################################

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

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.

[50]:
```
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:  0.494
```

[51]:
```
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:  0.507
```

[52]:
```
# 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.

YourAnswer : Larger dataset, increased regularization strencth.

YourExplanation : For 1, larger dataset means a more variated training set, which more likely will represent the info presented in the test set.

As for 3, more regularization means that the model will favor less complex systems as the lecture explained. This means that it will likely refrain from overfitting which will lower the gap between train and test.

[41]:
```
```

features

February 3, 2026

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

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

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

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

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

[2]:
```
!pip -q install -U ipython
```

```
                        621.4/621.4 kB
9.9 MB/s eta 0:00:00
                        1.6/1.6 MB
34.8 MB/s eta 0:00:00
                        85.4/85.4 kB
4.7 MB/s eta 0:00:00
```

```
ERROR: pip's dependency resolver does not currently take into account
all the packages that are installed. This behaviour is the source of the
following dependency conflicts.
google-colab 1.0.0 requires ipython==7.34.0, but you have ipython 9.9.0 which is
incompatible.
```

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.

[9]:
```
# 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, 1e-5, 1e-4, 5e-6]
regularization_strengths = [5e4, 5e5, 5e3, 5e2, 6e4]

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 lr in learning_rates:
    for reg in regularization_strengths:
      softmax = Softmax()
      loss_hist = softmax.train(X_train_feats, y_train, learning_rate=lr,␣
 ↪reg=reg, num_iters=2000)

      y_train_pred = softmax.predict(X_train_feats)
      y_val_pred = softmax.predict(X_val_feats)

      train_acc = np.mean(y_train == y_train_pred)
      val_acc = np.mean(y_val == y_val_pred)
      results[(lr, reg)] = (train_acc, val_acc)

      if val_acc > best_val:
        best_val = val_acc
        best_softmax = softmax

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

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

```
lr 1.000000e-07 reg 5.000000e+02 train accuracy: 0.150816 val accuracy: 0.167000
lr 1.000000e-07 reg 5.000000e+03 train accuracy: 0.125735 val accuracy: 0.121000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.425245 val accuracy: 0.430000
lr 1.000000e-07 reg 6.000000e+04 train accuracy: 0.419776 val accuracy: 0.415000
lr 1.000000e-07 reg 5.000000e+05 train accuracy: 0.411347 val accuracy: 0.413000
lr 1.000000e-06 reg 5.000000e+02 train accuracy: 0.178224 val accuracy: 0.184000
lr 1.000000e-06 reg 5.000000e+03 train accuracy: 0.423000 val accuracy: 0.436000
lr 1.000000e-06 reg 5.000000e+04 train accuracy: 0.425755 val accuracy: 0.428000
lr 1.000000e-06 reg 6.000000e+04 train accuracy: 0.408000 val accuracy: 0.434000
lr 1.000000e-06 reg 5.000000e+05 train accuracy: 0.336408 val accuracy: 0.339000
lr 5.000000e-06 reg 5.000000e+02 train accuracy: 0.421857 val accuracy: 0.422000
lr 5.000000e-06 reg 5.000000e+03 train accuracy: 0.409776 val accuracy: 0.395000
lr 5.000000e-06 reg 5.000000e+04 train accuracy: 0.365898 val accuracy: 0.359000
lr 5.000000e-06 reg 6.000000e+04 train accuracy: 0.381449 val accuracy: 0.408000
lr 5.000000e-06 reg 5.000000e+05 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-05 reg 5.000000e+02 train accuracy: 0.420878 val accuracy: 0.420000
lr 1.000000e-05 reg 5.000000e+03 train accuracy: 0.409776 val accuracy: 0.397000
lr 1.000000e-05 reg 5.000000e+04 train accuracy: 0.328939 val accuracy: 0.326000
lr 1.000000e-05 reg 6.000000e+04 train accuracy: 0.308714 val accuracy: 0.325000
```

```
lr 1.000000e-05 reg 5.000000e+05 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-04 reg 5.000000e+02 train accuracy: 0.417755 val accuracy: 0.420000
lr 1.000000e-04 reg 5.000000e+03 train accuracy: 0.353980 val accuracy: 0.379000
lr 1.000000e-04 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-04 reg 6.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-04 reg 5.000000e+05 train accuracy: 0.100265 val accuracy: 0.087000
best validation accuracy achieved: 0.436000
```

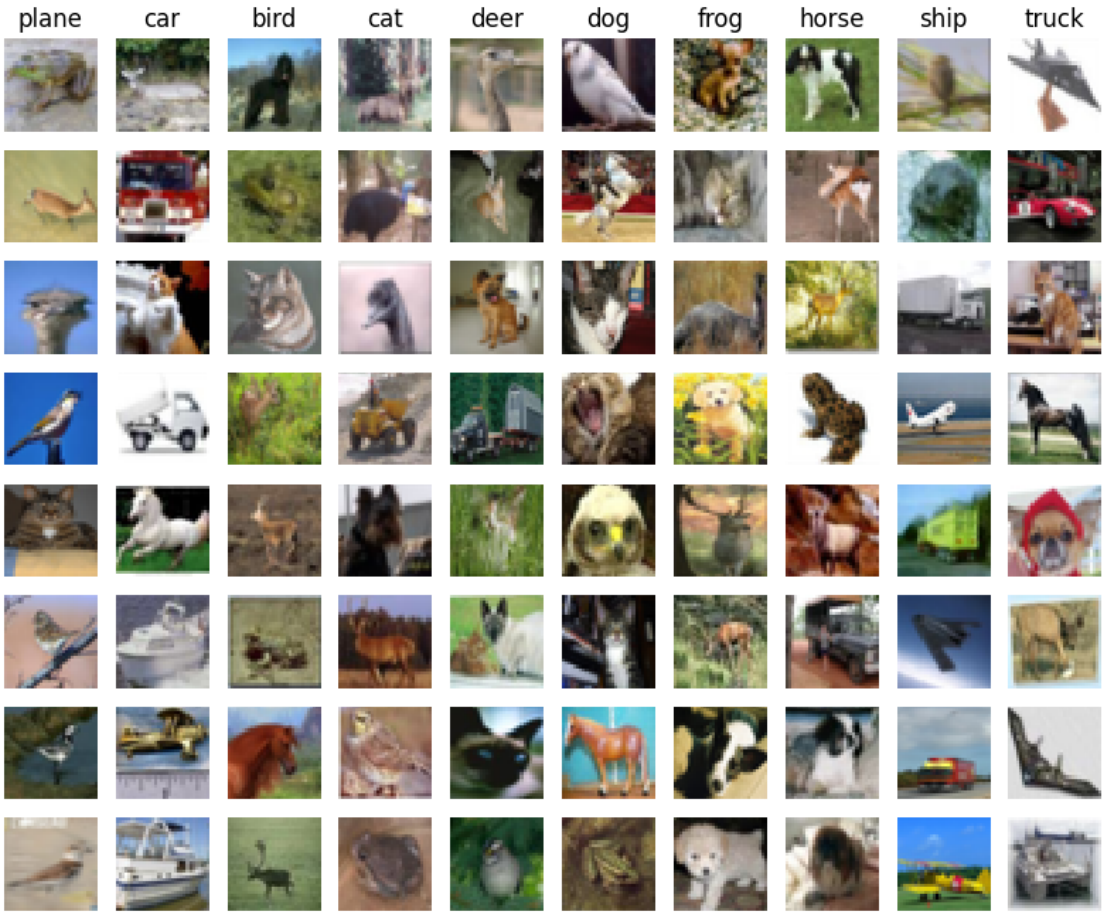
[10]:
```
# Evaluate your trained Softmax on the test set: you should be able to get at␣
 ↪least 0.42
y_test_pred = best_softmax.predict(X_test_feats)
test_accuracy = np.mean(y_test == y_test_pred)
print(test_accuracy)
```

```
0.421
```

[12]:
```
# Save best softmax model
best_softmax.save("best_softmax_features.npy")
```

```
best_softmax_features.npy saved.
```

[ ]:
```
# 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 : Honestly make very little sense why misclassified, although the images themselves are low quality. A few make sense, like a low quality boat looking like a truck from an angle.

1.4 Neural Network on image features

Earlier in this assigment we saw that training a two-layer neural network on raw pixels achieved better classification performance than linear classifiers on raw pixels. In this notebook we have seen that linear classifiers on image features outperform linear classifiers on raw pixels.

For completeness, we should also try training a neural network on image features. This approach should outperform all previous approaches: you should easily be able to achieve over 55% classification accuracy on the test set; our best model achieves about 60% classification accuracy.

[ ]:
```
# Preprocessing: Remove the bias dimension
# Make sure to run this cell only ONCE
print(X_train_feats.shape)
```

```
X_train_feats = X_train_feats[:, :-1]
X_val_feats = X_val_feats[:, :-1]
X_test_feats = X_test_feats[:, :-1]

print(X_train_feats.shape)
```

```
(49000, 170)
(49000, 169)
```

[ ]:
```
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.                                              #
################################################################################

solver = Solver(net, data,
                update_rule='sgd',
                optim_config={
                  'learning_rate': 2e-2
                }, lr_decay=0.98,
                num_epochs=50)
solver.train()

best_net = solver
```

```
(Iteration 1 / 24500) loss: 2.302630
(Epoch 0 / 50) train acc: 0.087000; val_acc: 0.085000
(Iteration 11 / 24500) loss: 2.302618
(Iteration 21 / 24500) loss: 2.302654
```

```
(Iteration 31 / 24500) loss: 2.302992
(Iteration 41 / 24500) loss: 2.302489
(Iteration 51 / 24500) loss: 2.302322
(Iteration 61 / 24500) loss: 2.301944
(Iteration 71 / 24500) loss: 2.302457
(Iteration 81 / 24500) loss: 2.302077
(Iteration 91 / 24500) loss: 2.301911
(Iteration 101 / 24500) loss: 2.302289
(Iteration 111 / 24500) loss: 2.302074
(Iteration 121 / 24500) loss: 2.301585
(Iteration 131 / 24500) loss: 2.301847
(Iteration 141 / 24500) loss: 2.302706
(Iteration 151 / 24500) loss: 2.302294
(Iteration 161 / 24500) loss: 2.302338
(Iteration 171 / 24500) loss: 2.301398
(Iteration 181 / 24500) loss: 2.301465
(Iteration 191 / 24500) loss: 2.301815
(Iteration 201 / 24500) loss: 2.301261
(Iteration 211 / 24500) loss: 2.300651
(Iteration 221 / 24500) loss: 2.300993
(Iteration 231 / 24500) loss: 2.300622
(Iteration 241 / 24500) loss: 2.300008
(Iteration 251 / 24500) loss: 2.299491
(Iteration 261 / 24500) loss: 2.299410
(Iteration 271 / 24500) loss: 2.299280
(Iteration 281 / 24500) loss: 2.298528
(Iteration 291 / 24500) loss: 2.297469
(Iteration 301 / 24500) loss: 2.296100
(Iteration 311 / 24500) loss: 2.296341
(Iteration 321 / 24500) loss: 2.295085
(Iteration 331 / 24500) loss: 2.293830
(Iteration 341 / 24500) loss: 2.294089
(Iteration 351 / 24500) loss: 2.291261
(Iteration 361 / 24500) loss: 2.291878
(Iteration 371 / 24500) loss: 2.285461
(Iteration 381 / 24500) loss: 2.284438
(Iteration 391 / 24500) loss: 2.281977
(Iteration 401 / 24500) loss: 2.285184
(Iteration 411 / 24500) loss: 2.276380
(Iteration 421 / 24500) loss: 2.275228
(Iteration 431 / 24500) loss: 2.269949
(Iteration 441 / 24500) loss: 2.265185
(Iteration 451 / 24500) loss: 2.256423
(Iteration 461 / 24500) loss: 2.252467
(Iteration 471 / 24500) loss: 2.261837
(Iteration 481 / 24500) loss: 2.240026
(Epoch 1 / 50) train acc: 0.269000; val_acc: 0.259000
(Iteration 491 / 24500) loss: 2.242098
```

```
(Iteration 501 / 24500) loss: 2.232383
(Iteration 511 / 24500) loss: 2.215367
(Iteration 521 / 24500) loss: 2.216822
(Iteration 531 / 24500) loss: 2.196363
(Iteration 541 / 24500) loss: 2.166394
(Iteration 551 / 24500) loss: 2.190293
(Iteration 561 / 24500) loss: 2.180665
(Iteration 571 / 24500) loss: 2.189870
(Iteration 581 / 24500) loss: 2.126435
(Iteration 591 / 24500) loss: 2.138463
(Iteration 601 / 24500) loss: 2.072349
(Iteration 611 / 24500) loss: 2.111948
(Iteration 621 / 24500) loss: 2.130170
(Iteration 631 / 24500) loss: 2.081466
(Iteration 641 / 24500) loss: 2.062874
(Iteration 651 / 24500) loss: 2.126819
(Iteration 661 / 24500) loss: 2.090806
(Iteration 671 / 24500) loss: 2.013124
(Iteration 681 / 24500) loss: 2.039642
(Iteration 691 / 24500) loss: 2.031870
(Iteration 701 / 24500) loss: 2.001051
(Iteration 711 / 24500) loss: 1.933487
(Iteration 721 / 24500) loss: 1.970631
(Iteration 731 / 24500) loss: 1.987006
(Iteration 741 / 24500) loss: 2.000693
(Iteration 751 / 24500) loss: 1.961344
(Iteration 761 / 24500) loss: 2.031996
(Iteration 771 / 24500) loss: 1.962184
(Iteration 781 / 24500) loss: 1.958885
(Iteration 791 / 24500) loss: 1.891769
(Iteration 801 / 24500) loss: 1.958133
(Iteration 811 / 24500) loss: 1.831306
(Iteration 821 / 24500) loss: 1.822847
(Iteration 831 / 24500) loss: 1.952278
(Iteration 841 / 24500) loss: 1.889701
(Iteration 851 / 24500) loss: 1.962453
(Iteration 861 / 24500) loss: 1.937287
(Iteration 871 / 24500) loss: 1.771308
(Iteration 881 / 24500) loss: 1.842643
(Iteration 891 / 24500) loss: 1.755551
(Iteration 901 / 24500) loss: 1.779141
(Iteration 911 / 24500) loss: 1.780095
(Iteration 921 / 24500) loss: 1.858769
(Iteration 931 / 24500) loss: 1.779578
(Iteration 941 / 24500) loss: 1.870665
(Iteration 951 / 24500) loss: 1.847691
(Iteration 961 / 24500) loss: 1.790168
(Iteration 971 / 24500) loss: 1.811250
```

```
(Epoch 2 / 50) train acc: 0.379000; val_acc: 0.375000
(Iteration 981 / 24500) loss: 1.786581
(Iteration 991 / 24500) loss: 1.816201
(Iteration 1001 / 24500) loss: 1.781015
(Iteration 1011 / 24500) loss: 1.737635
(Iteration 1021 / 24500) loss: 1.819882
(Iteration 1031 / 24500) loss: 1.741388
(Iteration 1041 / 24500) loss: 1.761660
(Iteration 1051 / 24500) loss: 1.688837
(Iteration 1061 / 24500) loss: 1.783823
(Iteration 1071 / 24500) loss: 1.698330
(Iteration 1081 / 24500) loss: 1.646667
(Iteration 1091 / 24500) loss: 1.714423
(Iteration 1101 / 24500) loss: 1.759590
(Iteration 1111 / 24500) loss: 1.681453
(Iteration 1121 / 24500) loss: 1.738066
(Iteration 1131 / 24500) loss: 1.712734
(Iteration 1141 / 24500) loss: 1.675394
(Iteration 1151 / 24500) loss: 1.531954
(Iteration 1161 / 24500) loss: 1.678433
(Iteration 1171 / 24500) loss: 1.669724
(Iteration 1181 / 24500) loss: 1.643295
(Iteration 1191 / 24500) loss: 1.699387
(Iteration 1201 / 24500) loss: 1.614935
(Iteration 1211 / 24500) loss: 1.629879
(Iteration 1221 / 24500) loss: 1.612767
(Iteration 1231 / 24500) loss: 1.538906
(Iteration 1241 / 24500) loss: 1.511275
(Iteration 1251 / 24500) loss: 1.642798
(Iteration 1261 / 24500) loss: 1.663818
(Iteration 1271 / 24500) loss: 1.616279
(Iteration 1281 / 24500) loss: 1.605929
(Iteration 1291 / 24500) loss: 1.595990
(Iteration 1301 / 24500) loss: 1.581132
(Iteration 1311 / 24500) loss: 1.755497
(Iteration 1321 / 24500) loss: 1.563735
(Iteration 1331 / 24500) loss: 1.539722
(Iteration 1341 / 24500) loss: 1.641633
(Iteration 1351 / 24500) loss: 1.478988
(Iteration 1361 / 24500) loss: 1.469742
(Iteration 1371 / 24500) loss: 1.590539
(Iteration 1381 / 24500) loss: 1.534051
(Iteration 1391 / 24500) loss: 1.539986
(Iteration 1401 / 24500) loss: 1.553734
(Iteration 1411 / 24500) loss: 1.657806
(Iteration 1421 / 24500) loss: 1.579651
(Iteration 1431 / 24500) loss: 1.713156
(Iteration 1441 / 24500) loss: 1.580469
```

```
(Iteration 1451 / 24500) loss: 1.423282
(Iteration 1461 / 24500) loss: 1.476763
(Epoch 3 / 50) train acc: 0.452000; val_acc: 0.445000
(Iteration 1471 / 24500) loss: 1.465648
(Iteration 1481 / 24500) loss: 1.502955
(Iteration 1491 / 24500) loss: 1.625352
(Iteration 1501 / 24500) loss: 1.490925
(Iteration 1511 / 24500) loss: 1.421751
(Iteration 1521 / 24500) loss: 1.429921
(Iteration 1531 / 24500) loss: 1.683475
(Iteration 1541 / 24500) loss: 1.512301
(Iteration 1551 / 24500) loss: 1.505129
(Iteration 1561 / 24500) loss: 1.504675
(Iteration 1571 / 24500) loss: 1.540149
(Iteration 1581 / 24500) loss: 1.486537
(Iteration 1591 / 24500) loss: 1.438338
(Iteration 1601 / 24500) loss: 1.396622
(Iteration 1611 / 24500) loss: 1.535810
(Iteration 1621 / 24500) loss: 1.546744
(Iteration 1631 / 24500) loss: 1.433217
(Iteration 1641 / 24500) loss: 1.516864
(Iteration 1651 / 24500) loss: 1.569798
(Iteration 1661 / 24500) loss: 1.611651
(Iteration 1671 / 24500) loss: 1.435387
(Iteration 1681 / 24500) loss: 1.431372
(Iteration 1691 / 24500) loss: 1.437079
(Iteration 1701 / 24500) loss: 1.460492
(Iteration 1711 / 24500) loss: 1.504821
(Iteration 1721 / 24500) loss: 1.426068
(Iteration 1731 / 24500) loss: 1.545976
(Iteration 1741 / 24500) loss: 1.628963
(Iteration 1751 / 24500) loss: 1.468988
(Iteration 1761 / 24500) loss: 1.431781
(Iteration 1771 / 24500) loss: 1.468292
(Iteration 1781 / 24500) loss: 1.430321
(Iteration 1791 / 24500) loss: 1.494039
(Iteration 1801 / 24500) loss: 1.450676
(Iteration 1811 / 24500) loss: 1.312969
(Iteration 1821 / 24500) loss: 1.517365
(Iteration 1831 / 24500) loss: 1.402519
(Iteration 1841 / 24500) loss: 1.611018
(Iteration 1851 / 24500) loss: 1.441382
(Iteration 1861 / 24500) loss: 1.522943
(Iteration 1871 / 24500) loss: 1.449617
(Iteration 1881 / 24500) loss: 1.358963
(Iteration 1891 / 24500) loss: 1.468469
(Iteration 1901 / 24500) loss: 1.316087
(Iteration 1911 / 24500) loss: 1.453193
```

```
(Iteration 1921 / 24500) loss: 1.438944
(Iteration 1931 / 24500) loss: 1.359321
(Iteration 1941 / 24500) loss: 1.417582
(Iteration 1951 / 24500) loss: 1.367675
(Epoch 4 / 50) train acc: 0.476000; val_acc: 0.500000
(Iteration 1961 / 24500) loss: 1.561250
(Iteration 1971 / 24500) loss: 1.478415
(Iteration 1981 / 24500) loss: 1.418228
(Iteration 1991 / 24500) loss: 1.273572
(Iteration 2001 / 24500) loss: 1.532415
(Iteration 2011 / 24500) loss: 1.376712
(Iteration 2021 / 24500) loss: 1.409755
(Iteration 2031 / 24500) loss: 1.286234
(Iteration 2041 / 24500) loss: 1.418371
(Iteration 2051 / 24500) loss: 1.261373
(Iteration 2061 / 24500) loss: 1.326224
(Iteration 2071 / 24500) loss: 1.282574
(Iteration 2081 / 24500) loss: 1.443835
(Iteration 2091 / 24500) loss: 1.525618
(Iteration 2101 / 24500) loss: 1.580939
(Iteration 2111 / 24500) loss: 1.355257
(Iteration 2121 / 24500) loss: 1.623664
(Iteration 2131 / 24500) loss: 1.305414
(Iteration 2141 / 24500) loss: 1.310742
(Iteration 2151 / 24500) loss: 1.548989
(Iteration 2161 / 24500) loss: 1.346409
(Iteration 2171 / 24500) loss: 1.414795
(Iteration 2181 / 24500) loss: 1.409468
(Iteration 2191 / 24500) loss: 1.293907
(Iteration 2201 / 24500) loss: 1.364523
(Iteration 2211 / 24500) loss: 1.514806
(Iteration 2221 / 24500) loss: 1.402532
(Iteration 2231 / 24500) loss: 1.436832
(Iteration 2241 / 24500) loss: 1.423500
(Iteration 2251 / 24500) loss: 1.295090
(Iteration 2261 / 24500) loss: 1.394663
(Iteration 2271 / 24500) loss: 1.295266
(Iteration 2281 / 24500) loss: 1.295349
(Iteration 2291 / 24500) loss: 1.325516
(Iteration 2301 / 24500) loss: 1.400082
(Iteration 2311 / 24500) loss: 1.471186
(Iteration 2321 / 24500) loss: 1.402971
(Iteration 2331 / 24500) loss: 1.579110
(Iteration 2341 / 24500) loss: 1.417308
(Iteration 2351 / 24500) loss: 1.451851
(Iteration 2361 / 24500) loss: 1.453922
(Iteration 2371 / 24500) loss: 1.278146
(Iteration 2381 / 24500) loss: 1.357872
```

```
(Iteration 2391 / 24500) loss: 1.407853
(Iteration 2401 / 24500) loss: 1.253326
(Iteration 2411 / 24500) loss: 1.457335
(Iteration 2421 / 24500) loss: 1.462924
(Iteration 2431 / 24500) loss: 1.479269
(Iteration 2441 / 24500) loss: 1.348789
(Epoch 5 / 50) train acc: 0.505000; val_acc: 0.512000
(Iteration 2451 / 24500) loss: 1.302519
(Iteration 2461 / 24500) loss: 1.380519
(Iteration 2471 / 24500) loss: 1.243349
(Iteration 2481 / 24500) loss: 1.407998
(Iteration 2491 / 24500) loss: 1.348712
(Iteration 2501 / 24500) loss: 1.166153
(Iteration 2511 / 24500) loss: 1.563776
(Iteration 2521 / 24500) loss: 1.362340
(Iteration 2531 / 24500) loss: 1.169346
(Iteration 2541 / 24500) loss: 1.322971
(Iteration 2551 / 24500) loss: 1.432279
(Iteration 2561 / 24500) loss: 1.429568
(Iteration 2571 / 24500) loss: 1.451403
(Iteration 2581 / 24500) loss: 1.476922
(Iteration 2591 / 24500) loss: 1.385101
(Iteration 2601 / 24500) loss: 1.217669
(Iteration 2611 / 24500) loss: 1.316557
(Iteration 2621 / 24500) loss: 1.395505
(Iteration 2631 / 24500) loss: 1.334640
(Iteration 2641 / 24500) loss: 1.463865
(Iteration 2651 / 24500) loss: 1.391532
(Iteration 2661 / 24500) loss: 1.292870
(Iteration 2671 / 24500) loss: 1.350200
(Iteration 2681 / 24500) loss: 1.404086
(Iteration 2691 / 24500) loss: 1.358176
(Iteration 2701 / 24500) loss: 1.354893
(Iteration 2711 / 24500) loss: 1.381762
(Iteration 2721 / 24500) loss: 1.345728
(Iteration 2731 / 24500) loss: 1.256577
(Iteration 2741 / 24500) loss: 1.364370
(Iteration 2751 / 24500) loss: 1.209419
(Iteration 2761 / 24500) loss: 1.255539
(Iteration 2771 / 24500) loss: 1.263497
(Iteration 2781 / 24500) loss: 1.414784
(Iteration 2791 / 24500) loss: 1.447670
(Iteration 2801 / 24500) loss: 1.575290
(Iteration 2811 / 24500) loss: 1.499700
(Iteration 2821 / 24500) loss: 1.449107
(Iteration 2831 / 24500) loss: 1.334338
(Iteration 2841 / 24500) loss: 1.351059
(Iteration 2851 / 24500) loss: 1.440507
```

```
(Iteration 2861 / 24500) loss: 1.318783
(Iteration 2871 / 24500) loss: 1.284957
(Iteration 2881 / 24500) loss: 1.362108
(Iteration 2891 / 24500) loss: 1.278083
(Iteration 2901 / 24500) loss: 1.279281
(Iteration 2911 / 24500) loss: 1.323608
(Iteration 2921 / 24500) loss: 1.528691
(Iteration 2931 / 24500) loss: 1.359670
(Epoch 6 / 50) train acc: 0.506000; val_acc: 0.521000
(Iteration 2941 / 24500) loss: 1.211630
(Iteration 2951 / 24500) loss: 1.379456
(Iteration 2961 / 24500) loss: 1.613239
(Iteration 2971 / 24500) loss: 1.252168
(Iteration 2981 / 24500) loss: 1.579050
(Iteration 2991 / 24500) loss: 1.483739
(Iteration 3001 / 24500) loss: 1.298863
(Iteration 3011 / 24500) loss: 1.363011
(Iteration 3021 / 24500) loss: 1.327714
(Iteration 3031 / 24500) loss: 1.492191
(Iteration 3041 / 24500) loss: 1.341170
(Iteration 3051 / 24500) loss: 1.427379
(Iteration 3061 / 24500) loss: 1.372718
(Iteration 3071 / 24500) loss: 1.153576
(Iteration 3081 / 24500) loss: 1.419668
(Iteration 3091 / 24500) loss: 1.263781
(Iteration 3101 / 24500) loss: 1.309845
(Iteration 3111 / 24500) loss: 1.476814
(Iteration 3121 / 24500) loss: 1.468164
(Iteration 3131 / 24500) loss: 1.171304
(Iteration 3141 / 24500) loss: 1.384345
(Iteration 3151 / 24500) loss: 1.425350
(Iteration 3161 / 24500) loss: 1.295222
(Iteration 3171 / 24500) loss: 1.259225
(Iteration 3181 / 24500) loss: 1.109071
(Iteration 3191 / 24500) loss: 1.449816
(Iteration 3201 / 24500) loss: 1.266717
(Iteration 3211 / 24500) loss: 1.313782
(Iteration 3221 / 24500) loss: 1.479327
(Iteration 3231 / 24500) loss: 1.182112
(Iteration 3241 / 24500) loss: 1.236689
(Iteration 3251 / 24500) loss: 1.240016
(Iteration 3261 / 24500) loss: 1.301121
(Iteration 3271 / 24500) loss: 1.220153
(Iteration 3281 / 24500) loss: 1.194931
(Iteration 3291 / 24500) loss: 1.377527
(Iteration 3301 / 24500) loss: 1.415184
(Iteration 3311 / 24500) loss: 1.244978
(Iteration 3321 / 24500) loss: 1.290575
```

```
(Iteration 3331 / 24500) loss: 1.340030
(Iteration 3341 / 24500) loss: 1.277303
(Iteration 3351 / 24500) loss: 1.541448
(Iteration 3361 / 24500) loss: 1.294072
(Iteration 3371 / 24500) loss: 1.415452
(Iteration 3381 / 24500) loss: 1.301406
(Iteration 3391 / 24500) loss: 1.336171
(Iteration 3401 / 24500) loss: 1.423821
(Iteration 3411 / 24500) loss: 1.191727
(Iteration 3421 / 24500) loss: 1.240182
(Epoch 7 / 50) train acc: 0.538000; val_acc: 0.519000
(Iteration 3431 / 24500) loss: 1.268640
(Iteration 3441 / 24500) loss: 1.423063
(Iteration 3451 / 24500) loss: 1.421636
(Iteration 3461 / 24500) loss: 1.350509
(Iteration 3471 / 24500) loss: 1.397988
(Iteration 3481 / 24500) loss: 1.463517
(Iteration 3491 / 24500) loss: 1.274527
(Iteration 3501 / 24500) loss: 1.467017
(Iteration 3511 / 24500) loss: 1.441833
(Iteration 3521 / 24500) loss: 1.326188
(Iteration 3531 / 24500) loss: 1.350755
(Iteration 3541 / 24500) loss: 1.286617
(Iteration 3551 / 24500) loss: 1.456514
(Iteration 3561 / 24500) loss: 1.409672
(Iteration 3571 / 24500) loss: 1.226731
(Iteration 3581 / 24500) loss: 1.280058
(Iteration 3591 / 24500) loss: 1.302843
(Iteration 3601 / 24500) loss: 1.240013
(Iteration 3611 / 24500) loss: 1.168886
(Iteration 3621 / 24500) loss: 1.494483
(Iteration 3631 / 24500) loss: 1.424915
(Iteration 3641 / 24500) loss: 1.296703
(Iteration 3651 / 24500) loss: 1.491864
(Iteration 3661 / 24500) loss: 1.421869
(Iteration 3671 / 24500) loss: 1.132977
(Iteration 3681 / 24500) loss: 1.287060
(Iteration 3691 / 24500) loss: 1.382087
(Iteration 3701 / 24500) loss: 1.506597
(Iteration 3711 / 24500) loss: 1.313479
(Iteration 3721 / 24500) loss: 1.435624
(Iteration 3731 / 24500) loss: 1.345228
(Iteration 3741 / 24500) loss: 1.179473
(Iteration 3751 / 24500) loss: 1.356293
(Iteration 3761 / 24500) loss: 1.245988
(Iteration 3771 / 24500) loss: 1.373442
(Iteration 3781 / 24500) loss: 1.291599
(Iteration 3791 / 24500) loss: 1.379420
```

```
(Iteration 3801 / 24500) loss: 1.257515
(Iteration 3811 / 24500) loss: 1.353964
(Iteration 3821 / 24500) loss: 1.400430
(Iteration 3831 / 24500) loss: 1.530864
(Iteration 3841 / 24500) loss: 1.348398
(Iteration 3851 / 24500) loss: 1.362728
(Iteration 3861 / 24500) loss: 1.340400
(Iteration 3871 / 24500) loss: 1.238595
(Iteration 3881 / 24500) loss: 1.458176
(Iteration 3891 / 24500) loss: 1.503534
(Iteration 3901 / 24500) loss: 1.319830
(Iteration 3911 / 24500) loss: 1.325441
(Epoch 8 / 50) train acc: 0.534000; val_acc: 0.520000
(Iteration 3921 / 24500) loss: 1.243138
(Iteration 3931 / 24500) loss: 1.324324
(Iteration 3941 / 24500) loss: 1.369678
(Iteration 3951 / 24500) loss: 1.368750
(Iteration 3961 / 24500) loss: 1.273448
(Iteration 3971 / 24500) loss: 1.367309
(Iteration 3981 / 24500) loss: 1.319580
(Iteration 3991 / 24500) loss: 1.415199
(Iteration 4001 / 24500) loss: 1.510965
(Iteration 4011 / 24500) loss: 1.423411
(Iteration 4021 / 24500) loss: 1.333833
(Iteration 4031 / 24500) loss: 1.393954
(Iteration 4041 / 24500) loss: 1.160897
(Iteration 4051 / 24500) loss: 1.353122
(Iteration 4061 / 24500) loss: 1.421269
(Iteration 4071 / 24500) loss: 1.395873
(Iteration 4081 / 24500) loss: 1.318308
(Iteration 4091 / 24500) loss: 1.230474
(Iteration 4101 / 24500) loss: 1.333943
(Iteration 4111 / 24500) loss: 1.233934
(Iteration 4121 / 24500) loss: 1.355713
(Iteration 4131 / 24500) loss: 1.184627
(Iteration 4141 / 24500) loss: 1.329580
(Iteration 4151 / 24500) loss: 1.445894
(Iteration 4161 / 24500) loss: 1.410661
(Iteration 4171 / 24500) loss: 1.382078
(Iteration 4181 / 24500) loss: 1.398644
(Iteration 4191 / 24500) loss: 1.432106
(Iteration 4201 / 24500) loss: 1.306450
(Iteration 4211 / 24500) loss: 1.473213
(Iteration 4221 / 24500) loss: 1.374119
(Iteration 4231 / 24500) loss: 1.283111
(Iteration 4241 / 24500) loss: 1.153704
(Iteration 4251 / 24500) loss: 1.254038
(Iteration 4261 / 24500) loss: 1.294583
```

```
(Iteration 4271 / 24500) loss: 1.077723
(Iteration 4281 / 24500) loss: 1.416300
(Iteration 4291 / 24500) loss: 1.337220
(Iteration 4301 / 24500) loss: 1.281679
(Iteration 4311 / 24500) loss: 1.275989
(Iteration 4321 / 24500) loss: 1.290384
(Iteration 4331 / 24500) loss: 1.314640
(Iteration 4341 / 24500) loss: 1.143615
(Iteration 4351 / 24500) loss: 1.321180
(Iteration 4361 / 24500) loss: 1.090851
(Iteration 4371 / 24500) loss: 1.349897
(Iteration 4381 / 24500) loss: 1.394416
(Iteration 4391 / 24500) loss: 1.167327
(Iteration 4401 / 24500) loss: 1.187544
(Epoch 9 / 50) train acc: 0.521000; val_acc: 0.520000
(Iteration 4411 / 24500) loss: 1.161424
(Iteration 4421 / 24500) loss: 1.435143
(Iteration 4431 / 24500) loss: 1.163004
(Iteration 4441 / 24500) loss: 1.356743
(Iteration 4451 / 24500) loss: 1.200494
(Iteration 4461 / 24500) loss: 1.278940
(Iteration 4471 / 24500) loss: 1.468516
(Iteration 4481 / 24500) loss: 1.232406
(Iteration 4491 / 24500) loss: 1.188523
(Iteration 4501 / 24500) loss: 1.309011
(Iteration 4511 / 24500) loss: 1.097715
(Iteration 4521 / 24500) loss: 1.341436
(Iteration 4531 / 24500) loss: 1.368153
(Iteration 4541 / 24500) loss: 1.252825
(Iteration 4551 / 24500) loss: 1.332657
(Iteration 4561 / 24500) loss: 1.323950
(Iteration 4571 / 24500) loss: 1.256643
(Iteration 4581 / 24500) loss: 1.334688
(Iteration 4591 / 24500) loss: 1.262005
(Iteration 4601 / 24500) loss: 1.201601
(Iteration 4611 / 24500) loss: 1.387020
(Iteration 4621 / 24500) loss: 1.355351
(Iteration 4631 / 24500) loss: 1.239995
(Iteration 4641 / 24500) loss: 1.256297
(Iteration 4651 / 24500) loss: 1.400653
(Iteration 4661 / 24500) loss: 1.220255
(Iteration 4671 / 24500) loss: 1.247474
(Iteration 4681 / 24500) loss: 1.441807
(Iteration 4691 / 24500) loss: 1.453459
(Iteration 4701 / 24500) loss: 1.387460
(Iteration 4711 / 24500) loss: 1.292499
(Iteration 4721 / 24500) loss: 1.296382
(Iteration 4731 / 24500) loss: 1.440236
```

```
(Iteration 4741 / 24500) loss: 1.296600
(Iteration 4751 / 24500) loss: 1.175788
(Iteration 4761 / 24500) loss: 1.397826
(Iteration 4771 / 24500) loss: 1.374583
(Iteration 4781 / 24500) loss: 1.309629
(Iteration 4791 / 24500) loss: 1.311381
(Iteration 4801 / 24500) loss: 1.434809
(Iteration 4811 / 24500) loss: 1.228235
(Iteration 4821 / 24500) loss: 1.342652
(Iteration 4831 / 24500) loss: 1.187342
(Iteration 4841 / 24500) loss: 1.420083
(Iteration 4851 / 24500) loss: 1.198676
(Iteration 4861 / 24500) loss: 1.245279
(Iteration 4871 / 24500) loss: 1.149165
(Iteration 4881 / 24500) loss: 1.405552
(Iteration 4891 / 24500) loss: 1.397723
(Epoch 10 / 50) train acc: 0.547000; val_acc: 0.514000
(Iteration 4901 / 24500) loss: 1.484641
(Iteration 4911 / 24500) loss: 1.352152
(Iteration 4921 / 24500) loss: 1.362698
(Iteration 4931 / 24500) loss: 1.199467
(Iteration 4941 / 24500) loss: 1.137962
(Iteration 4951 / 24500) loss: 1.400506
(Iteration 4961 / 24500) loss: 1.351834
(Iteration 4971 / 24500) loss: 1.387226
(Iteration 4981 / 24500) loss: 1.327887
(Iteration 4991 / 24500) loss: 1.177634
(Iteration 5001 / 24500) loss: 1.404000
(Iteration 5011 / 24500) loss: 1.206869
(Iteration 5021 / 24500) loss: 1.305474
(Iteration 5031 / 24500) loss: 1.535129
(Iteration 5041 / 24500) loss: 1.279905
(Iteration 5051 / 24500) loss: 1.376574
(Iteration 5061 / 24500) loss: 1.134086
(Iteration 5071 / 24500) loss: 1.171772
(Iteration 5081 / 24500) loss: 1.289519
(Iteration 5091 / 24500) loss: 1.325767
(Iteration 5101 / 24500) loss: 1.434048
(Iteration 5111 / 24500) loss: 1.263917
(Iteration 5121 / 24500) loss: 1.383477
(Iteration 5131 / 24500) loss: 1.214552
(Iteration 5141 / 24500) loss: 1.219419
(Iteration 5151 / 24500) loss: 1.348255
(Iteration 5161 / 24500) loss: 1.072105
(Iteration 5171 / 24500) loss: 1.251518
(Iteration 5181 / 24500) loss: 1.293370
(Iteration 5191 / 24500) loss: 1.358269
(Iteration 5201 / 24500) loss: 1.330573
```

```
(Iteration 5211 / 24500) loss: 1.206732
(Iteration 5221 / 24500) loss: 1.482023
(Iteration 5231 / 24500) loss: 1.312876
(Iteration 5241 / 24500) loss: 1.316728
(Iteration 5251 / 24500) loss: 1.316371
(Iteration 5261 / 24500) loss: 1.359410
(Iteration 5271 / 24500) loss: 1.413684
(Iteration 5281 / 24500) loss: 1.336335
(Iteration 5291 / 24500) loss: 1.508518
(Iteration 5301 / 24500) loss: 1.224999
(Iteration 5311 / 24500) loss: 1.387514
(Iteration 5321 / 24500) loss: 0.883756
(Iteration 5331 / 24500) loss: 1.302520
(Iteration 5341 / 24500) loss: 1.352493
(Iteration 5351 / 24500) loss: 1.368616
(Iteration 5361 / 24500) loss: 1.266351
(Iteration 5371 / 24500) loss: 1.220011
(Iteration 5381 / 24500) loss: 1.114521
(Epoch 11 / 50) train acc: 0.549000; val_acc: 0.527000
(Iteration 5391 / 24500) loss: 1.399125
(Iteration 5401 / 24500) loss: 1.359601
(Iteration 5411 / 24500) loss: 1.328946
(Iteration 5421 / 24500) loss: 1.327863
(Iteration 5431 / 24500) loss: 1.232950
(Iteration 5441 / 24500) loss: 1.254877
(Iteration 5451 / 24500) loss: 1.248222
(Iteration 5461 / 24500) loss: 1.156473
(Iteration 5471 / 24500) loss: 1.281552
(Iteration 5481 / 24500) loss: 1.507108
(Iteration 5491 / 24500) loss: 1.130814
(Iteration 5501 / 24500) loss: 1.238039
(Iteration 5511 / 24500) loss: 1.292110
(Iteration 5521 / 24500) loss: 1.146327
(Iteration 5531 / 24500) loss: 1.198139
(Iteration 5541 / 24500) loss: 1.310000
(Iteration 5551 / 24500) loss: 0.997983
(Iteration 5561 / 24500) loss: 1.301326
(Iteration 5571 / 24500) loss: 1.401935
(Iteration 5581 / 24500) loss: 1.206938
(Iteration 5591 / 24500) loss: 1.472610
(Iteration 5601 / 24500) loss: 1.249858
(Iteration 5611 / 24500) loss: 1.119685
(Iteration 5621 / 24500) loss: 1.372618
(Iteration 5631 / 24500) loss: 1.296282
(Iteration 5641 / 24500) loss: 1.329860
(Iteration 5651 / 24500) loss: 1.163552
(Iteration 5661 / 24500) loss: 1.216127
(Iteration 5671 / 24500) loss: 1.205339
```

```
(Iteration 5681 / 24500) loss: 1.125727
(Iteration 5691 / 24500) loss: 1.250499
(Iteration 5701 / 24500) loss: 1.192480
(Iteration 5711 / 24500) loss: 1.324391
(Iteration 5721 / 24500) loss: 1.359436
(Iteration 5731 / 24500) loss: 1.296691
(Iteration 5741 / 24500) loss: 1.209366
(Iteration 5751 / 24500) loss: 1.472262
(Iteration 5761 / 24500) loss: 1.264795
(Iteration 5771 / 24500) loss: 1.128148
(Iteration 5781 / 24500) loss: 1.330855
(Iteration 5791 / 24500) loss: 1.132033
(Iteration 5801 / 24500) loss: 1.319649
(Iteration 5811 / 24500) loss: 1.208920
(Iteration 5821 / 24500) loss: 1.120803
(Iteration 5831 / 24500) loss: 1.282242
(Iteration 5841 / 24500) loss: 1.338260
(Iteration 5851 / 24500) loss: 1.160701
(Iteration 5861 / 24500) loss: 1.307504
(Iteration 5871 / 24500) loss: 1.194248
(Epoch 12 / 50) train acc: 0.549000; val_acc: 0.530000
(Iteration 5881 / 24500) loss: 1.281432
(Iteration 5891 / 24500) loss: 1.418482
(Iteration 5901 / 24500) loss: 1.370085
(Iteration 5911 / 24500) loss: 1.302874
(Iteration 5921 / 24500) loss: 1.297522
(Iteration 5931 / 24500) loss: 1.097211
(Iteration 5941 / 24500) loss: 1.198985
(Iteration 5951 / 24500) loss: 1.249418
(Iteration 5961 / 24500) loss: 1.256564
(Iteration 5971 / 24500) loss: 1.148431
(Iteration 5981 / 24500) loss: 1.247281
(Iteration 5991 / 24500) loss: 1.243353
(Iteration 6001 / 24500) loss: 1.201148
(Iteration 6011 / 24500) loss: 1.281173
(Iteration 6021 / 24500) loss: 1.256200
(Iteration 6031 / 24500) loss: 1.238422
(Iteration 6041 / 24500) loss: 1.520925
(Iteration 6051 / 24500) loss: 1.269045
(Iteration 6061 / 24500) loss: 1.241954
(Iteration 6071 / 24500) loss: 1.133505
(Iteration 6081 / 24500) loss: 0.977730
(Iteration 6091 / 24500) loss: 1.131294
(Iteration 6101 / 24500) loss: 1.269864
(Iteration 6111 / 24500) loss: 1.251720
(Iteration 6121 / 24500) loss: 1.161914
(Iteration 6131 / 24500) loss: 1.480361
(Iteration 6141 / 24500) loss: 1.642124
```

```
(Iteration 6151 / 24500) loss: 1.268089
(Iteration 6161 / 24500) loss: 1.294489
(Iteration 6171 / 24500) loss: 1.404356
(Iteration 6181 / 24500) loss: 1.366811
(Iteration 6191 / 24500) loss: 1.480121
(Iteration 6201 / 24500) loss: 1.317667
(Iteration 6211 / 24500) loss: 1.268910
(Iteration 6221 / 24500) loss: 1.291413
(Iteration 6231 / 24500) loss: 1.293212
(Iteration 6241 / 24500) loss: 1.131944
(Iteration 6251 / 24500) loss: 1.282286
(Iteration 6261 / 24500) loss: 1.217794
(Iteration 6271 / 24500) loss: 1.223823
(Iteration 6281 / 24500) loss: 1.375132
(Iteration 6291 / 24500) loss: 1.518629
(Iteration 6301 / 24500) loss: 1.271658
(Iteration 6311 / 24500) loss: 1.137832
(Iteration 6321 / 24500) loss: 1.289488
(Iteration 6331 / 24500) loss: 1.167798
(Iteration 6341 / 24500) loss: 1.110887
(Iteration 6351 / 24500) loss: 1.255872
(Iteration 6361 / 24500) loss: 1.472809
(Epoch 13 / 50) train acc: 0.563000; val_acc: 0.538000
(Iteration 6371 / 24500) loss: 1.151185
(Iteration 6381 / 24500) loss: 1.332883
(Iteration 6391 / 24500) loss: 1.330872
(Iteration 6401 / 24500) loss: 1.270206
(Iteration 6411 / 24500) loss: 1.291334
(Iteration 6421 / 24500) loss: 1.307523
(Iteration 6431 / 24500) loss: 1.241375
(Iteration 6441 / 24500) loss: 1.245626
(Iteration 6451 / 24500) loss: 1.162393
(Iteration 6461 / 24500) loss: 1.174719
(Iteration 6471 / 24500) loss: 0.984483
(Iteration 6481 / 24500) loss: 1.330748
(Iteration 6491 / 24500) loss: 1.283838
(Iteration 6501 / 24500) loss: 1.131431
(Iteration 6511 / 24500) loss: 1.345309
(Iteration 6521 / 24500) loss: 1.104786
(Iteration 6531 / 24500) loss: 1.341903
(Iteration 6541 / 24500) loss: 1.420841
(Iteration 6551 / 24500) loss: 1.362392
(Iteration 6561 / 24500) loss: 1.331997
(Iteration 6571 / 24500) loss: 1.232477
(Iteration 6581 / 24500) loss: 1.223144
(Iteration 6591 / 24500) loss: 1.016487
(Iteration 6601 / 24500) loss: 1.100146
(Iteration 6611 / 24500) loss: 1.190672
```

```
(Iteration 6621 / 24500) loss: 1.175669
(Iteration 6631 / 24500) loss: 1.112662
(Iteration 6641 / 24500) loss: 1.372216
(Iteration 6651 / 24500) loss: 1.324095
(Iteration 6661 / 24500) loss: 1.167341
(Iteration 6671 / 24500) loss: 1.521332
(Iteration 6681 / 24500) loss: 1.112338
(Iteration 6691 / 24500) loss: 1.152304
(Iteration 6701 / 24500) loss: 1.179591
(Iteration 6711 / 24500) loss: 1.241500
(Iteration 6721 / 24500) loss: 1.237491
(Iteration 6731 / 24500) loss: 1.087366
(Iteration 6741 / 24500) loss: 1.234387
(Iteration 6751 / 24500) loss: 1.130523
(Iteration 6761 / 24500) loss: 1.337700
(Iteration 6771 / 24500) loss: 1.130654
(Iteration 6781 / 24500) loss: 1.181564
(Iteration 6791 / 24500) loss: 1.314923
(Iteration 6801 / 24500) loss: 1.085847
(Iteration 6811 / 24500) loss: 1.330575
(Iteration 6821 / 24500) loss: 1.407656
(Iteration 6831 / 24500) loss: 1.384983
(Iteration 6841 / 24500) loss: 1.372322
(Iteration 6851 / 24500) loss: 1.062486
(Epoch 14 / 50) train acc: 0.547000; val_acc: 0.541000
(Iteration 6861 / 24500) loss: 1.391562
(Iteration 6871 / 24500) loss: 1.239223
(Iteration 6881 / 24500) loss: 1.151580
(Iteration 6891 / 24500) loss: 1.112986
(Iteration 6901 / 24500) loss: 1.047107
(Iteration 6911 / 24500) loss: 1.209231
(Iteration 6921 / 24500) loss: 1.091698
(Iteration 6931 / 24500) loss: 1.244040
(Iteration 6941 / 24500) loss: 1.303961
(Iteration 6951 / 24500) loss: 1.190638
(Iteration 6961 / 24500) loss: 1.254126
(Iteration 6971 / 24500) loss: 1.221525
(Iteration 6981 / 24500) loss: 1.250331
(Iteration 6991 / 24500) loss: 1.569548
(Iteration 7001 / 24500) loss: 1.410259
(Iteration 7011 / 24500) loss: 1.031250
(Iteration 7021 / 24500) loss: 1.341186
(Iteration 7031 / 24500) loss: 1.321729
(Iteration 7041 / 24500) loss: 1.230352
(Iteration 7051 / 24500) loss: 1.281873
(Iteration 7061 / 24500) loss: 1.304038
(Iteration 7071 / 24500) loss: 1.525555
(Iteration 7081 / 24500) loss: 1.190234
```

```
(Iteration 7091 / 24500) loss: 1.419943
(Iteration 7101 / 24500) loss: 1.392862
(Iteration 7111 / 24500) loss: 1.226931
(Iteration 7121 / 24500) loss: 1.206697
(Iteration 7131 / 24500) loss: 1.172725
(Iteration 7141 / 24500) loss: 1.136548
(Iteration 7151 / 24500) loss: 1.426128
(Iteration 7161 / 24500) loss: 1.189614
(Iteration 7171 / 24500) loss: 1.145303
(Iteration 7181 / 24500) loss: 1.264986
(Iteration 7191 / 24500) loss: 1.329794
(Iteration 7201 / 24500) loss: 1.170749
(Iteration 7211 / 24500) loss: 1.164766
(Iteration 7221 / 24500) loss: 1.371385
(Iteration 7231 / 24500) loss: 1.296630
(Iteration 7241 / 24500) loss: 1.203082
(Iteration 7251 / 24500) loss: 1.225814
(Iteration 7261 / 24500) loss: 1.168743
(Iteration 7271 / 24500) loss: 1.026012
(Iteration 7281 / 24500) loss: 1.163427
(Iteration 7291 / 24500) loss: 1.269631
(Iteration 7301 / 24500) loss: 1.308140
(Iteration 7311 / 24500) loss: 1.496442
(Iteration 7321 / 24500) loss: 1.120005
(Iteration 7331 / 24500) loss: 1.227927
(Iteration 7341 / 24500) loss: 1.212741
(Epoch 15 / 50) train acc: 0.565000; val_acc: 0.549000
(Iteration 7351 / 24500) loss: 1.017502
(Iteration 7361 / 24500) loss: 1.368954
(Iteration 7371 / 24500) loss: 1.179703
(Iteration 7381 / 24500) loss: 1.201997
(Iteration 7391 / 24500) loss: 1.413831
(Iteration 7401 / 24500) loss: 1.146316
(Iteration 7411 / 24500) loss: 1.372998
(Iteration 7421 / 24500) loss: 1.232551
(Iteration 7431 / 24500) loss: 1.348507
(Iteration 7441 / 24500) loss: 1.338755
(Iteration 7451 / 24500) loss: 1.227307
(Iteration 7461 / 24500) loss: 1.053580
(Iteration 7471 / 24500) loss: 1.357053
(Iteration 7481 / 24500) loss: 1.308807
(Iteration 7491 / 24500) loss: 1.215677
(Iteration 7501 / 24500) loss: 1.344949
(Iteration 7511 / 24500) loss: 1.251540
(Iteration 7521 / 24500) loss: 1.213776
(Iteration 7531 / 24500) loss: 1.293007
(Iteration 7541 / 24500) loss: 1.334070
(Iteration 7551 / 24500) loss: 1.314196
```

```
(Iteration 7561 / 24500) loss: 1.357098
(Iteration 7571 / 24500) loss: 1.183307
(Iteration 7581 / 24500) loss: 1.238728
(Iteration 7591 / 24500) loss: 1.274905
(Iteration 7601 / 24500) loss: 1.169574
(Iteration 7611 / 24500) loss: 1.370798
(Iteration 7621 / 24500) loss: 1.264612
(Iteration 7631 / 24500) loss: 1.212822
(Iteration 7641 / 24500) loss: 0.994269
(Iteration 7651 / 24500) loss: 1.349139
(Iteration 7661 / 24500) loss: 1.282314
(Iteration 7671 / 24500) loss: 1.209198
(Iteration 7681 / 24500) loss: 1.150922
(Iteration 7691 / 24500) loss: 1.152910
(Iteration 7701 / 24500) loss: 1.201451
(Iteration 7711 / 24500) loss: 1.211319
(Iteration 7721 / 24500) loss: 1.346475
(Iteration 7731 / 24500) loss: 1.169899
(Iteration 7741 / 24500) loss: 1.202584
(Iteration 7751 / 24500) loss: 1.159727
(Iteration 7761 / 24500) loss: 1.151822
(Iteration 7771 / 24500) loss: 1.294185
(Iteration 7781 / 24500) loss: 1.482542
(Iteration 7791 / 24500) loss: 1.071404
(Iteration 7801 / 24500) loss: 1.230192
(Iteration 7811 / 24500) loss: 1.024991
(Iteration 7821 / 24500) loss: 1.091659
(Iteration 7831 / 24500) loss: 1.172083
(Epoch 16 / 50) train acc: 0.587000; val_acc: 0.546000
(Iteration 7841 / 24500) loss: 1.299742
(Iteration 7851 / 24500) loss: 1.238080
(Iteration 7861 / 24500) loss: 1.176187
(Iteration 7871 / 24500) loss: 1.232046
(Iteration 7881 / 24500) loss: 1.198498
(Iteration 7891 / 24500) loss: 1.115355
(Iteration 7901 / 24500) loss: 1.357973
(Iteration 7911 / 24500) loss: 1.193196
(Iteration 7921 / 24500) loss: 1.125566
(Iteration 7931 / 24500) loss: 1.299646
(Iteration 7941 / 24500) loss: 1.326078
(Iteration 7951 / 24500) loss: 1.427811
(Iteration 7961 / 24500) loss: 1.151060
(Iteration 7971 / 24500) loss: 1.315069
(Iteration 7981 / 24500) loss: 1.274738
(Iteration 7991 / 24500) loss: 1.361875
(Iteration 8001 / 24500) loss: 1.320414
(Iteration 8011 / 24500) loss: 1.124213
(Iteration 8021 / 24500) loss: 1.097269
```

```
(Iteration 8031 / 24500) loss: 1.242194
(Iteration 8041 / 24500) loss: 1.109721
(Iteration 8051 / 24500) loss: 1.063820
(Iteration 8061 / 24500) loss: 1.197102
(Iteration 8071 / 24500) loss: 1.262058
(Iteration 8081 / 24500) loss: 1.258144
(Iteration 8091 / 24500) loss: 1.233401
(Iteration 8101 / 24500) loss: 1.193993
(Iteration 8111 / 24500) loss: 1.177467
(Iteration 8121 / 24500) loss: 1.019276
(Iteration 8131 / 24500) loss: 1.193246
(Iteration 8141 / 24500) loss: 1.101876
(Iteration 8151 / 24500) loss: 1.182630
(Iteration 8161 / 24500) loss: 0.980666
(Iteration 8171 / 24500) loss: 1.162785
(Iteration 8181 / 24500) loss: 1.464010
(Iteration 8191 / 24500) loss: 1.303194
(Iteration 8201 / 24500) loss: 1.201781
(Iteration 8211 / 24500) loss: 1.451871
(Iteration 8221 / 24500) loss: 1.340463
(Iteration 8231 / 24500) loss: 1.179517
(Iteration 8241 / 24500) loss: 1.228638
(Iteration 8251 / 24500) loss: 1.240807
(Iteration 8261 / 24500) loss: 1.123572
(Iteration 8271 / 24500) loss: 1.198087
(Iteration 8281 / 24500) loss: 1.202258
(Iteration 8291 / 24500) loss: 1.100683
(Iteration 8301 / 24500) loss: 1.048741
(Iteration 8311 / 24500) loss: 1.304859
(Iteration 8321 / 24500) loss: 1.146744
(Epoch 17 / 50) train acc: 0.578000; val_acc: 0.549000
(Iteration 8331 / 24500) loss: 1.110286
(Iteration 8341 / 24500) loss: 1.274507
(Iteration 8351 / 24500) loss: 1.276065
(Iteration 8361 / 24500) loss: 1.190882
(Iteration 8371 / 24500) loss: 1.078061
(Iteration 8381 / 24500) loss: 1.115342
(Iteration 8391 / 24500) loss: 1.128258
(Iteration 8401 / 24500) loss: 1.156009
(Iteration 8411 / 24500) loss: 1.308970
(Iteration 8421 / 24500) loss: 1.201576
(Iteration 8431 / 24500) loss: 1.196219
(Iteration 8441 / 24500) loss: 1.319181
(Iteration 8451 / 24500) loss: 1.211924
(Iteration 8461 / 24500) loss: 1.311818
(Iteration 8471 / 24500) loss: 1.484806
(Iteration 8481 / 24500) loss: 1.127229
(Iteration 8491 / 24500) loss: 1.037642
```

```
(Iteration 8501 / 24500) loss: 1.239886
(Iteration 8511 / 24500) loss: 1.239237
(Iteration 8521 / 24500) loss: 1.174525
(Iteration 8531 / 24500) loss: 1.243734
(Iteration 8541 / 24500) loss: 1.268705
(Iteration 8551 / 24500) loss: 1.297087
(Iteration 8561 / 24500) loss: 1.420521
(Iteration 8571 / 24500) loss: 1.311697
(Iteration 8581 / 24500) loss: 1.220795
(Iteration 8591 / 24500) loss: 1.184048
(Iteration 8601 / 24500) loss: 1.174698
(Iteration 8611 / 24500) loss: 1.377864
(Iteration 8621 / 24500) loss: 1.071378
(Iteration 8631 / 24500) loss: 1.162316
(Iteration 8641 / 24500) loss: 1.440557
(Iteration 8651 / 24500) loss: 1.119653
(Iteration 8661 / 24500) loss: 1.108160
(Iteration 8671 / 24500) loss: 1.163078
(Iteration 8681 / 24500) loss: 1.131773
(Iteration 8691 / 24500) loss: 1.000944
(Iteration 8701 / 24500) loss: 1.152373
(Iteration 8711 / 24500) loss: 1.423785
(Iteration 8721 / 24500) loss: 1.302612
(Iteration 8731 / 24500) loss: 1.132358
(Iteration 8741 / 24500) loss: 1.059124
(Iteration 8751 / 24500) loss: 1.170200
(Iteration 8761 / 24500) loss: 1.266332
(Iteration 8771 / 24500) loss: 1.357760
(Iteration 8781 / 24500) loss: 1.209455
(Iteration 8791 / 24500) loss: 1.169915
(Iteration 8801 / 24500) loss: 1.131804
(Iteration 8811 / 24500) loss: 1.252444
(Epoch 18 / 50) train acc: 0.554000; val_acc: 0.544000
(Iteration 8821 / 24500) loss: 1.111070
(Iteration 8831 / 24500) loss: 1.154561
(Iteration 8841 / 24500) loss: 1.301985
(Iteration 8851 / 24500) loss: 1.309310
(Iteration 8861 / 24500) loss: 1.274065
(Iteration 8871 / 24500) loss: 0.871986
(Iteration 8881 / 24500) loss: 1.272959
(Iteration 8891 / 24500) loss: 1.242803
(Iteration 8901 / 24500) loss: 1.304374
(Iteration 8911 / 24500) loss: 1.330120
(Iteration 8921 / 24500) loss: 1.179631
(Iteration 8931 / 24500) loss: 1.254517
(Iteration 8941 / 24500) loss: 1.247349
(Iteration 8951 / 24500) loss: 1.139868
(Iteration 8961 / 24500) loss: 1.101251
```

```
(Iteration 8971 / 24500) loss: 1.224941
(Iteration 8981 / 24500) loss: 1.223572
(Iteration 8991 / 24500) loss: 1.203040
(Iteration 9001 / 24500) loss: 1.258107
(Iteration 9011 / 24500) loss: 1.133590
(Iteration 9021 / 24500) loss: 1.067164
(Iteration 9031 / 24500) loss: 1.326334
(Iteration 9041 / 24500) loss: 1.224898
(Iteration 9051 / 24500) loss: 1.061807
(Iteration 9061 / 24500) loss: 1.130589
(Iteration 9071 / 24500) loss: 1.136683
(Iteration 9081 / 24500) loss: 1.122121
(Iteration 9091 / 24500) loss: 1.389976
(Iteration 9101 / 24500) loss: 0.967154
(Iteration 9111 / 24500) loss: 1.237680
(Iteration 9121 / 24500) loss: 1.112864
(Iteration 9131 / 24500) loss: 1.229352
(Iteration 9141 / 24500) loss: 1.290264
(Iteration 9151 / 24500) loss: 1.215676
(Iteration 9161 / 24500) loss: 1.039630
(Iteration 9171 / 24500) loss: 1.155787
(Iteration 9181 / 24500) loss: 1.109772
(Iteration 9191 / 24500) loss: 1.345622
(Iteration 9201 / 24500) loss: 1.278545
(Iteration 9211 / 24500) loss: 1.147436
(Iteration 9221 / 24500) loss: 1.149171
(Iteration 9231 / 24500) loss: 1.171880
(Iteration 9241 / 24500) loss: 1.103172
(Iteration 9251 / 24500) loss: 1.294735
(Iteration 9261 / 24500) loss: 1.206306
(Iteration 9271 / 24500) loss: 1.030826
(Iteration 9281 / 24500) loss: 1.359326
(Iteration 9291 / 24500) loss: 1.172444
(Iteration 9301 / 24500) loss: 1.273866
(Epoch 19 / 50) train acc: 0.555000; val_acc: 0.552000
(Iteration 9311 / 24500) loss: 1.144442
(Iteration 9321 / 24500) loss: 1.242315
(Iteration 9331 / 24500) loss: 1.244799
(Iteration 9341 / 24500) loss: 1.196489
(Iteration 9351 / 24500) loss: 1.115092
(Iteration 9361 / 24500) loss: 1.191247
(Iteration 9371 / 24500) loss: 1.306665
(Iteration 9381 / 24500) loss: 1.177902
(Iteration 9391 / 24500) loss: 1.187529
(Iteration 9401 / 24500) loss: 1.138814
(Iteration 9411 / 24500) loss: 1.203087
(Iteration 9421 / 24500) loss: 1.200692
(Iteration 9431 / 24500) loss: 1.189176
```

```
(Iteration 9441 / 24500) loss: 1.139854
(Iteration 9451 / 24500) loss: 1.100365
(Iteration 9461 / 24500) loss: 1.261724
(Iteration 9471 / 24500) loss: 1.221654
(Iteration 9481 / 24500) loss: 1.139704
(Iteration 9491 / 24500) loss: 1.189980
(Iteration 9501 / 24500) loss: 1.128609
(Iteration 9511 / 24500) loss: 1.311630
(Iteration 9521 / 24500) loss: 1.125121
(Iteration 9531 / 24500) loss: 1.182774
(Iteration 9541 / 24500) loss: 1.076754
(Iteration 9551 / 24500) loss: 1.048238
(Iteration 9561 / 24500) loss: 0.964429
(Iteration 9571 / 24500) loss: 1.328659
(Iteration 9581 / 24500) loss: 1.027843
(Iteration 9591 / 24500) loss: 1.176144
(Iteration 9601 / 24500) loss: 1.461817
(Iteration 9611 / 24500) loss: 1.126339
(Iteration 9621 / 24500) loss: 1.190837
(Iteration 9631 / 24500) loss: 1.211073
(Iteration 9641 / 24500) loss: 1.175863
(Iteration 9651 / 24500) loss: 1.174651
(Iteration 9661 / 24500) loss: 1.206433
(Iteration 9671 / 24500) loss: 1.293884
(Iteration 9681 / 24500) loss: 1.148010
(Iteration 9691 / 24500) loss: 1.071172
(Iteration 9701 / 24500) loss: 1.040899
(Iteration 9711 / 24500) loss: 1.162025
(Iteration 9721 / 24500) loss: 1.089466
(Iteration 9731 / 24500) loss: 1.271809
(Iteration 9741 / 24500) loss: 1.152171
(Iteration 9751 / 24500) loss: 1.305221
(Iteration 9761 / 24500) loss: 1.261034
(Iteration 9771 / 24500) loss: 1.220334
(Iteration 9781 / 24500) loss: 1.163506
(Iteration 9791 / 24500) loss: 1.378853
(Epoch 20 / 50) train acc: 0.570000; val_acc: 0.559000
(Iteration 9801 / 24500) loss: 1.228368
(Iteration 9811 / 24500) loss: 1.188681
(Iteration 9821 / 24500) loss: 1.111548
(Iteration 9831 / 24500) loss: 1.264449
(Iteration 9841 / 24500) loss: 1.129328
(Iteration 9851 / 24500) loss: 1.208548
(Iteration 9861 / 24500) loss: 1.216026
(Iteration 9871 / 24500) loss: 1.306622
(Iteration 9881 / 24500) loss: 1.074517
(Iteration 9891 / 24500) loss: 1.232258
(Iteration 9901 / 24500) loss: 1.051884
```

```
(Iteration 9911 / 24500) loss: 1.178279
(Iteration 9921 / 24500) loss: 1.291007
(Iteration 9931 / 24500) loss: 1.244186
(Iteration 9941 / 24500) loss: 1.312110
(Iteration 9951 / 24500) loss: 1.292581
(Iteration 9961 / 24500) loss: 1.134115
(Iteration 9971 / 24500) loss: 1.063479
(Iteration 9981 / 24500) loss: 1.325707
(Iteration 9991 / 24500) loss: 1.179336
(Iteration 10001 / 24500) loss: 0.949823
(Iteration 10011 / 24500) loss: 1.244988
(Iteration 10021 / 24500) loss: 1.314647
(Iteration 10031 / 24500) loss: 1.056794
(Iteration 10041 / 24500) loss: 1.194076
(Iteration 10051 / 24500) loss: 1.109865
(Iteration 10061 / 24500) loss: 1.179605
(Iteration 10071 / 24500) loss: 1.172753
(Iteration 10081 / 24500) loss: 1.271045
(Iteration 10091 / 24500) loss: 1.286640
(Iteration 10101 / 24500) loss: 1.180924
(Iteration 10111 / 24500) loss: 1.117815
(Iteration 10121 / 24500) loss: 1.106278
(Iteration 10131 / 24500) loss: 1.339739
(Iteration 10141 / 24500) loss: 1.203367
(Iteration 10151 / 24500) loss: 1.192590
(Iteration 10161 / 24500) loss: 1.253860
(Iteration 10171 / 24500) loss: 1.238494
(Iteration 10181 / 24500) loss: 1.217806
(Iteration 10191 / 24500) loss: 1.183639
(Iteration 10201 / 24500) loss: 1.146931
(Iteration 10211 / 24500) loss: 1.141673
(Iteration 10221 / 24500) loss: 1.188307
(Iteration 10231 / 24500) loss: 1.144697
(Iteration 10241 / 24500) loss: 1.206839
(Iteration 10251 / 24500) loss: 1.156052
(Iteration 10261 / 24500) loss: 1.352227
(Iteration 10271 / 24500) loss: 1.285258
(Iteration 10281 / 24500) loss: 1.173993
(Epoch 21 / 50) train acc: 0.576000; val_acc: 0.562000
(Iteration 10291 / 24500) loss: 0.967593
(Iteration 10301 / 24500) loss: 1.213117
(Iteration 10311 / 24500) loss: 1.166397
(Iteration 10321 / 24500) loss: 1.061035
(Iteration 10331 / 24500) loss: 1.269611
(Iteration 10341 / 24500) loss: 1.153755
(Iteration 10351 / 24500) loss: 1.184687
(Iteration 10361 / 24500) loss: 1.192537
(Iteration 10371 / 24500) loss: 1.253857
```

```
(Iteration 10381 / 24500) loss: 1.131292
(Iteration 10391 / 24500) loss: 1.087235
(Iteration 10401 / 24500) loss: 1.303225
(Iteration 10411 / 24500) loss: 1.212795
(Iteration 10421 / 24500) loss: 1.090573
(Iteration 10431 / 24500) loss: 1.261742
(Iteration 10441 / 24500) loss: 1.114452
(Iteration 10451 / 24500) loss: 1.035198
(Iteration 10461 / 24500) loss: 1.167452
(Iteration 10471 / 24500) loss: 1.091522
(Iteration 10481 / 24500) loss: 1.138687
(Iteration 10491 / 24500) loss: 1.294916
(Iteration 10501 / 24500) loss: 1.214857
(Iteration 10511 / 24500) loss: 1.234631
(Iteration 10521 / 24500) loss: 1.151801
(Iteration 10531 / 24500) loss: 1.273510
(Iteration 10541 / 24500) loss: 1.029370
(Iteration 10551 / 24500) loss: 1.044073
(Iteration 10561 / 24500) loss: 1.188231
(Iteration 10571 / 24500) loss: 1.086942
(Iteration 10581 / 24500) loss: 1.087665
(Iteration 10591 / 24500) loss: 1.394228
(Iteration 10601 / 24500) loss: 1.158985
(Iteration 10611 / 24500) loss: 1.160670
(Iteration 10621 / 24500) loss: 1.324423
(Iteration 10631 / 24500) loss: 1.152393
(Iteration 10641 / 24500) loss: 1.083063
(Iteration 10651 / 24500) loss: 1.081676
(Iteration 10661 / 24500) loss: 1.139365
(Iteration 10671 / 24500) loss: 1.155989
(Iteration 10681 / 24500) loss: 1.175551
(Iteration 10691 / 24500) loss: 1.241271
(Iteration 10701 / 24500) loss: 1.108441
(Iteration 10711 / 24500) loss: 1.081262
(Iteration 10721 / 24500) loss: 1.195733
(Iteration 10731 / 24500) loss: 1.098140
(Iteration 10741 / 24500) loss: 1.201554
(Iteration 10751 / 24500) loss: 1.314396
(Iteration 10761 / 24500) loss: 1.157679
(Iteration 10771 / 24500) loss: 1.258225
(Epoch 22 / 50) train acc: 0.603000; val_acc: 0.566000
(Iteration 10781 / 24500) loss: 1.146192
(Iteration 10791 / 24500) loss: 1.118651
(Iteration 10801 / 24500) loss: 1.226248
(Iteration 10811 / 24500) loss: 1.113170
(Iteration 10821 / 24500) loss: 1.182637
(Iteration 10831 / 24500) loss: 1.125679
(Iteration 10841 / 24500) loss: 1.365761
```

```
(Iteration 10851 / 24500) loss: 1.040932
(Iteration 10861 / 24500) loss: 1.276938
(Iteration 10871 / 24500) loss: 1.232947
(Iteration 10881 / 24500) loss: 1.218119
(Iteration 10891 / 24500) loss: 1.269494
(Iteration 10901 / 24500) loss: 1.250626
(Iteration 10911 / 24500) loss: 1.262312
(Iteration 10921 / 24500) loss: 1.059932
(Iteration 10931 / 24500) loss: 1.201268
(Iteration 10941 / 24500) loss: 1.199795
(Iteration 10951 / 24500) loss: 1.111762
(Iteration 10961 / 24500) loss: 1.144071
(Iteration 10971 / 24500) loss: 1.040781
(Iteration 10981 / 24500) loss: 1.039851
(Iteration 10991 / 24500) loss: 1.136513
(Iteration 11001 / 24500) loss: 1.222772
(Iteration 11011 / 24500) loss: 1.220280
(Iteration 11021 / 24500) loss: 1.200275
(Iteration 11031 / 24500) loss: 1.129579
(Iteration 11041 / 24500) loss: 1.332076
(Iteration 11051 / 24500) loss: 1.138346
(Iteration 11061 / 24500) loss: 1.127290
(Iteration 11071 / 24500) loss: 1.232231
(Iteration 11081 / 24500) loss: 1.124189
(Iteration 11091 / 24500) loss: 1.006339
(Iteration 11101 / 24500) loss: 0.976143
(Iteration 11111 / 24500) loss: 1.353698
(Iteration 11121 / 24500) loss: 1.151568
(Iteration 11131 / 24500) loss: 1.198005
(Iteration 11141 / 24500) loss: 1.356845
(Iteration 11151 / 24500) loss: 1.280476
(Iteration 11161 / 24500) loss: 1.123343
(Iteration 11171 / 24500) loss: 1.199304
(Iteration 11181 / 24500) loss: 1.167238
(Iteration 11191 / 24500) loss: 1.110951
(Iteration 11201 / 24500) loss: 1.148875
(Iteration 11211 / 24500) loss: 1.137278
(Iteration 11221 / 24500) loss: 1.145245
(Iteration 11231 / 24500) loss: 1.204433
(Iteration 11241 / 24500) loss: 1.249065
(Iteration 11251 / 24500) loss: 1.006573
(Iteration 11261 / 24500) loss: 1.124379
(Epoch 23 / 50) train acc: 0.606000; val_acc: 0.561000
(Iteration 11271 / 24500) loss: 1.247653
(Iteration 11281 / 24500) loss: 1.082157
(Iteration 11291 / 24500) loss: 1.327401
(Iteration 11301 / 24500) loss: 1.074244
(Iteration 11311 / 24500) loss: 1.110786
```

```
(Iteration 11321 / 24500) loss: 1.118097
(Iteration 11331 / 24500) loss: 1.174790
(Iteration 11341 / 24500) loss: 1.081870
(Iteration 11351 / 24500) loss: 1.137654
(Iteration 11361 / 24500) loss: 1.051603
(Iteration 11371 / 24500) loss: 1.030111
(Iteration 11381 / 24500) loss: 1.029547
(Iteration 11391 / 24500) loss: 1.020601
(Iteration 11401 / 24500) loss: 1.134569
(Iteration 11411 / 24500) loss: 1.219181
(Iteration 11421 / 24500) loss: 1.257750
(Iteration 11431 / 24500) loss: 1.181934
(Iteration 11441 / 24500) loss: 1.207382
(Iteration 11451 / 24500) loss: 1.012719
(Iteration 11461 / 24500) loss: 1.182496
(Iteration 11471 / 24500) loss: 1.293403
(Iteration 11481 / 24500) loss: 1.035444
(Iteration 11491 / 24500) loss: 1.096743
(Iteration 11501 / 24500) loss: 1.180651
(Iteration 11511 / 24500) loss: 1.225549
(Iteration 11521 / 24500) loss: 1.200660
(Iteration 11531 / 24500) loss: 1.048903
(Iteration 11541 / 24500) loss: 1.122166
(Iteration 11551 / 24500) loss: 1.096140
(Iteration 11561 / 24500) loss: 1.041674
(Iteration 11571 / 24500) loss: 1.032131
(Iteration 11581 / 24500) loss: 1.205546
(Iteration 11591 / 24500) loss: 1.174282
(Iteration 11601 / 24500) loss: 1.250023
(Iteration 11611 / 24500) loss: 0.981612
(Iteration 11621 / 24500) loss: 1.258225
(Iteration 11631 / 24500) loss: 1.094424
(Iteration 11641 / 24500) loss: 1.021359
(Iteration 11651 / 24500) loss: 1.139917
(Iteration 11661 / 24500) loss: 1.118312
(Iteration 11671 / 24500) loss: 1.296734
(Iteration 11681 / 24500) loss: 1.001112
(Iteration 11691 / 24500) loss: 1.155494
(Iteration 11701 / 24500) loss: 0.958166
(Iteration 11711 / 24500) loss: 0.944170
(Iteration 11721 / 24500) loss: 0.960781
(Iteration 11731 / 24500) loss: 1.210759
(Iteration 11741 / 24500) loss: 1.149812
(Iteration 11751 / 24500) loss: 1.096380
(Epoch 24 / 50) train acc: 0.584000; val_acc: 0.566000
(Iteration 11761 / 24500) loss: 1.176406
(Iteration 11771 / 24500) loss: 1.061495
(Iteration 11781 / 24500) loss: 1.253538
```

```
(Iteration 11791 / 24500) loss: 1.114047
(Iteration 11801 / 24500) loss: 1.220198
(Iteration 11811 / 24500) loss: 1.277417
(Iteration 11821 / 24500) loss: 1.156991
(Iteration 11831 / 24500) loss: 1.182829
(Iteration 11841 / 24500) loss: 1.174248
(Iteration 11851 / 24500) loss: 1.162763
(Iteration 11861 / 24500) loss: 1.121561
(Iteration 11871 / 24500) loss: 1.193917
(Iteration 11881 / 24500) loss: 1.259884
(Iteration 11891 / 24500) loss: 1.208534
(Iteration 11901 / 24500) loss: 1.124552
(Iteration 11911 / 24500) loss: 1.281251
(Iteration 11921 / 24500) loss: 1.112822
(Iteration 11931 / 24500) loss: 1.330039
(Iteration 11941 / 24500) loss: 0.966528
(Iteration 11951 / 24500) loss: 1.126329
(Iteration 11961 / 24500) loss: 1.119742
(Iteration 11971 / 24500) loss: 1.204483
(Iteration 11981 / 24500) loss: 1.235326
(Iteration 11991 / 24500) loss: 1.122985
(Iteration 12001 / 24500) loss: 1.163419
(Iteration 12011 / 24500) loss: 1.258883
(Iteration 12021 / 24500) loss: 1.172376
(Iteration 12031 / 24500) loss: 1.160400
(Iteration 12041 / 24500) loss: 1.046566
(Iteration 12051 / 24500) loss: 1.148754
(Iteration 12061 / 24500) loss: 1.058620
(Iteration 12071 / 24500) loss: 0.948765
(Iteration 12081 / 24500) loss: 1.235181
(Iteration 12091 / 24500) loss: 1.068916
(Iteration 12101 / 24500) loss: 1.158426
(Iteration 12111 / 24500) loss: 1.090254
(Iteration 12121 / 24500) loss: 1.067206
(Iteration 12131 / 24500) loss: 1.308564
(Iteration 12141 / 24500) loss: 1.167212
(Iteration 12151 / 24500) loss: 1.106762
(Iteration 12161 / 24500) loss: 1.265586
(Iteration 12171 / 24500) loss: 1.290161
(Iteration 12181 / 24500) loss: 1.113766
(Iteration 12191 / 24500) loss: 1.076918
(Iteration 12201 / 24500) loss: 0.962987
(Iteration 12211 / 24500) loss: 1.169428
(Iteration 12221 / 24500) loss: 1.189395
(Iteration 12231 / 24500) loss: 1.086900
(Iteration 12241 / 24500) loss: 1.083625
(Epoch 25 / 50) train acc: 0.592000; val_acc: 0.566000
(Iteration 12251 / 24500) loss: 1.159745
```

```
(Iteration 12261 / 24500) loss: 1.072463
(Iteration 12271 / 24500) loss: 1.147716
(Iteration 12281 / 24500) loss: 1.190456
(Iteration 12291 / 24500) loss: 1.098966
(Iteration 12301 / 24500) loss: 1.164664
(Iteration 12311 / 24500) loss: 1.091449
(Iteration 12321 / 24500) loss: 1.219447
(Iteration 12331 / 24500) loss: 1.297888
(Iteration 12341 / 24500) loss: 1.128274
(Iteration 12351 / 24500) loss: 1.096640
(Iteration 12361 / 24500) loss: 1.224830
(Iteration 12371 / 24500) loss: 1.193895
(Iteration 12381 / 24500) loss: 1.246027
(Iteration 12391 / 24500) loss: 1.219467
(Iteration 12401 / 24500) loss: 1.177625
(Iteration 12411 / 24500) loss: 1.104505
(Iteration 12421 / 24500) loss: 1.074104
(Iteration 12431 / 24500) loss: 1.246029
(Iteration 12441 / 24500) loss: 1.139822
(Iteration 12451 / 24500) loss: 1.146619
(Iteration 12461 / 24500) loss: 1.062648
(Iteration 12471 / 24500) loss: 1.251669
(Iteration 12481 / 24500) loss: 1.187132
(Iteration 12491 / 24500) loss: 0.971960
(Iteration 12501 / 24500) loss: 1.106986
(Iteration 12511 / 24500) loss: 1.250331
(Iteration 12521 / 24500) loss: 1.188787
(Iteration 12531 / 24500) loss: 1.087793
(Iteration 12541 / 24500) loss: 1.256568
(Iteration 12551 / 24500) loss: 1.159523
(Iteration 12561 / 24500) loss: 1.001879
(Iteration 12571 / 24500) loss: 1.243254
(Iteration 12581 / 24500) loss: 1.298049
(Iteration 12591 / 24500) loss: 1.275313
(Iteration 12601 / 24500) loss: 1.149048
(Iteration 12611 / 24500) loss: 1.294945
(Iteration 12621 / 24500) loss: 1.196204
(Iteration 12631 / 24500) loss: 1.049542
(Iteration 12641 / 24500) loss: 1.170945
(Iteration 12651 / 24500) loss: 1.027237
(Iteration 12661 / 24500) loss: 1.150479
(Iteration 12671 / 24500) loss: 1.248260
(Iteration 12681 / 24500) loss: 1.124460
(Iteration 12691 / 24500) loss: 1.032179
(Iteration 12701 / 24500) loss: 1.186552
(Iteration 12711 / 24500) loss: 1.139283
(Iteration 12721 / 24500) loss: 1.252692
(Iteration 12731 / 24500) loss: 1.189288
```

```
(Epoch 26 / 50) train acc: 0.569000; val_acc: 0.573000
(Iteration 12741 / 24500) loss: 1.096351
(Iteration 12751 / 24500) loss: 1.135096
(Iteration 12761 / 24500) loss: 1.093108
(Iteration 12771 / 24500) loss: 1.070847
(Iteration 12781 / 24500) loss: 1.069108
(Iteration 12791 / 24500) loss: 1.171406
(Iteration 12801 / 24500) loss: 1.226068
(Iteration 12811 / 24500) loss: 1.408358
(Iteration 12821 / 24500) loss: 1.087349
(Iteration 12831 / 24500) loss: 1.231225
(Iteration 12841 / 24500) loss: 1.074159
(Iteration 12851 / 24500) loss: 1.002022
(Iteration 12861 / 24500) loss: 0.918373
(Iteration 12871 / 24500) loss: 0.984458
(Iteration 12881 / 24500) loss: 1.140567
(Iteration 12891 / 24500) loss: 1.196831
(Iteration 12901 / 24500) loss: 1.115970
(Iteration 12911 / 24500) loss: 1.054184
(Iteration 12921 / 24500) loss: 1.037573
(Iteration 12931 / 24500) loss: 1.054343
(Iteration 12941 / 24500) loss: 1.127949
(Iteration 12951 / 24500) loss: 1.213558
(Iteration 12961 / 24500) loss: 0.937826
(Iteration 12971 / 24500) loss: 1.135215
(Iteration 12981 / 24500) loss: 1.087456
(Iteration 12991 / 24500) loss: 1.109628
(Iteration 13001 / 24500) loss: 1.098710
(Iteration 13011 / 24500) loss: 0.921858
(Iteration 13021 / 24500) loss: 1.117051
(Iteration 13031 / 24500) loss: 1.212409
(Iteration 13041 / 24500) loss: 1.138045
(Iteration 13051 / 24500) loss: 1.118699
(Iteration 13061 / 24500) loss: 1.011849
(Iteration 13071 / 24500) loss: 1.076170
(Iteration 13081 / 24500) loss: 1.351990
(Iteration 13091 / 24500) loss: 1.265623
(Iteration 13101 / 24500) loss: 1.152459
(Iteration 13111 / 24500) loss: 1.064528
(Iteration 13121 / 24500) loss: 1.121994
(Iteration 13131 / 24500) loss: 1.003523
(Iteration 13141 / 24500) loss: 1.191653
(Iteration 13151 / 24500) loss: 0.938055
(Iteration 13161 / 24500) loss: 1.132097
(Iteration 13171 / 24500) loss: 1.139351
(Iteration 13181 / 24500) loss: 1.049716
(Iteration 13191 / 24500) loss: 1.081710
(Iteration 13201 / 24500) loss: 1.029133
```

```
(Iteration 13211 / 24500) loss: 1.054080
(Iteration 13221 / 24500) loss: 1.310188
(Epoch 27 / 50) train acc: 0.615000; val_acc: 0.566000
(Iteration 13231 / 24500) loss: 1.244853
(Iteration 13241 / 24500) loss: 1.041468
(Iteration 13251 / 24500) loss: 1.206078
(Iteration 13261 / 24500) loss: 1.189642
(Iteration 13271 / 24500) loss: 0.968061
(Iteration 13281 / 24500) loss: 0.958261
(Iteration 13291 / 24500) loss: 1.080427
(Iteration 13301 / 24500) loss: 1.130520
(Iteration 13311 / 24500) loss: 1.014085
(Iteration 13321 / 24500) loss: 1.010015
(Iteration 13331 / 24500) loss: 0.949275
(Iteration 13341 / 24500) loss: 1.012719
(Iteration 13351 / 24500) loss: 1.109257
(Iteration 13361 / 24500) loss: 1.153348
(Iteration 13371 / 24500) loss: 1.101842
(Iteration 13381 / 24500) loss: 1.067678
(Iteration 13391 / 24500) loss: 0.912534
(Iteration 13401 / 24500) loss: 0.984021
(Iteration 13411 / 24500) loss: 1.061003
(Iteration 13421 / 24500) loss: 1.068933
(Iteration 13431 / 24500) loss: 1.121221
(Iteration 13441 / 24500) loss: 1.053554
(Iteration 13451 / 24500) loss: 1.162975
(Iteration 13461 / 24500) loss: 1.078739
(Iteration 13471 / 24500) loss: 1.033874
(Iteration 13481 / 24500) loss: 1.124855
(Iteration 13491 / 24500) loss: 1.170005
(Iteration 13501 / 24500) loss: 1.140203
(Iteration 13511 / 24500) loss: 1.132378
(Iteration 13521 / 24500) loss: 1.200354
(Iteration 13531 / 24500) loss: 1.034731
(Iteration 13541 / 24500) loss: 1.041787
(Iteration 13551 / 24500) loss: 1.144956
(Iteration 13561 / 24500) loss: 1.136896
(Iteration 13571 / 24500) loss: 0.931678
(Iteration 13581 / 24500) loss: 1.054046
(Iteration 13591 / 24500) loss: 1.279262
(Iteration 13601 / 24500) loss: 1.253129
(Iteration 13611 / 24500) loss: 1.231866
(Iteration 13621 / 24500) loss: 1.108254
(Iteration 13631 / 24500) loss: 1.043867
(Iteration 13641 / 24500) loss: 1.161256
(Iteration 13651 / 24500) loss: 1.089367
(Iteration 13661 / 24500) loss: 1.093579
(Iteration 13671 / 24500) loss: 1.024924
```

```
(Iteration 13681 / 24500) loss: 1.291192
(Iteration 13691 / 24500) loss: 1.131030
(Iteration 13701 / 24500) loss: 1.214072
(Iteration 13711 / 24500) loss: 1.198506
(Epoch 28 / 50) train acc: 0.598000; val_acc: 0.574000
(Iteration 13721 / 24500) loss: 1.233705
(Iteration 13731 / 24500) loss: 1.353421
(Iteration 13741 / 24500) loss: 1.063569
(Iteration 13751 / 24500) loss: 0.972828
(Iteration 13761 / 24500) loss: 1.142493
(Iteration 13771 / 24500) loss: 1.189306
(Iteration 13781 / 24500) loss: 1.168695
(Iteration 13791 / 24500) loss: 1.283953
(Iteration 13801 / 24500) loss: 1.093205
(Iteration 13811 / 24500) loss: 1.090697
(Iteration 13821 / 24500) loss: 1.199834
(Iteration 13831 / 24500) loss: 1.052303
(Iteration 13841 / 24500) loss: 1.250274
(Iteration 13851 / 24500) loss: 1.188617
(Iteration 13861 / 24500) loss: 1.039268
(Iteration 13871 / 24500) loss: 1.063769
(Iteration 13881 / 24500) loss: 1.075918
(Iteration 13891 / 24500) loss: 1.066207
(Iteration 13901 / 24500) loss: 1.191121
(Iteration 13911 / 24500) loss: 1.172093
(Iteration 13921 / 24500) loss: 1.090067
(Iteration 13931 / 24500) loss: 0.978558
(Iteration 13941 / 24500) loss: 1.075659
(Iteration 13951 / 24500) loss: 1.001548
(Iteration 13961 / 24500) loss: 1.058628
(Iteration 13971 / 24500) loss: 1.067605
(Iteration 13981 / 24500) loss: 1.298155
(Iteration 13991 / 24500) loss: 1.172794
(Iteration 14001 / 24500) loss: 1.287988
(Iteration 14011 / 24500) loss: 1.018277
(Iteration 14021 / 24500) loss: 1.157177
(Iteration 14031 / 24500) loss: 1.162310
(Iteration 14041 / 24500) loss: 1.288670
(Iteration 14051 / 24500) loss: 0.990257
(Iteration 14061 / 24500) loss: 1.023966
(Iteration 14071 / 24500) loss: 0.997873
(Iteration 14081 / 24500) loss: 0.976599
(Iteration 14091 / 24500) loss: 1.013416
(Iteration 14101 / 24500) loss: 1.214687
(Iteration 14111 / 24500) loss: 1.095518
(Iteration 14121 / 24500) loss: 0.986089
(Iteration 14131 / 24500) loss: 1.154571
(Iteration 14141 / 24500) loss: 0.996330
```

```
(Iteration 14151 / 24500) loss: 1.012505
(Iteration 14161 / 24500) loss: 1.039638
(Iteration 14171 / 24500) loss: 1.108865
(Iteration 14181 / 24500) loss: 1.181400
(Iteration 14191 / 24500) loss: 1.337398
(Iteration 14201 / 24500) loss: 1.050535
(Epoch 29 / 50) train acc: 0.632000; val_acc: 0.576000
(Iteration 14211 / 24500) loss: 1.275090
(Iteration 14221 / 24500) loss: 1.051950
(Iteration 14231 / 24500) loss: 1.034407
(Iteration 14241 / 24500) loss: 1.228424
(Iteration 14251 / 24500) loss: 1.153029
(Iteration 14261 / 24500) loss: 0.924529
(Iteration 14271 / 24500) loss: 1.172270
(Iteration 14281 / 24500) loss: 1.147939
(Iteration 14291 / 24500) loss: 1.029505
(Iteration 14301 / 24500) loss: 1.136490
(Iteration 14311 / 24500) loss: 0.951944
(Iteration 14321 / 24500) loss: 1.156991
(Iteration 14331 / 24500) loss: 1.071435
(Iteration 14341 / 24500) loss: 1.214886
(Iteration 14351 / 24500) loss: 1.130683
(Iteration 14361 / 24500) loss: 1.328807
(Iteration 14371 / 24500) loss: 1.109442
(Iteration 14381 / 24500) loss: 1.198016
(Iteration 14391 / 24500) loss: 1.126524
(Iteration 14401 / 24500) loss: 1.008785
(Iteration 14411 / 24500) loss: 1.006935
(Iteration 14421 / 24500) loss: 0.924123
(Iteration 14431 / 24500) loss: 1.164039
(Iteration 14441 / 24500) loss: 1.020292
(Iteration 14451 / 24500) loss: 1.077456
(Iteration 14461 / 24500) loss: 1.134371
(Iteration 14471 / 24500) loss: 1.076101
(Iteration 14481 / 24500) loss: 1.000528
(Iteration 14491 / 24500) loss: 1.017048
(Iteration 14501 / 24500) loss: 1.109779
(Iteration 14511 / 24500) loss: 0.893556
(Iteration 14521 / 24500) loss: 1.093716
(Iteration 14531 / 24500) loss: 1.076747
(Iteration 14541 / 24500) loss: 1.035628
(Iteration 14551 / 24500) loss: 0.926263
(Iteration 14561 / 24500) loss: 1.233861
(Iteration 14571 / 24500) loss: 0.961464
(Iteration 14581 / 24500) loss: 1.073494
(Iteration 14591 / 24500) loss: 1.127530
(Iteration 14601 / 24500) loss: 1.032899
(Iteration 14611 / 24500) loss: 1.118761
```

```
(Iteration 14621 / 24500) loss: 1.223174
(Iteration 14631 / 24500) loss: 1.230932
(Iteration 14641 / 24500) loss: 1.147572
(Iteration 14651 / 24500) loss: 1.056307
(Iteration 14661 / 24500) loss: 1.135019
(Iteration 14671 / 24500) loss: 1.059248
(Iteration 14681 / 24500) loss: 0.892792
(Iteration 14691 / 24500) loss: 1.202829
(Epoch 30 / 50) train acc: 0.635000; val_acc: 0.578000
(Iteration 14701 / 24500) loss: 1.198060
(Iteration 14711 / 24500) loss: 1.123180
(Iteration 14721 / 24500) loss: 1.135709
(Iteration 14731 / 24500) loss: 0.920117
(Iteration 14741 / 24500) loss: 1.145660
(Iteration 14751 / 24500) loss: 1.014783
(Iteration 14761 / 24500) loss: 1.146752
(Iteration 14771 / 24500) loss: 0.912540
(Iteration 14781 / 24500) loss: 1.170322
(Iteration 14791 / 24500) loss: 1.129396
(Iteration 14801 / 24500) loss: 1.251603
(Iteration 14811 / 24500) loss: 1.255989
(Iteration 14821 / 24500) loss: 1.008922
(Iteration 14831 / 24500) loss: 1.003400
(Iteration 14841 / 24500) loss: 1.176042
(Iteration 14851 / 24500) loss: 1.115741
(Iteration 14861 / 24500) loss: 1.128209
(Iteration 14871 / 24500) loss: 1.062010
(Iteration 14881 / 24500) loss: 1.089981
(Iteration 14891 / 24500) loss: 1.211183
(Iteration 14901 / 24500) loss: 1.027838
(Iteration 14911 / 24500) loss: 1.013636
(Iteration 14921 / 24500) loss: 1.088656
(Iteration 14931 / 24500) loss: 0.964394
(Iteration 14941 / 24500) loss: 1.093293
(Iteration 14951 / 24500) loss: 1.130156
(Iteration 14961 / 24500) loss: 1.238512
(Iteration 14971 / 24500) loss: 1.014657
(Iteration 14981 / 24500) loss: 1.229528
(Iteration 14991 / 24500) loss: 1.114588
(Iteration 15001 / 24500) loss: 1.175897
(Iteration 15011 / 24500) loss: 1.128790
(Iteration 15021 / 24500) loss: 0.986229
(Iteration 15031 / 24500) loss: 1.206327
(Iteration 15041 / 24500) loss: 1.170786
(Iteration 15051 / 24500) loss: 1.006754
(Iteration 15061 / 24500) loss: 1.090848
(Iteration 15071 / 24500) loss: 1.104029
(Iteration 15081 / 24500) loss: 1.183306
```

```
(Iteration 15091 / 24500) loss: 0.964900
(Iteration 15101 / 24500) loss: 0.794202
(Iteration 15111 / 24500) loss: 1.006452
(Iteration 15121 / 24500) loss: 1.077196
(Iteration 15131 / 24500) loss: 1.043838
(Iteration 15141 / 24500) loss: 1.022898
(Iteration 15151 / 24500) loss: 0.890319
(Iteration 15161 / 24500) loss: 1.104803
(Iteration 15171 / 24500) loss: 0.983110
(Iteration 15181 / 24500) loss: 1.135437
(Epoch 31 / 50) train acc: 0.615000; val_acc: 0.584000
(Iteration 15191 / 24500) loss: 1.115524
(Iteration 15201 / 24500) loss: 0.973374
(Iteration 15211 / 24500) loss: 1.295928
(Iteration 15221 / 24500) loss: 1.133407
(Iteration 15231 / 24500) loss: 0.974653
(Iteration 15241 / 24500) loss: 0.995220
(Iteration 15251 / 24500) loss: 1.126598
(Iteration 15261 / 24500) loss: 1.084062
(Iteration 15271 / 24500) loss: 1.087483
(Iteration 15281 / 24500) loss: 1.046431
(Iteration 15291 / 24500) loss: 1.234240
(Iteration 15301 / 24500) loss: 1.168202
(Iteration 15311 / 24500) loss: 1.107483
(Iteration 15321 / 24500) loss: 1.098930
(Iteration 15331 / 24500) loss: 1.135015
(Iteration 15341 / 24500) loss: 1.070909
(Iteration 15351 / 24500) loss: 1.117934
(Iteration 15361 / 24500) loss: 1.048310
(Iteration 15371 / 24500) loss: 1.091001
(Iteration 15381 / 24500) loss: 1.031786
(Iteration 15391 / 24500) loss: 1.008466
(Iteration 15401 / 24500) loss: 0.951588
(Iteration 15411 / 24500) loss: 0.915699
(Iteration 15421 / 24500) loss: 1.218045
(Iteration 15431 / 24500) loss: 1.088478
(Iteration 15441 / 24500) loss: 1.098661
(Iteration 15451 / 24500) loss: 1.310403
(Iteration 15461 / 24500) loss: 0.977630
(Iteration 15471 / 24500) loss: 1.135770
(Iteration 15481 / 24500) loss: 0.999269
(Iteration 15491 / 24500) loss: 1.176594
(Iteration 15501 / 24500) loss: 1.157474
(Iteration 15511 / 24500) loss: 1.161437
(Iteration 15521 / 24500) loss: 0.975378
(Iteration 15531 / 24500) loss: 1.122737
(Iteration 15541 / 24500) loss: 1.169363
(Iteration 15551 / 24500) loss: 1.070321
```

```
(Iteration 15561 / 24500) loss: 1.260212
(Iteration 15571 / 24500) loss: 1.017794
(Iteration 15581 / 24500) loss: 1.187290
(Iteration 15591 / 24500) loss: 1.084405
(Iteration 15601 / 24500) loss: 1.145398
(Iteration 15611 / 24500) loss: 1.255858
(Iteration 15621 / 24500) loss: 1.051595
(Iteration 15631 / 24500) loss: 1.080420
(Iteration 15641 / 24500) loss: 1.275190
(Iteration 15651 / 24500) loss: 1.019732
(Iteration 15661 / 24500) loss: 1.125144
(Iteration 15671 / 24500) loss: 1.020169
(Epoch 32 / 50) train acc: 0.638000; val_acc: 0.588000
(Iteration 15681 / 24500) loss: 1.174985
(Iteration 15691 / 24500) loss: 1.096771
(Iteration 15701 / 24500) loss: 1.164207
(Iteration 15711 / 24500) loss: 1.159755
(Iteration 15721 / 24500) loss: 1.006963
(Iteration 15731 / 24500) loss: 0.924103
(Iteration 15741 / 24500) loss: 0.961342
(Iteration 15751 / 24500) loss: 1.096631
(Iteration 15761 / 24500) loss: 1.069200
(Iteration 15771 / 24500) loss: 1.120816
(Iteration 15781 / 24500) loss: 1.110101
(Iteration 15791 / 24500) loss: 0.985845
(Iteration 15801 / 24500) loss: 1.216845
(Iteration 15811 / 24500) loss: 0.986763
(Iteration 15821 / 24500) loss: 1.058484
(Iteration 15831 / 24500) loss: 0.942062
(Iteration 15841 / 24500) loss: 1.125041
(Iteration 15851 / 24500) loss: 1.152284
(Iteration 15861 / 24500) loss: 1.142045
(Iteration 15871 / 24500) loss: 1.132734
(Iteration 15881 / 24500) loss: 1.127151
(Iteration 15891 / 24500) loss: 1.182949
(Iteration 15901 / 24500) loss: 1.117428
(Iteration 15911 / 24500) loss: 1.152396
(Iteration 15921 / 24500) loss: 1.053362
(Iteration 15931 / 24500) loss: 0.986193
(Iteration 15941 / 24500) loss: 1.094832
(Iteration 15951 / 24500) loss: 1.020261
(Iteration 15961 / 24500) loss: 1.257558
(Iteration 15971 / 24500) loss: 1.139678
(Iteration 15981 / 24500) loss: 1.215343
(Iteration 15991 / 24500) loss: 1.056859
(Iteration 16001 / 24500) loss: 1.099883
(Iteration 16011 / 24500) loss: 1.174772
(Iteration 16021 / 24500) loss: 0.961161
```

```
(Iteration 16031 / 24500) loss: 1.005301
(Iteration 16041 / 24500) loss: 1.067489
(Iteration 16051 / 24500) loss: 1.008417
(Iteration 16061 / 24500) loss: 1.050957
(Iteration 16071 / 24500) loss: 0.967349
(Iteration 16081 / 24500) loss: 0.892006
(Iteration 16091 / 24500) loss: 0.935687
(Iteration 16101 / 24500) loss: 1.191397
(Iteration 16111 / 24500) loss: 1.137833
(Iteration 16121 / 24500) loss: 1.067298
(Iteration 16131 / 24500) loss: 1.144119
(Iteration 16141 / 24500) loss: 1.216399
(Iteration 16151 / 24500) loss: 1.102298
(Iteration 16161 / 24500) loss: 1.232031
(Epoch 33 / 50) train acc: 0.631000; val_acc: 0.591000
(Iteration 16171 / 24500) loss: 0.916776
(Iteration 16181 / 24500) loss: 1.169724
(Iteration 16191 / 24500) loss: 0.973372
(Iteration 16201 / 24500) loss: 1.083134
(Iteration 16211 / 24500) loss: 0.905204
(Iteration 16221 / 24500) loss: 1.047332
(Iteration 16231 / 24500) loss: 0.911368
(Iteration 16241 / 24500) loss: 1.026842
(Iteration 16251 / 24500) loss: 0.852456
(Iteration 16261 / 24500) loss: 0.895962
(Iteration 16271 / 24500) loss: 0.913371
(Iteration 16281 / 24500) loss: 1.000578
(Iteration 16291 / 24500) loss: 1.215543
(Iteration 16301 / 24500) loss: 1.110543
(Iteration 16311 / 24500) loss: 1.092018
(Iteration 16321 / 24500) loss: 1.048822
(Iteration 16331 / 24500) loss: 1.059189
(Iteration 16341 / 24500) loss: 1.105203
(Iteration 16351 / 24500) loss: 0.995690
(Iteration 16361 / 24500) loss: 0.987304
(Iteration 16371 / 24500) loss: 1.241019
(Iteration 16381 / 24500) loss: 0.928721
(Iteration 16391 / 24500) loss: 1.048390
(Iteration 16401 / 24500) loss: 1.145894
(Iteration 16411 / 24500) loss: 0.992663
(Iteration 16421 / 24500) loss: 0.945723
(Iteration 16431 / 24500) loss: 1.179252
(Iteration 16441 / 24500) loss: 0.987800
(Iteration 16451 / 24500) loss: 1.122579
(Iteration 16461 / 24500) loss: 1.043147
(Iteration 16471 / 24500) loss: 0.885921
(Iteration 16481 / 24500) loss: 1.211557
(Iteration 16491 / 24500) loss: 0.990963
```

```
(Iteration 16501 / 24500) loss: 0.977277
(Iteration 16511 / 24500) loss: 0.966621
(Iteration 16521 / 24500) loss: 1.121640
(Iteration 16531 / 24500) loss: 0.943754
(Iteration 16541 / 24500) loss: 1.076432
(Iteration 16551 / 24500) loss: 1.092792
(Iteration 16561 / 24500) loss: 0.988578
(Iteration 16571 / 24500) loss: 1.005797
(Iteration 16581 / 24500) loss: 1.123050
(Iteration 16591 / 24500) loss: 1.028219
(Iteration 16601 / 24500) loss: 1.174139
(Iteration 16611 / 24500) loss: 1.134189
(Iteration 16621 / 24500) loss: 1.146785
(Iteration 16631 / 24500) loss: 1.153671
(Iteration 16641 / 24500) loss: 1.043043
(Iteration 16651 / 24500) loss: 0.935560
(Epoch 34 / 50) train acc: 0.624000; val_acc: 0.584000
(Iteration 16661 / 24500) loss: 1.085947
(Iteration 16671 / 24500) loss: 1.081378
(Iteration 16681 / 24500) loss: 0.926823
(Iteration 16691 / 24500) loss: 1.102833
(Iteration 16701 / 24500) loss: 1.205343
(Iteration 16711 / 24500) loss: 1.092595
(Iteration 16721 / 24500) loss: 1.032618
(Iteration 16731 / 24500) loss: 1.104496
(Iteration 16741 / 24500) loss: 1.007577
(Iteration 16751 / 24500) loss: 1.171793
(Iteration 16761 / 24500) loss: 1.075259
(Iteration 16771 / 24500) loss: 1.028566
(Iteration 16781 / 24500) loss: 1.039586
(Iteration 16791 / 24500) loss: 0.970010
(Iteration 16801 / 24500) loss: 1.071168
(Iteration 16811 / 24500) loss: 1.113811
(Iteration 16821 / 24500) loss: 1.181082
(Iteration 16831 / 24500) loss: 0.831580
(Iteration 16841 / 24500) loss: 0.952473
(Iteration 16851 / 24500) loss: 1.180598
(Iteration 16861 / 24500) loss: 1.188098
(Iteration 16871 / 24500) loss: 0.963331
(Iteration 16881 / 24500) loss: 0.997385
(Iteration 16891 / 24500) loss: 1.184214
(Iteration 16901 / 24500) loss: 0.996960
(Iteration 16911 / 24500) loss: 1.120612
(Iteration 16921 / 24500) loss: 1.075236
(Iteration 16931 / 24500) loss: 1.106859
(Iteration 16941 / 24500) loss: 1.130890
(Iteration 16951 / 24500) loss: 1.007536
(Iteration 16961 / 24500) loss: 1.026840
```

```
(Iteration 16971 / 24500) loss: 0.931921
(Iteration 16981 / 24500) loss: 0.958351
(Iteration 16991 / 24500) loss: 1.047937
(Iteration 17001 / 24500) loss: 1.151185
(Iteration 17011 / 24500) loss: 1.150480
(Iteration 17021 / 24500) loss: 1.144352
(Iteration 17031 / 24500) loss: 0.934522
(Iteration 17041 / 24500) loss: 1.245966
(Iteration 17051 / 24500) loss: 0.916179
(Iteration 17061 / 24500) loss: 1.076940
(Iteration 17071 / 24500) loss: 0.926606
(Iteration 17081 / 24500) loss: 1.042257
(Iteration 17091 / 24500) loss: 1.117525
(Iteration 17101 / 24500) loss: 1.202307
(Iteration 17111 / 24500) loss: 1.114835
(Iteration 17121 / 24500) loss: 1.195336
(Iteration 17131 / 24500) loss: 0.930453
(Iteration 17141 / 24500) loss: 1.070083
(Epoch 35 / 50) train acc: 0.643000; val_acc: 0.586000
(Iteration 17151 / 24500) loss: 1.155217
(Iteration 17161 / 24500) loss: 1.031312
(Iteration 17171 / 24500) loss: 1.017723
(Iteration 17181 / 24500) loss: 0.967498
(Iteration 17191 / 24500) loss: 1.306768
(Iteration 17201 / 24500) loss: 1.190395
(Iteration 17211 / 24500) loss: 0.964806
(Iteration 17221 / 24500) loss: 1.005142
(Iteration 17231 / 24500) loss: 1.006540
(Iteration 17241 / 24500) loss: 0.929006
(Iteration 17251 / 24500) loss: 1.167552
(Iteration 17261 / 24500) loss: 1.115115
(Iteration 17271 / 24500) loss: 0.848666
(Iteration 17281 / 24500) loss: 1.177074
(Iteration 17291 / 24500) loss: 1.204896
(Iteration 17301 / 24500) loss: 1.014766
(Iteration 17311 / 24500) loss: 0.980316
(Iteration 17321 / 24500) loss: 1.189018
(Iteration 17331 / 24500) loss: 1.017226
(Iteration 17341 / 24500) loss: 1.165124
(Iteration 17351 / 24500) loss: 0.984088
(Iteration 17361 / 24500) loss: 1.024087
(Iteration 17371 / 24500) loss: 0.918211
(Iteration 17381 / 24500) loss: 1.018263
(Iteration 17391 / 24500) loss: 1.188932
(Iteration 17401 / 24500) loss: 1.135181
(Iteration 17411 / 24500) loss: 1.131608
(Iteration 17421 / 24500) loss: 1.051324
(Iteration 17431 / 24500) loss: 0.963376
```

```
(Iteration 17441 / 24500) loss: 1.073931
(Iteration 17451 / 24500) loss: 1.052901
(Iteration 17461 / 24500) loss: 1.095020
(Iteration 17471 / 24500) loss: 0.981071
(Iteration 17481 / 24500) loss: 0.956670
(Iteration 17491 / 24500) loss: 1.003268
(Iteration 17501 / 24500) loss: 1.049875
(Iteration 17511 / 24500) loss: 1.289865
(Iteration 17521 / 24500) loss: 1.063092
(Iteration 17531 / 24500) loss: 0.919508
(Iteration 17541 / 24500) loss: 1.029440
(Iteration 17551 / 24500) loss: 0.888872
(Iteration 17561 / 24500) loss: 1.017260
(Iteration 17571 / 24500) loss: 1.177242
(Iteration 17581 / 24500) loss: 1.092385
(Iteration 17591 / 24500) loss: 1.047695
(Iteration 17601 / 24500) loss: 1.071723
(Iteration 17611 / 24500) loss: 1.078129
(Iteration 17621 / 24500) loss: 0.840945
(Iteration 17631 / 24500) loss: 0.996078
(Epoch 36 / 50) train acc: 0.639000; val_acc: 0.585000
(Iteration 17641 / 24500) loss: 0.960786
(Iteration 17651 / 24500) loss: 1.033355
(Iteration 17661 / 24500) loss: 1.074737
(Iteration 17671 / 24500) loss: 0.881716
(Iteration 17681 / 24500) loss: 1.142331
(Iteration 17691 / 24500) loss: 1.040491
(Iteration 17701 / 24500) loss: 1.003914
(Iteration 17711 / 24500) loss: 1.080175
(Iteration 17721 / 24500) loss: 1.068122
(Iteration 17731 / 24500) loss: 1.121759
(Iteration 17741 / 24500) loss: 0.879641
(Iteration 17751 / 24500) loss: 1.034536
(Iteration 17761 / 24500) loss: 1.153452
(Iteration 17771 / 24500) loss: 1.108089
(Iteration 17781 / 24500) loss: 1.018852
(Iteration 17791 / 24500) loss: 1.177387
(Iteration 17801 / 24500) loss: 1.156937
(Iteration 17811 / 24500) loss: 1.025133
(Iteration 17821 / 24500) loss: 1.088177
(Iteration 17831 / 24500) loss: 1.071669
(Iteration 17841 / 24500) loss: 0.879366
(Iteration 17851 / 24500) loss: 0.859631
(Iteration 17861 / 24500) loss: 1.088461
(Iteration 17871 / 24500) loss: 0.989235
(Iteration 17881 / 24500) loss: 1.099164
(Iteration 17891 / 24500) loss: 1.012750
(Iteration 17901 / 24500) loss: 1.149964
```

```
(Iteration 17911 / 24500) loss: 0.918466
(Iteration 17921 / 24500) loss: 1.110532
(Iteration 17931 / 24500) loss: 1.034662
(Iteration 17941 / 24500) loss: 1.069170
(Iteration 17951 / 24500) loss: 1.063561
(Iteration 17961 / 24500) loss: 1.284226
(Iteration 17971 / 24500) loss: 1.330816
(Iteration 17981 / 24500) loss: 1.096267
(Iteration 17991 / 24500) loss: 1.120920
(Iteration 18001 / 24500) loss: 1.064107
(Iteration 18011 / 24500) loss: 1.118982
(Iteration 18021 / 24500) loss: 0.973984
(Iteration 18031 / 24500) loss: 0.874330
(Iteration 18041 / 24500) loss: 1.032345
(Iteration 18051 / 24500) loss: 1.106281
(Iteration 18061 / 24500) loss: 1.025747
(Iteration 18071 / 24500) loss: 1.008247
(Iteration 18081 / 24500) loss: 0.969836
(Iteration 18091 / 24500) loss: 0.961277
(Iteration 18101 / 24500) loss: 1.109637
(Iteration 18111 / 24500) loss: 1.113045
(Iteration 18121 / 24500) loss: 1.034012
(Epoch 37 / 50) train acc: 0.640000; val_acc: 0.586000
(Iteration 18131 / 24500) loss: 1.179274
(Iteration 18141 / 24500) loss: 0.978093
(Iteration 18151 / 24500) loss: 1.109095
(Iteration 18161 / 24500) loss: 1.230646
(Iteration 18171 / 24500) loss: 1.114778
(Iteration 18181 / 24500) loss: 1.086106
(Iteration 18191 / 24500) loss: 0.962194
(Iteration 18201 / 24500) loss: 0.907824
(Iteration 18211 / 24500) loss: 0.957468
(Iteration 18221 / 24500) loss: 1.220345
(Iteration 18231 / 24500) loss: 1.031421
(Iteration 18241 / 24500) loss: 1.191617
(Iteration 18251 / 24500) loss: 1.111450
(Iteration 18261 / 24500) loss: 1.049844
(Iteration 18271 / 24500) loss: 1.089413
(Iteration 18281 / 24500) loss: 0.997257
(Iteration 18291 / 24500) loss: 0.991234
(Iteration 18301 / 24500) loss: 1.195062
(Iteration 18311 / 24500) loss: 1.094142
(Iteration 18321 / 24500) loss: 1.227448
(Iteration 18331 / 24500) loss: 1.086599
(Iteration 18341 / 24500) loss: 0.953925
(Iteration 18351 / 24500) loss: 0.902762
(Iteration 18361 / 24500) loss: 1.017627
(Iteration 18371 / 24500) loss: 1.240073
```

```
(Iteration 18381 / 24500) loss: 1.021358
(Iteration 18391 / 24500) loss: 0.994520
(Iteration 18401 / 24500) loss: 0.837345
(Iteration 18411 / 24500) loss: 0.998971
(Iteration 18421 / 24500) loss: 0.920549
(Iteration 18431 / 24500) loss: 1.162956
(Iteration 18441 / 24500) loss: 0.900517
(Iteration 18451 / 24500) loss: 0.987750
(Iteration 18461 / 24500) loss: 0.928931
(Iteration 18471 / 24500) loss: 1.123776
(Iteration 18481 / 24500) loss: 0.990157
(Iteration 18491 / 24500) loss: 1.169121
(Iteration 18501 / 24500) loss: 0.916108
(Iteration 18511 / 24500) loss: 0.865345
(Iteration 18521 / 24500) loss: 0.940660
(Iteration 18531 / 24500) loss: 1.074491
(Iteration 18541 / 24500) loss: 1.132762
(Iteration 18551 / 24500) loss: 1.071664
(Iteration 18561 / 24500) loss: 1.083003
(Iteration 18571 / 24500) loss: 0.982136
(Iteration 18581 / 24500) loss: 0.983084
(Iteration 18591 / 24500) loss: 1.097399
(Iteration 18601 / 24500) loss: 1.007122
(Iteration 18611 / 24500) loss: 1.268596
(Epoch 38 / 50) train acc: 0.632000; val_acc: 0.587000
(Iteration 18621 / 24500) loss: 1.102414
(Iteration 18631 / 24500) loss: 1.242585
(Iteration 18641 / 24500) loss: 1.107865
(Iteration 18651 / 24500) loss: 1.194830
(Iteration 18661 / 24500) loss: 0.933923
(Iteration 18671 / 24500) loss: 1.096429
(Iteration 18681 / 24500) loss: 1.034409
(Iteration 18691 / 24500) loss: 0.930217
(Iteration 18701 / 24500) loss: 1.156902
(Iteration 18711 / 24500) loss: 1.195706
(Iteration 18721 / 24500) loss: 1.081821
(Iteration 18731 / 24500) loss: 1.038588
(Iteration 18741 / 24500) loss: 0.928406
(Iteration 18751 / 24500) loss: 1.063527
(Iteration 18761 / 24500) loss: 1.171630
(Iteration 18771 / 24500) loss: 1.174907
(Iteration 18781 / 24500) loss: 1.075425
(Iteration 18791 / 24500) loss: 1.106040
(Iteration 18801 / 24500) loss: 0.943460
(Iteration 18811 / 24500) loss: 1.360936
(Iteration 18821 / 24500) loss: 0.935322
(Iteration 18831 / 24500) loss: 0.954706
(Iteration 18841 / 24500) loss: 1.085439
```

```
(Iteration 18851 / 24500) loss: 0.889468
(Iteration 18861 / 24500) loss: 1.091226
(Iteration 18871 / 24500) loss: 1.022150
(Iteration 18881 / 24500) loss: 1.047636
(Iteration 18891 / 24500) loss: 0.908627
(Iteration 18901 / 24500) loss: 1.009350
(Iteration 18911 / 24500) loss: 1.040290
(Iteration 18921 / 24500) loss: 1.226083
(Iteration 18931 / 24500) loss: 1.125046
(Iteration 18941 / 24500) loss: 1.046895
(Iteration 18951 / 24500) loss: 1.086029
(Iteration 18961 / 24500) loss: 0.912029
(Iteration 18971 / 24500) loss: 0.888773
(Iteration 18981 / 24500) loss: 0.866059
(Iteration 18991 / 24500) loss: 0.933898
(Iteration 19001 / 24500) loss: 1.031515
(Iteration 19011 / 24500) loss: 1.095033
(Iteration 19021 / 24500) loss: 0.999525
(Iteration 19031 / 24500) loss: 1.043520
(Iteration 19041 / 24500) loss: 1.049609
(Iteration 19051 / 24500) loss: 0.945003
(Iteration 19061 / 24500) loss: 1.003573
(Iteration 19071 / 24500) loss: 1.185573
(Iteration 19081 / 24500) loss: 0.992358
(Iteration 19091 / 24500) loss: 1.106196
(Iteration 19101 / 24500) loss: 0.969637
(Epoch 39 / 50) train acc: 0.648000; val_acc: 0.594000
(Iteration 19111 / 24500) loss: 0.921879
(Iteration 19121 / 24500) loss: 1.097002
(Iteration 19131 / 24500) loss: 1.032927
(Iteration 19141 / 24500) loss: 0.983830
(Iteration 19151 / 24500) loss: 1.064573
(Iteration 19161 / 24500) loss: 1.009436
(Iteration 19171 / 24500) loss: 0.990081
(Iteration 19181 / 24500) loss: 0.982486
(Iteration 19191 / 24500) loss: 1.083435
(Iteration 19201 / 24500) loss: 1.124516
(Iteration 19211 / 24500) loss: 1.064261
(Iteration 19221 / 24500) loss: 1.003248
(Iteration 19231 / 24500) loss: 1.012153
(Iteration 19241 / 24500) loss: 1.219273
(Iteration 19251 / 24500) loss: 1.017235
(Iteration 19261 / 24500) loss: 1.022452
(Iteration 19271 / 24500) loss: 0.949470
(Iteration 19281 / 24500) loss: 0.959905
(Iteration 19291 / 24500) loss: 0.985095
(Iteration 19301 / 24500) loss: 0.955728
(Iteration 19311 / 24500) loss: 1.231702
```

```
(Iteration 19321 / 24500) loss: 0.850067
(Iteration 19331 / 24500) loss: 1.007412
(Iteration 19341 / 24500) loss: 1.117958
(Iteration 19351 / 24500) loss: 0.911166
(Iteration 19361 / 24500) loss: 0.966342
(Iteration 19371 / 24500) loss: 1.006581
(Iteration 19381 / 24500) loss: 0.837197
(Iteration 19391 / 24500) loss: 1.011646
(Iteration 19401 / 24500) loss: 1.063781
(Iteration 19411 / 24500) loss: 0.854243
(Iteration 19421 / 24500) loss: 0.974959
(Iteration 19431 / 24500) loss: 1.019661
(Iteration 19441 / 24500) loss: 1.092779
(Iteration 19451 / 24500) loss: 1.037947
(Iteration 19461 / 24500) loss: 1.146307
(Iteration 19471 / 24500) loss: 1.051480
(Iteration 19481 / 24500) loss: 1.210437
(Iteration 19491 / 24500) loss: 0.870877
(Iteration 19501 / 24500) loss: 0.988062
(Iteration 19511 / 24500) loss: 1.128883
(Iteration 19521 / 24500) loss: 0.967289
(Iteration 19531 / 24500) loss: 0.964792
(Iteration 19541 / 24500) loss: 0.807196
(Iteration 19551 / 24500) loss: 0.866207
(Iteration 19561 / 24500) loss: 1.096472
(Iteration 19571 / 24500) loss: 0.875288
(Iteration 19581 / 24500) loss: 0.952157
(Iteration 19591 / 24500) loss: 1.003730
(Epoch 40 / 50) train acc: 0.648000; val_acc: 0.587000
(Iteration 19601 / 24500) loss: 1.179671
(Iteration 19611 / 24500) loss: 1.003872
(Iteration 19621 / 24500) loss: 0.882716
(Iteration 19631 / 24500) loss: 0.992676
(Iteration 19641 / 24500) loss: 0.938455
(Iteration 19651 / 24500) loss: 0.838219
(Iteration 19661 / 24500) loss: 1.134350
(Iteration 19671 / 24500) loss: 1.108274
(Iteration 19681 / 24500) loss: 0.828984
(Iteration 19691 / 24500) loss: 0.951635
(Iteration 19701 / 24500) loss: 0.836549
(Iteration 19711 / 24500) loss: 1.039858
(Iteration 19721 / 24500) loss: 1.050875
(Iteration 19731 / 24500) loss: 1.082726
(Iteration 19741 / 24500) loss: 0.842040
(Iteration 19751 / 24500) loss: 1.011939
(Iteration 19761 / 24500) loss: 1.009534
(Iteration 19771 / 24500) loss: 0.923381
(Iteration 19781 / 24500) loss: 1.110106
```

```
(Iteration 19791 / 24500) loss: 1.023716
(Iteration 19801 / 24500) loss: 0.901077
(Iteration 19811 / 24500) loss: 1.201400
(Iteration 19821 / 24500) loss: 1.005002
(Iteration 19831 / 24500) loss: 1.151815
(Iteration 19841 / 24500) loss: 0.887933
(Iteration 19851 / 24500) loss: 1.080910
(Iteration 19861 / 24500) loss: 1.032451
(Iteration 19871 / 24500) loss: 0.987959
(Iteration 19881 / 24500) loss: 0.921673
(Iteration 19891 / 24500) loss: 0.948590
(Iteration 19901 / 24500) loss: 0.940797
(Iteration 19911 / 24500) loss: 0.919743
(Iteration 19921 / 24500) loss: 0.898807
(Iteration 19931 / 24500) loss: 1.162027
(Iteration 19941 / 24500) loss: 0.932400
(Iteration 19951 / 24500) loss: 0.949145
(Iteration 19961 / 24500) loss: 1.013471
(Iteration 19971 / 24500) loss: 1.064432
(Iteration 19981 / 24500) loss: 0.831933
(Iteration 19991 / 24500) loss: 1.019437
(Iteration 20001 / 24500) loss: 1.194098
(Iteration 20011 / 24500) loss: 0.941681
(Iteration 20021 / 24500) loss: 1.106285
(Iteration 20031 / 24500) loss: 0.960640
(Iteration 20041 / 24500) loss: 1.183901
(Iteration 20051 / 24500) loss: 0.916966
(Iteration 20061 / 24500) loss: 1.002302
(Iteration 20071 / 24500) loss: 1.034303
(Iteration 20081 / 24500) loss: 0.817486
(Epoch 41 / 50) train acc: 0.630000; val_acc: 0.590000
(Iteration 20091 / 24500) loss: 1.076197
(Iteration 20101 / 24500) loss: 1.100015
(Iteration 20111 / 24500) loss: 1.275672
(Iteration 20121 / 24500) loss: 1.014821
(Iteration 20131 / 24500) loss: 0.979136
(Iteration 20141 / 24500) loss: 1.142949
(Iteration 20151 / 24500) loss: 1.210126
(Iteration 20161 / 24500) loss: 0.931656
(Iteration 20171 / 24500) loss: 0.892390
(Iteration 20181 / 24500) loss: 0.882381
(Iteration 20191 / 24500) loss: 0.944591
(Iteration 20201 / 24500) loss: 1.256709
(Iteration 20211 / 24500) loss: 1.033598
(Iteration 20221 / 24500) loss: 1.062713
(Iteration 20231 / 24500) loss: 0.944053
(Iteration 20241 / 24500) loss: 1.046475
(Iteration 20251 / 24500) loss: 0.961757
```

```
(Iteration 20261 / 24500) loss: 1.021177
(Iteration 20271 / 24500) loss: 0.968497
(Iteration 20281 / 24500) loss: 0.990615
(Iteration 20291 / 24500) loss: 0.849649
(Iteration 20301 / 24500) loss: 1.005873
(Iteration 20311 / 24500) loss: 1.046288
(Iteration 20321 / 24500) loss: 1.051048
(Iteration 20331 / 24500) loss: 1.012767
(Iteration 20341 / 24500) loss: 0.945649
(Iteration 20351 / 24500) loss: 1.036530
(Iteration 20361 / 24500) loss: 1.090426
(Iteration 20371 / 24500) loss: 1.192763
(Iteration 20381 / 24500) loss: 0.994652
(Iteration 20391 / 24500) loss: 0.996836
(Iteration 20401 / 24500) loss: 0.979890
(Iteration 20411 / 24500) loss: 0.970342
(Iteration 20421 / 24500) loss: 1.024410
(Iteration 20431 / 24500) loss: 1.172742
(Iteration 20441 / 24500) loss: 1.005428
(Iteration 20451 / 24500) loss: 0.900140
(Iteration 20461 / 24500) loss: 1.116671
(Iteration 20471 / 24500) loss: 1.168794
(Iteration 20481 / 24500) loss: 0.898764
(Iteration 20491 / 24500) loss: 1.048895
(Iteration 20501 / 24500) loss: 1.020654
(Iteration 20511 / 24500) loss: 1.149472
(Iteration 20521 / 24500) loss: 0.877973
(Iteration 20531 / 24500) loss: 0.982892
(Iteration 20541 / 24500) loss: 1.020385
(Iteration 20551 / 24500) loss: 1.020977
(Iteration 20561 / 24500) loss: 1.000680
(Iteration 20571 / 24500) loss: 1.156202
(Epoch 42 / 50) train acc: 0.666000; val_acc: 0.579000
(Iteration 20581 / 24500) loss: 1.136270
(Iteration 20591 / 24500) loss: 0.894734
(Iteration 20601 / 24500) loss: 0.963267
(Iteration 20611 / 24500) loss: 1.073845
(Iteration 20621 / 24500) loss: 1.031465
(Iteration 20631 / 24500) loss: 0.817584
(Iteration 20641 / 24500) loss: 0.881226
(Iteration 20651 / 24500) loss: 1.014726
(Iteration 20661 / 24500) loss: 1.105094
(Iteration 20671 / 24500) loss: 0.964453
(Iteration 20681 / 24500) loss: 0.902818
(Iteration 20691 / 24500) loss: 1.049362
(Iteration 20701 / 24500) loss: 1.023990
(Iteration 20711 / 24500) loss: 1.037067
(Iteration 20721 / 24500) loss: 1.061182
```

```
(Iteration 20731 / 24500) loss: 0.871041
(Iteration 20741 / 24500) loss: 0.942686
(Iteration 20751 / 24500) loss: 1.075516
(Iteration 20761 / 24500) loss: 1.255859
(Iteration 20771 / 24500) loss: 0.966506
(Iteration 20781 / 24500) loss: 1.004903
(Iteration 20791 / 24500) loss: 1.082921
(Iteration 20801 / 24500) loss: 0.944207
(Iteration 20811 / 24500) loss: 0.951801
(Iteration 20821 / 24500) loss: 0.730193
(Iteration 20831 / 24500) loss: 0.928128
(Iteration 20841 / 24500) loss: 0.990624
(Iteration 20851 / 24500) loss: 0.942848
(Iteration 20861 / 24500) loss: 0.941905
(Iteration 20871 / 24500) loss: 1.044578
(Iteration 20881 / 24500) loss: 1.121119
(Iteration 20891 / 24500) loss: 0.922257
(Iteration 20901 / 24500) loss: 1.013529
(Iteration 20911 / 24500) loss: 1.116861
(Iteration 20921 / 24500) loss: 0.978040
(Iteration 20931 / 24500) loss: 0.943463
(Iteration 20941 / 24500) loss: 1.027875
(Iteration 20951 / 24500) loss: 0.827853
(Iteration 20961 / 24500) loss: 0.919725
(Iteration 20971 / 24500) loss: 1.000108
(Iteration 20981 / 24500) loss: 1.069669
(Iteration 20991 / 24500) loss: 0.836317
(Iteration 21001 / 24500) loss: 0.929272
(Iteration 21011 / 24500) loss: 0.776055
(Iteration 21021 / 24500) loss: 0.922518
(Iteration 21031 / 24500) loss: 0.872813
(Iteration 21041 / 24500) loss: 0.889815
(Iteration 21051 / 24500) loss: 0.870176
(Iteration 21061 / 24500) loss: 0.901652
(Epoch 43 / 50) train acc: 0.655000; val_acc: 0.592000
(Iteration 21071 / 24500) loss: 1.010927
(Iteration 21081 / 24500) loss: 0.929597
(Iteration 21091 / 24500) loss: 0.962077
(Iteration 21101 / 24500) loss: 1.059618
(Iteration 21111 / 24500) loss: 0.917249
(Iteration 21121 / 24500) loss: 1.030217
(Iteration 21131 / 24500) loss: 0.977571
(Iteration 21141 / 24500) loss: 1.035782
(Iteration 21151 / 24500) loss: 1.094612
(Iteration 21161 / 24500) loss: 0.915830
(Iteration 21171 / 24500) loss: 0.874774
(Iteration 21181 / 24500) loss: 0.960398
(Iteration 21191 / 24500) loss: 0.968887
```

```
(Iteration 21201 / 24500) loss: 1.002724
(Iteration 21211 / 24500) loss: 0.957720
(Iteration 21221 / 24500) loss: 0.964950
(Iteration 21231 / 24500) loss: 0.920955
(Iteration 21241 / 24500) loss: 0.999345
(Iteration 21251 / 24500) loss: 0.750292
(Iteration 21261 / 24500) loss: 1.210630
(Iteration 21271 / 24500) loss: 0.891257
(Iteration 21281 / 24500) loss: 0.999017
(Iteration 21291 / 24500) loss: 1.097920
(Iteration 21301 / 24500) loss: 0.996855
(Iteration 21311 / 24500) loss: 1.100625
(Iteration 21321 / 24500) loss: 1.128473
(Iteration 21331 / 24500) loss: 0.821446
(Iteration 21341 / 24500) loss: 0.973382
(Iteration 21351 / 24500) loss: 0.933951
(Iteration 21361 / 24500) loss: 1.053327
(Iteration 21371 / 24500) loss: 0.884134
(Iteration 21381 / 24500) loss: 1.031473
(Iteration 21391 / 24500) loss: 1.097812
(Iteration 21401 / 24500) loss: 0.931286
(Iteration 21411 / 24500) loss: 1.100397
(Iteration 21421 / 24500) loss: 1.005926
(Iteration 21431 / 24500) loss: 0.883404
(Iteration 21441 / 24500) loss: 1.122384
(Iteration 21451 / 24500) loss: 1.025740
(Iteration 21461 / 24500) loss: 0.973076
(Iteration 21471 / 24500) loss: 0.984977
(Iteration 21481 / 24500) loss: 1.033315
(Iteration 21491 / 24500) loss: 1.015228
(Iteration 21501 / 24500) loss: 0.985526
(Iteration 21511 / 24500) loss: 0.962779
(Iteration 21521 / 24500) loss: 1.093814
(Iteration 21531 / 24500) loss: 0.940984
(Iteration 21541 / 24500) loss: 0.901276
(Iteration 21551 / 24500) loss: 0.948925
(Epoch 44 / 50) train acc: 0.689000; val_acc: 0.594000
(Iteration 21561 / 24500) loss: 0.974199
(Iteration 21571 / 24500) loss: 0.957114
(Iteration 21581 / 24500) loss: 0.990239
(Iteration 21591 / 24500) loss: 0.959355
(Iteration 21601 / 24500) loss: 0.927942
(Iteration 21611 / 24500) loss: 1.031689
(Iteration 21621 / 24500) loss: 0.944948
(Iteration 21631 / 24500) loss: 1.009932
(Iteration 21641 / 24500) loss: 1.020454
(Iteration 21651 / 24500) loss: 0.964761
(Iteration 21661 / 24500) loss: 1.204773
```

```
(Iteration 21671 / 24500) loss: 0.867234
(Iteration 21681 / 24500) loss: 0.985043
(Iteration 21691 / 24500) loss: 0.945430
(Iteration 21701 / 24500) loss: 0.853869
(Iteration 21711 / 24500) loss: 0.958344
(Iteration 21721 / 24500) loss: 0.886729
(Iteration 21731 / 24500) loss: 0.969466
(Iteration 21741 / 24500) loss: 1.124060
(Iteration 21751 / 24500) loss: 1.030869
(Iteration 21761 / 24500) loss: 0.944259
(Iteration 21771 / 24500) loss: 0.991640
(Iteration 21781 / 24500) loss: 1.010249
(Iteration 21791 / 24500) loss: 0.922710
(Iteration 21801 / 24500) loss: 1.109820
(Iteration 21811 / 24500) loss: 0.972393
(Iteration 21821 / 24500) loss: 1.128517
(Iteration 21831 / 24500) loss: 0.890227
(Iteration 21841 / 24500) loss: 0.989659
(Iteration 21851 / 24500) loss: 1.077442
(Iteration 21861 / 24500) loss: 1.015379
(Iteration 21871 / 24500) loss: 0.931746
(Iteration 21881 / 24500) loss: 0.966212
(Iteration 21891 / 24500) loss: 0.990503
(Iteration 21901 / 24500) loss: 0.986594
(Iteration 21911 / 24500) loss: 0.891500
(Iteration 21921 / 24500) loss: 1.033954
(Iteration 21931 / 24500) loss: 0.865583
(Iteration 21941 / 24500) loss: 1.093692
(Iteration 21951 / 24500) loss: 0.937072
(Iteration 21961 / 24500) loss: 1.064760
(Iteration 21971 / 24500) loss: 1.030160
(Iteration 21981 / 24500) loss: 1.121095
(Iteration 21991 / 24500) loss: 0.872314
(Iteration 22001 / 24500) loss: 0.915812
(Iteration 22011 / 24500) loss: 1.181260
(Iteration 22021 / 24500) loss: 1.030501
(Iteration 22031 / 24500) loss: 0.789195
(Iteration 22041 / 24500) loss: 1.083727
(Epoch 45 / 50) train acc: 0.658000; val_acc: 0.595000
(Iteration 22051 / 24500) loss: 1.094227
(Iteration 22061 / 24500) loss: 0.887907
(Iteration 22071 / 24500) loss: 0.860353
(Iteration 22081 / 24500) loss: 0.845149
(Iteration 22091 / 24500) loss: 1.067826
(Iteration 22101 / 24500) loss: 0.896271
(Iteration 22111 / 24500) loss: 0.989669
(Iteration 22121 / 24500) loss: 1.169271
(Iteration 22131 / 24500) loss: 0.970722
```

```
(Iteration 22141 / 24500) loss: 1.047891
(Iteration 22151 / 24500) loss: 0.960815
(Iteration 22161 / 24500) loss: 0.878441
(Iteration 22171 / 24500) loss: 0.909392
(Iteration 22181 / 24500) loss: 0.781696
(Iteration 22191 / 24500) loss: 0.868529
(Iteration 22201 / 24500) loss: 1.130569
(Iteration 22211 / 24500) loss: 0.924509
(Iteration 22221 / 24500) loss: 1.010564
(Iteration 22231 / 24500) loss: 0.954940
(Iteration 22241 / 24500) loss: 1.017957
(Iteration 22251 / 24500) loss: 1.006994
(Iteration 22261 / 24500) loss: 1.113496
(Iteration 22271 / 24500) loss: 1.078320
(Iteration 22281 / 24500) loss: 0.883981
(Iteration 22291 / 24500) loss: 0.891352
(Iteration 22301 / 24500) loss: 1.107846
(Iteration 22311 / 24500) loss: 1.021063
(Iteration 22321 / 24500) loss: 0.934870
(Iteration 22331 / 24500) loss: 0.932112
(Iteration 22341 / 24500) loss: 1.051727
(Iteration 22351 / 24500) loss: 0.810390
(Iteration 22361 / 24500) loss: 0.932630
(Iteration 22371 / 24500) loss: 1.074515
(Iteration 22381 / 24500) loss: 0.809206
(Iteration 22391 / 24500) loss: 0.936474
(Iteration 22401 / 24500) loss: 1.034340
(Iteration 22411 / 24500) loss: 0.865120
(Iteration 22421 / 24500) loss: 0.999008
(Iteration 22431 / 24500) loss: 1.002201
(Iteration 22441 / 24500) loss: 1.005671
(Iteration 22451 / 24500) loss: 1.018332
(Iteration 22461 / 24500) loss: 1.147095
(Iteration 22471 / 24500) loss: 1.043917
(Iteration 22481 / 24500) loss: 0.954340
(Iteration 22491 / 24500) loss: 0.963473
(Iteration 22501 / 24500) loss: 0.843946
(Iteration 22511 / 24500) loss: 1.031406
(Iteration 22521 / 24500) loss: 0.868738
(Iteration 22531 / 24500) loss: 0.995611
(Epoch 46 / 50) train acc: 0.667000; val_acc: 0.598000
(Iteration 22541 / 24500) loss: 1.006195
(Iteration 22551 / 24500) loss: 0.845546
(Iteration 22561 / 24500) loss: 1.006516
(Iteration 22571 / 24500) loss: 0.994555
(Iteration 22581 / 24500) loss: 0.883576
(Iteration 22591 / 24500) loss: 1.050793
(Iteration 22601 / 24500) loss: 1.045860
```

```
(Iteration 22611 / 24500) loss: 0.986414
(Iteration 22621 / 24500) loss: 0.982374
(Iteration 22631 / 24500) loss: 1.051964
(Iteration 22641 / 24500) loss: 1.008145
(Iteration 22651 / 24500) loss: 0.962597
(Iteration 22661 / 24500) loss: 0.789068
(Iteration 22671 / 24500) loss: 0.999848
(Iteration 22681 / 24500) loss: 0.860008
(Iteration 22691 / 24500) loss: 0.954085
(Iteration 22701 / 24500) loss: 1.087818
(Iteration 22711 / 24500) loss: 0.899737
(Iteration 22721 / 24500) loss: 0.890630
(Iteration 22731 / 24500) loss: 0.899192
(Iteration 22741 / 24500) loss: 0.975319
(Iteration 22751 / 24500) loss: 0.919111
(Iteration 22761 / 24500) loss: 0.894183
(Iteration 22771 / 24500) loss: 1.068536
(Iteration 22781 / 24500) loss: 0.872319
(Iteration 22791 / 24500) loss: 1.049886
(Iteration 22801 / 24500) loss: 0.949132
(Iteration 22811 / 24500) loss: 0.920496
(Iteration 22821 / 24500) loss: 0.923032
(Iteration 22831 / 24500) loss: 0.986813
(Iteration 22841 / 24500) loss: 0.835045
(Iteration 22851 / 24500) loss: 0.940386
(Iteration 22861 / 24500) loss: 0.886475
(Iteration 22871 / 24500) loss: 1.066959
(Iteration 22881 / 24500) loss: 0.760875
(Iteration 22891 / 24500) loss: 0.994251
(Iteration 22901 / 24500) loss: 1.109369
(Iteration 22911 / 24500) loss: 0.954486
(Iteration 22921 / 24500) loss: 1.073552
(Iteration 22931 / 24500) loss: 1.153204
(Iteration 22941 / 24500) loss: 0.866420
(Iteration 22951 / 24500) loss: 1.002095
(Iteration 22961 / 24500) loss: 1.074712
(Iteration 22971 / 24500) loss: 1.037429
(Iteration 22981 / 24500) loss: 0.752243
(Iteration 22991 / 24500) loss: 0.889469
(Iteration 23001 / 24500) loss: 0.914035
(Iteration 23011 / 24500) loss: 0.926586
(Iteration 23021 / 24500) loss: 0.904152
(Epoch 47 / 50) train acc: 0.682000; val_acc: 0.583000
(Iteration 23031 / 24500) loss: 0.834558
(Iteration 23041 / 24500) loss: 0.945627
(Iteration 23051 / 24500) loss: 1.046036
(Iteration 23061 / 24500) loss: 1.063996
(Iteration 23071 / 24500) loss: 0.932689
```

```
(Iteration 23081 / 24500) loss: 0.977048
(Iteration 23091 / 24500) loss: 0.844542
(Iteration 23101 / 24500) loss: 1.056167
(Iteration 23111 / 24500) loss: 0.954714
(Iteration 23121 / 24500) loss: 1.169695
(Iteration 23131 / 24500) loss: 0.970888
(Iteration 23141 / 24500) loss: 0.820931
(Iteration 23151 / 24500) loss: 0.990398
(Iteration 23161 / 24500) loss: 0.968155
(Iteration 23171 / 24500) loss: 0.710184
(Iteration 23181 / 24500) loss: 0.749788
(Iteration 23191 / 24500) loss: 0.887713
(Iteration 23201 / 24500) loss: 1.128392
(Iteration 23211 / 24500) loss: 0.863329
(Iteration 23221 / 24500) loss: 1.005426
(Iteration 23231 / 24500) loss: 0.973024
(Iteration 23241 / 24500) loss: 1.094144
(Iteration 23251 / 24500) loss: 1.054068
(Iteration 23261 / 24500) loss: 0.833476
(Iteration 23271 / 24500) loss: 0.940610
(Iteration 23281 / 24500) loss: 1.091966
(Iteration 23291 / 24500) loss: 1.018888
(Iteration 23301 / 24500) loss: 0.807684
(Iteration 23311 / 24500) loss: 0.995909
(Iteration 23321 / 24500) loss: 0.820496
(Iteration 23331 / 24500) loss: 0.908720
(Iteration 23341 / 24500) loss: 0.879650
(Iteration 23351 / 24500) loss: 1.157369
(Iteration 23361 / 24500) loss: 0.790268
(Iteration 23371 / 24500) loss: 1.008732
(Iteration 23381 / 24500) loss: 1.063757
(Iteration 23391 / 24500) loss: 1.044881
(Iteration 23401 / 24500) loss: 1.112282
(Iteration 23411 / 24500) loss: 0.886871
(Iteration 23421 / 24500) loss: 0.854007
(Iteration 23431 / 24500) loss: 0.978670
(Iteration 23441 / 24500) loss: 0.974854
(Iteration 23451 / 24500) loss: 1.245298
(Iteration 23461 / 24500) loss: 0.923776
(Iteration 23471 / 24500) loss: 1.122015
(Iteration 23481 / 24500) loss: 1.166000
(Iteration 23491 / 24500) loss: 0.996408
(Iteration 23501 / 24500) loss: 1.123567
(Iteration 23511 / 24500) loss: 0.886712
(Epoch 48 / 50) train acc: 0.674000; val_acc: 0.592000
(Iteration 23521 / 24500) loss: 0.906126
(Iteration 23531 / 24500) loss: 1.077524
(Iteration 23541 / 24500) loss: 0.789839
```

```
(Iteration 23551 / 24500) loss: 1.085838
(Iteration 23561 / 24500) loss: 1.064617
(Iteration 23571 / 24500) loss: 0.929616
(Iteration 23581 / 24500) loss: 1.025451
(Iteration 23591 / 24500) loss: 0.895074
(Iteration 23601 / 24500) loss: 0.839256
(Iteration 23611 / 24500) loss: 0.921096
(Iteration 23621 / 24500) loss: 0.934574
(Iteration 23631 / 24500) loss: 1.276957
(Iteration 23641 / 24500) loss: 1.046781
(Iteration 23651 / 24500) loss: 0.796333
(Iteration 23661 / 24500) loss: 1.053733
(Iteration 23671 / 24500) loss: 1.057865
(Iteration 23681 / 24500) loss: 0.945903
(Iteration 23691 / 24500) loss: 0.808632
(Iteration 23701 / 24500) loss: 1.072014
(Iteration 23711 / 24500) loss: 0.969743
(Iteration 23721 / 24500) loss: 1.019983
(Iteration 23731 / 24500) loss: 0.824423
(Iteration 23741 / 24500) loss: 1.012747
(Iteration 23751 / 24500) loss: 0.995643
(Iteration 23761 / 24500) loss: 1.000455
(Iteration 23771 / 24500) loss: 0.918550
(Iteration 23781 / 24500) loss: 1.045087
(Iteration 23791 / 24500) loss: 0.896526
(Iteration 23801 / 24500) loss: 1.178298
(Iteration 23811 / 24500) loss: 1.037316
(Iteration 23821 / 24500) loss: 0.895608
(Iteration 23831 / 24500) loss: 1.032016
(Iteration 23841 / 24500) loss: 1.009805
(Iteration 23851 / 24500) loss: 0.954429
(Iteration 23861 / 24500) loss: 1.051115
(Iteration 23871 / 24500) loss: 0.987674
(Iteration 23881 / 24500) loss: 1.081412
(Iteration 23891 / 24500) loss: 1.047136
(Iteration 23901 / 24500) loss: 0.808268
(Iteration 23911 / 24500) loss: 1.028771
(Iteration 23921 / 24500) loss: 0.955010
(Iteration 23931 / 24500) loss: 1.013855
(Iteration 23941 / 24500) loss: 1.060854
(Iteration 23951 / 24500) loss: 1.072843
(Iteration 23961 / 24500) loss: 0.996968
(Iteration 23971 / 24500) loss: 0.990787
(Iteration 23981 / 24500) loss: 0.876532
(Iteration 23991 / 24500) loss: 0.782761
(Iteration 24001 / 24500) loss: 0.904989
(Epoch 49 / 50) train acc: 0.678000; val_acc: 0.601000
(Iteration 24011 / 24500) loss: 0.954026
```

```
(Iteration 24021 / 24500) loss: 0.780643
(Iteration 24031 / 24500) loss: 1.159327
(Iteration 24041 / 24500) loss: 0.970595
(Iteration 24051 / 24500) loss: 0.937180
(Iteration 24061 / 24500) loss: 0.867183
(Iteration 24071 / 24500) loss: 1.019055
(Iteration 24081 / 24500) loss: 1.143080
(Iteration 24091 / 24500) loss: 0.868054
(Iteration 24101 / 24500) loss: 0.994531
(Iteration 24111 / 24500) loss: 0.983047
(Iteration 24121 / 24500) loss: 0.924359
(Iteration 24131 / 24500) loss: 0.963131
(Iteration 24141 / 24500) loss: 0.955451
(Iteration 24151 / 24500) loss: 0.813301
(Iteration 24161 / 24500) loss: 1.025651
(Iteration 24171 / 24500) loss: 0.850034
(Iteration 24181 / 24500) loss: 0.968455
(Iteration 24191 / 24500) loss: 1.009031
(Iteration 24201 / 24500) loss: 0.930690
(Iteration 24211 / 24500) loss: 1.024483
(Iteration 24221 / 24500) loss: 1.164491
(Iteration 24231 / 24500) loss: 0.905990
(Iteration 24241 / 24500) loss: 0.952442
(Iteration 24251 / 24500) loss: 1.038019
(Iteration 24261 / 24500) loss: 0.975215
(Iteration 24271 / 24500) loss: 0.857004
(Iteration 24281 / 24500) loss: 0.909197
(Iteration 24291 / 24500) loss: 0.763739
(Iteration 24301 / 24500) loss: 0.923629
(Iteration 24311 / 24500) loss: 1.008296
(Iteration 24321 / 24500) loss: 0.907039
(Iteration 24331 / 24500) loss: 0.863468
(Iteration 24341 / 24500) loss: 1.168463
(Iteration 24351 / 24500) loss: 1.028203
(Iteration 24361 / 24500) loss: 1.103561
(Iteration 24371 / 24500) loss: 0.960962
(Iteration 24381 / 24500) loss: 1.033564
(Iteration 24391 / 24500) loss: 0.969908
(Iteration 24401 / 24500) loss: 1.126981
(Iteration 24411 / 24500) loss: 1.003145
(Iteration 24421 / 24500) loss: 1.027633
(Iteration 24431 / 24500) loss: 0.924722
(Iteration 24441 / 24500) loss: 1.023211
(Iteration 24451 / 24500) loss: 0.860252
(Iteration 24461 / 24500) loss: 0.834182
(Iteration 24471 / 24500) loss: 0.983931
(Iteration 24481 / 24500) loss: 1.021018
(Iteration 24491 / 24500) loss: 1.018471
```

```
(Epoch 50 / 50) train acc: 0.662000; val_acc: 0.606000
```

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

```
0.589
```

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

```
best_two_layer_net_features.npy saved.
```

[ ]:
```
```

FullyConnectedNets

February 3, 2026

[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/$FOLDERNAME/cs231n/datasets/
# %cd /content/drive/My\ Drive/$FOLDERNAME
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

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

[2]:
```
!pip -q install -U ipython
```

```
                        0.0/621.4 kB
? eta -:--:--
                      621.4/621.4 kB
27.7 MB/s eta 0:00:00
                        0.0/1.6
MB ? eta -:--:--
                      1.6/1.6 MB 57.3
MB/s eta 0:00:00
```

```
                                0.0/85.4
kB ? eta -:--:--
                               85.4/85.4 kB 5.7
MB/s eta 0:00:00
ERROR: pip's dependency resolver does not currently take into account
all the packages that are installed. This behaviour is the source of the
following dependency conflicts.
google-colab 1.0.0 requires ipython==7.34.0, but you have ipython 9.9.0 which is
incompatible.
```

1 Multi-Layer Fully Connected Network

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

[ ]:
```
# 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 assignment, 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.

[4]:
```
# 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
```

[5]:
```
# 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.

[7]:
```
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])}")
```

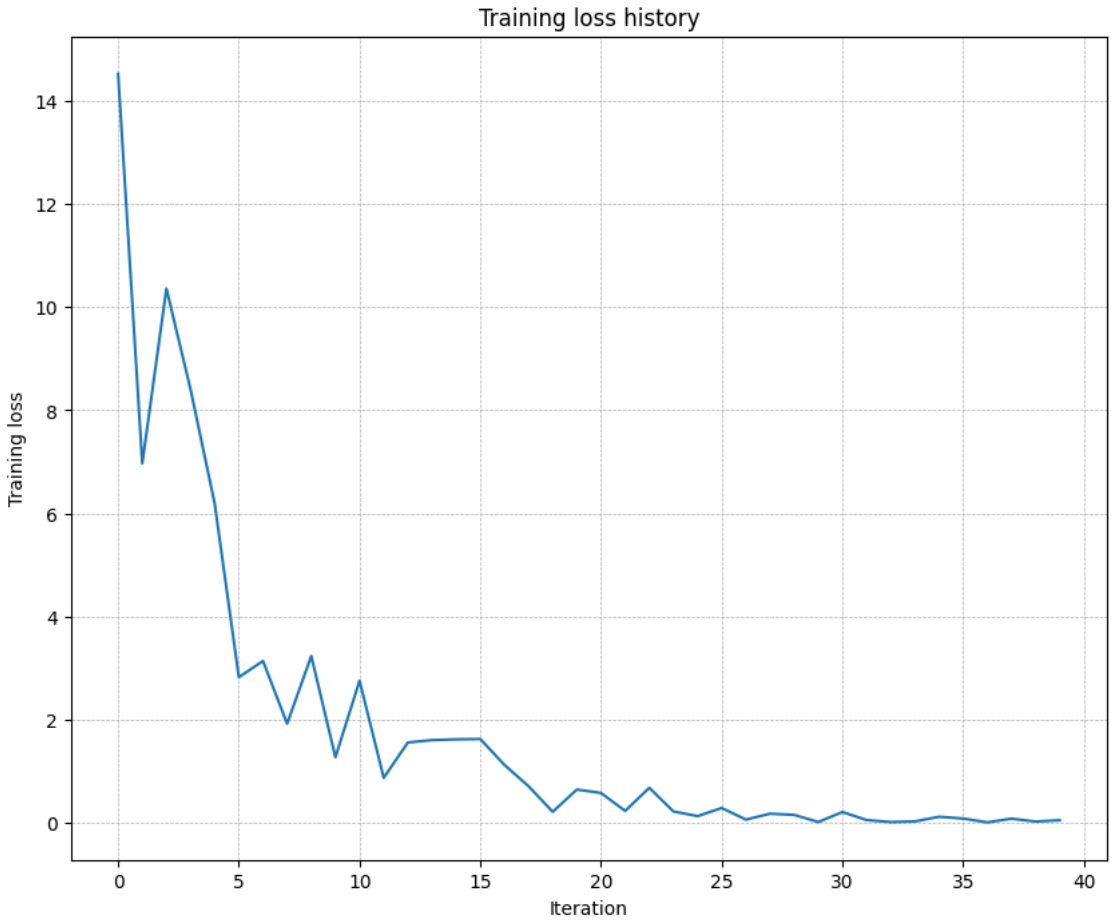
```
Running check with reg =  0
Initial loss:  2.300479089768492
W1 relative error: 1.0252674471656573e-07
W2 relative error: 2.2120479295080622e-05
W3 relative error: 4.5623278736665505e-07
b1 relative error: 4.6600944653202505e-09
b2 relative error: 2.085654276112763e-09
b3 relative error: 1.689724888469736e-10
Running check with reg =  3.14
Initial loss:  7.052114776533016
W1 relative error: 6.862884860440611e-09
W2 relative error: 3.522821562176466e-08
W3 relative error: 2.6171457283983532e-08
b1 relative error: 1.4752427965311745e-08
b2 relative error: 1.7223751746766738e-09
b3 relative error: 2.378772438198909e-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.

[15]:
```
# 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 = 4e-2   # Experiment with this!
learning_rate = 4e-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: 14.516118
(Epoch 0 / 20) train acc: 0.160000; val_acc: 0.089000
(Epoch 1 / 20) train acc: 0.240000; val_acc: 0.103000
(Epoch 2 / 20) train acc: 0.320000; val_acc: 0.104000
(Epoch 3 / 20) train acc: 0.540000; val_acc: 0.127000
(Epoch 4 / 20) train acc: 0.660000; val_acc: 0.114000
(Epoch 5 / 20) train acc: 0.620000; val_acc: 0.123000
(Iteration 11 / 40) loss: 2.760061
(Epoch 6 / 20) train acc: 0.720000; val_acc: 0.128000
(Epoch 7 / 20) train acc: 0.800000; val_acc: 0.127000
(Epoch 8 / 20) train acc: 0.800000; val_acc: 0.133000
(Epoch 9 / 20) train acc: 0.820000; val_acc: 0.148000
(Epoch 10 / 20) train acc: 0.920000; val_acc: 0.140000
(Iteration 21 / 40) loss: 0.585895
(Epoch 11 / 20) train acc: 0.940000; val_acc: 0.141000
(Epoch 12 / 20) train acc: 0.880000; val_acc: 0.143000
(Epoch 13 / 20) train acc: 0.960000; val_acc: 0.129000
(Epoch 14 / 20) train acc: 0.960000; val_acc: 0.140000
(Epoch 15 / 20) train acc: 0.960000; val_acc: 0.143000
(Iteration 31 / 40) loss: 0.216973
(Epoch 16 / 20) train acc: 0.980000; val_acc: 0.140000
(Epoch 17 / 20) train acc: 0.980000; val_acc: 0.141000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.141000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.139000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.141000
```



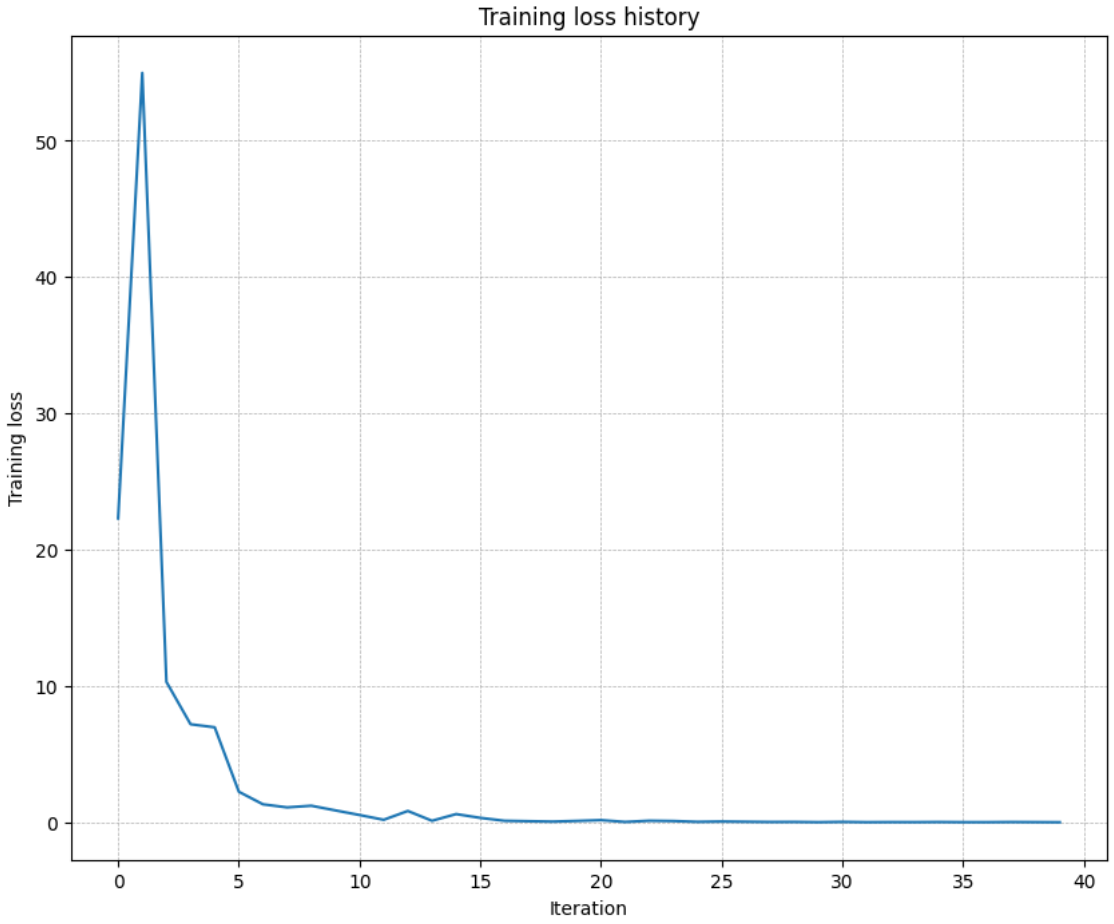
Now, try to use a five-layer network with 100 units on each layer to overfit on 50 training examples. Again, you will have to adjust the learning rate and weight initialization scale, but you should be able to achieve 100% training accuracy within 20 epochs.

[22]:
```
# 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 = 4e-3  # Experiment with this!
weight_scale = 7e-2   # 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: 22.298820
(Epoch 0 / 20) train acc: 0.140000; val_acc: 0.096000
(Epoch 1 / 20) train acc: 0.100000; val_acc: 0.103000
(Epoch 2 / 20) train acc: 0.400000; val_acc: 0.119000
(Epoch 3 / 20) train acc: 0.580000; val_acc: 0.124000
(Epoch 4 / 20) train acc: 0.780000; val_acc: 0.124000
(Epoch 5 / 20) train acc: 0.840000; val_acc: 0.119000
(Iteration 11 / 40) loss: 0.541687
(Epoch 6 / 20) train acc: 0.880000; val_acc: 0.116000
(Epoch 7 / 20) train acc: 0.960000; val_acc: 0.113000
(Epoch 8 / 20) train acc: 0.980000; val_acc: 0.102000
(Epoch 9 / 20) train acc: 0.980000; val_acc: 0.106000
(Epoch 10 / 20) train acc: 1.000000; val_acc: 0.106000
(Iteration 21 / 40) loss: 0.166612
(Epoch 11 / 20) train acc: 1.000000; val_acc: 0.104000
(Epoch 12 / 20) train acc: 1.000000; val_acc: 0.102000
(Epoch 13 / 20) train acc: 1.000000; val_acc: 0.100000
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.104000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.105000
(Iteration 31 / 40) loss: 0.037763
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.104000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.106000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.105000
```

```
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.108000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.109000
```



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:

[FILL THIS IN]

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.

[23]:
```
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
```

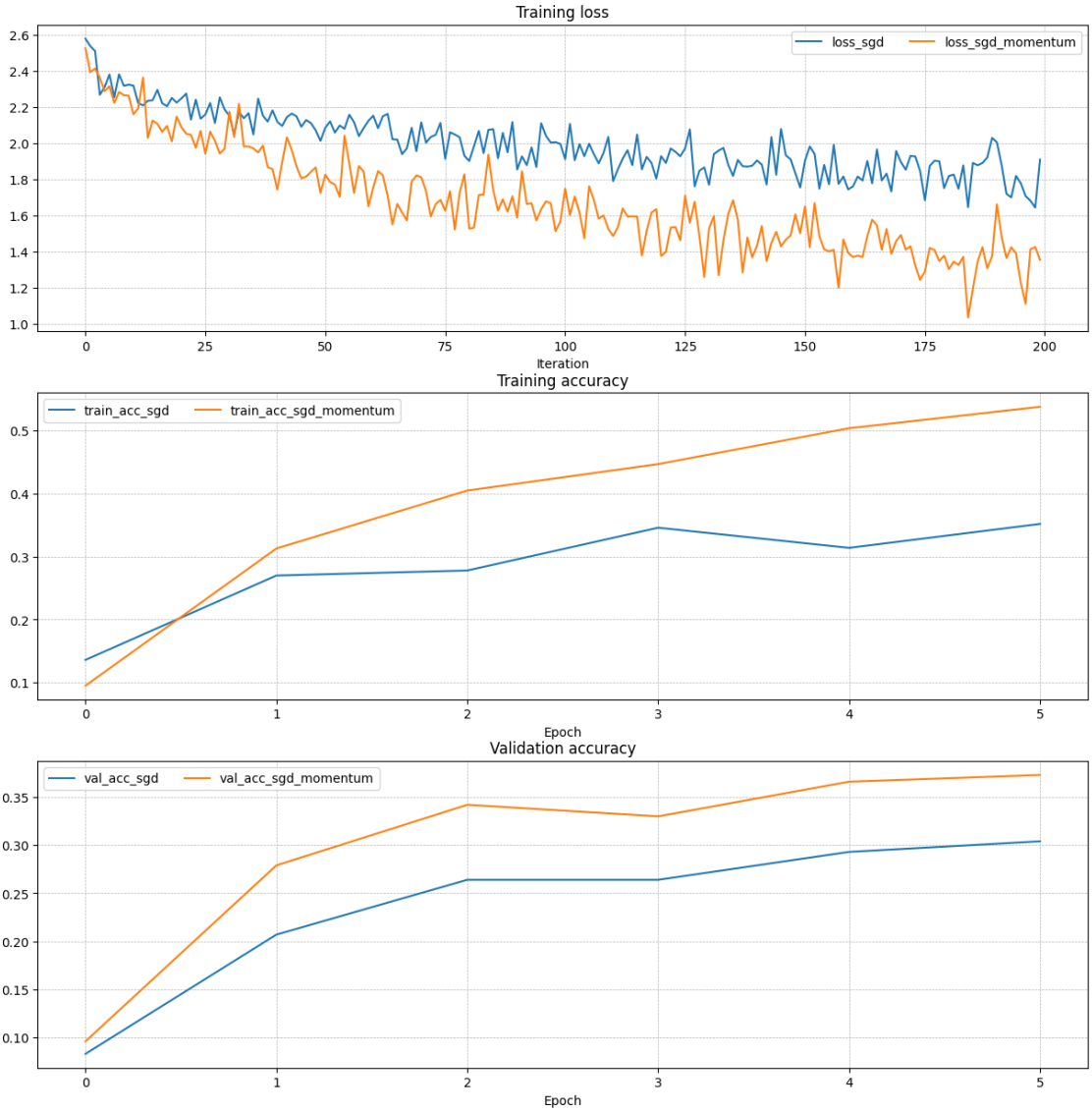
Once you have done so, run the following to train a six-layer network with both SGD and SGD+momentum. You should see the SGD+momentum update rule converge faster.

[24]:
```
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: 2.576831
(Epoch 0 / 5) train acc: 0.136000; val_acc: 0.083000
(Iteration 11 / 200) loss: 2.317323
(Iteration 21 / 200) loss: 2.246541
(Iteration 31 / 200) loss: 2.151031
```

```
(Epoch 1 / 5) train acc: 0.270000; val_acc: 0.207000
(Iteration 41 / 200) loss: 2.117219
(Iteration 51 / 200) loss: 2.084733
(Iteration 61 / 200) loss: 2.151344
(Iteration 71 / 200) loss: 2.114606
(Epoch 2 / 5) train acc: 0.278000; val_acc: 0.264000
(Iteration 81 / 200) loss: 1.900981
(Iteration 91 / 200) loss: 1.853521
(Iteration 101 / 200) loss: 1.911379
(Iteration 111 / 200) loss: 1.789633
(Epoch 3 / 5) train acc: 0.346000; val_acc: 0.264000
(Iteration 121 / 200) loss: 1.927723
(Iteration 131 / 200) loss: 1.769483
(Iteration 141 / 200) loss: 1.904034
(Iteration 151 / 200) loss: 1.902717
(Epoch 4 / 5) train acc: 0.314000; val_acc: 0.293000
(Iteration 161 / 200) loss: 1.760488
(Iteration 171 / 200) loss: 1.895027
(Iteration 181 / 200) loss: 1.818108
(Iteration 191 / 200) loss: 2.004564
(Epoch 5 / 5) train acc: 0.352000; val_acc: 0.304000
Running with  sgd_momentum
(Iteration 1 / 200) loss: 2.524473
(Epoch 0 / 5) train acc: 0.095000; val_acc: 0.096000
(Iteration 11 / 200) loss: 2.158811
(Iteration 21 / 200) loss: 2.088423
(Iteration 31 / 200) loss: 2.172682
(Epoch 1 / 5) train acc: 0.313000; val_acc: 0.279000
(Iteration 41 / 200) loss: 1.742422
(Iteration 51 / 200) loss: 1.825751
(Iteration 61 / 200) loss: 1.754791
(Iteration 71 / 200) loss: 1.810993
(Epoch 2 / 5) train acc: 0.405000; val_acc: 0.342000
(Iteration 81 / 200) loss: 1.526788
(Iteration 91 / 200) loss: 1.588004
(Iteration 101 / 200) loss: 1.747597
(Iteration 111 / 200) loss: 1.486469
(Epoch 3 / 5) train acc: 0.447000; val_acc: 0.330000
(Iteration 121 / 200) loss: 1.376262
(Iteration 131 / 200) loss: 1.527502
(Iteration 141 / 200) loss: 1.432711
(Iteration 151 / 200) loss: 1.649150
(Epoch 4 / 5) train acc: 0.504000; val_acc: 0.366000
(Iteration 161 / 200) loss: 1.370059
(Iteration 171 / 200) loss: 1.491510
(Iteration 181 / 200) loss: 1.303253
(Iteration 191 / 200) loss: 1.661749
(Epoch 5 / 5) train acc: 0.538000; val_acc: 0.373000
```



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.

[25]:
```
# 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
```

[26]:
```
# 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:  1.1395691798535431e-07
v error:  4.208314038113071e-09
m error:  4.214963193114416e-09
```

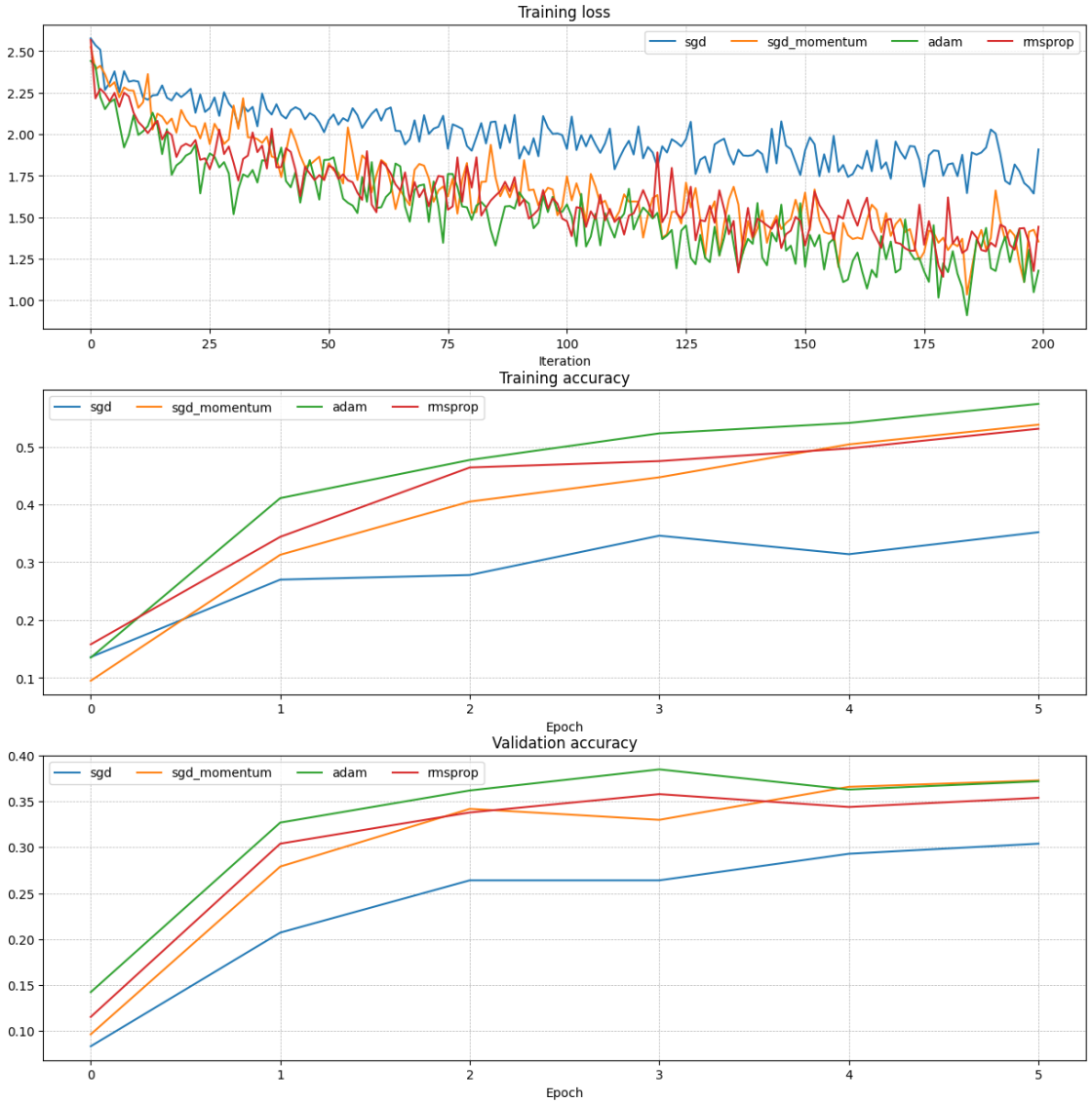
Once you have debugged your RMSProp and Adam implementations, run the following to train a pair of deep networks using these new update rules:

[27]:
```
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: 2.441300
(Epoch 0 / 5) train acc: 0.135000; val_acc: 0.142000
(Iteration 11 / 200) loss: 1.996038
(Iteration 21 / 200) loss: 1.872091
(Iteration 31 / 200) loss: 1.519262
(Epoch 1 / 5) train acc: 0.411000; val_acc: 0.327000
(Iteration 41 / 200) loss: 1.919909
(Iteration 51 / 200) loss: 1.847204
(Iteration 61 / 200) loss: 1.557118
(Iteration 71 / 200) loss: 1.697801
(Epoch 2 / 5) train acc: 0.477000; val_acc: 0.362000
(Iteration 81 / 200) loss: 1.485011
(Iteration 91 / 200) loss: 1.651037
(Iteration 101 / 200) loss: 1.577964
(Iteration 111 / 200) loss: 1.378397
(Epoch 3 / 5) train acc: 0.523000; val_acc: 0.385000
(Iteration 121 / 200) loss: 1.369252
(Iteration 131 / 200) loss: 1.230648
(Iteration 141 / 200) loss: 1.586313
(Iteration 151 / 200) loss: 1.202789
(Epoch 4 / 5) train acc: 0.541000; val_acc: 0.363000
(Iteration 161 / 200) loss: 1.237655
(Iteration 171 / 200) loss: 1.188742
(Iteration 181 / 200) loss: 1.170731
(Iteration 191 / 200) loss: 1.177620
(Epoch 5 / 5) train acc: 0.574000; val_acc: 0.372000

Running with  rmsprop
(Iteration 1 / 200) loss: 2.567900
(Epoch 0 / 5) train acc: 0.158000; val_acc: 0.115000
(Iteration 11 / 200) loss: 2.074414
```

```
(Iteration 21 / 200) loss: 1.942859
(Iteration 31 / 200) loss: 1.828746
(Epoch 1 / 5) train acc: 0.344000; val_acc: 0.304000
(Iteration 41 / 200) loss: 1.802791
(Iteration 51 / 200) loss: 1.810409
(Iteration 61 / 200) loss: 1.530768
(Iteration 71 / 200) loss: 1.670891
(Epoch 2 / 5) train acc: 0.464000; val_acc: 0.338000
(Iteration 81 / 200) loss: 1.678600
(Iteration 91 / 200) loss: 1.570821
(Iteration 101 / 200) loss: 1.477022
(Iteration 111 / 200) loss: 1.472715
(Epoch 3 / 5) train acc: 0.475000; val_acc: 0.358000
(Iteration 121 / 200) loss: 1.470870
(Iteration 131 / 200) loss: 1.569875
(Iteration 141 / 200) loss: 1.492717
(Iteration 151 / 200) loss: 1.330635
(Epoch 4 / 5) train acc: 0.497000; val_acc: 0.344000
(Iteration 161 / 200) loss: 1.518674
(Iteration 171 / 200) loss: 1.341464
(Iteration 181 / 200) loss: 1.620243
(Iteration 191 / 200) loss: 1.324073
(Epoch 5 / 5) train acc: 0.531000; val_acc: 0.354000
```



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:

[FILL THIS IN]

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.

[34]:
```
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)*****

model = FullyConnectedNet(
    [100, 100],
    weight_scale=1e-2,
    reg=1e-3
)
solver = Solver(
    model,
    data,
    num_epochs=20,
    batch_size=100,
    update_rule='adam',
    optim_config={'learning_rate': 1e-3},
    verbose=True
)
solver.train()

best_model = model


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

```
(Iteration 1 / 9800) loss: 2.351523
(Epoch 0 / 20) train acc: 0.166000; val_acc: 0.148000
```

```
(Iteration 11 / 9800) loss: 2.037272
(Iteration 21 / 9800) loss: 1.992362
(Iteration 31 / 9800) loss: 1.914466
(Iteration 41 / 9800) loss: 1.744384
(Iteration 51 / 9800) loss: 1.832170
(Iteration 61 / 9800) loss: 1.863656
(Iteration 71 / 9800) loss: 1.787718
(Iteration 81 / 9800) loss: 1.727013
(Iteration 91 / 9800) loss: 1.692949
(Iteration 101 / 9800) loss: 1.901322
(Iteration 111 / 9800) loss: 1.728736
(Iteration 121 / 9800) loss: 1.627161
(Iteration 131 / 9800) loss: 1.669387
(Iteration 141 / 9800) loss: 1.596757
(Iteration 151 / 9800) loss: 1.441296
(Iteration 161 / 9800) loss: 1.691854
(Iteration 171 / 9800) loss: 1.683299
(Iteration 181 / 9800) loss: 1.842529
(Iteration 191 / 9800) loss: 1.667980
(Iteration 201 / 9800) loss: 1.661979
(Iteration 211 / 9800) loss: 1.561174
(Iteration 221 / 9800) loss: 1.796666
(Iteration 231 / 9800) loss: 2.026525
(Iteration 241 / 9800) loss: 1.833808
(Iteration 251 / 9800) loss: 1.560072
(Iteration 261 / 9800) loss: 1.537095
(Iteration 271 / 9800) loss: 1.635772
(Iteration 281 / 9800) loss: 1.817692
(Iteration 291 / 9800) loss: 1.539828
(Iteration 301 / 9800) loss: 1.751740
(Iteration 311 / 9800) loss: 1.604604
(Iteration 321 / 9800) loss: 1.680345
(Iteration 331 / 9800) loss: 1.696387
(Iteration 341 / 9800) loss: 1.779914
(Iteration 351 / 9800) loss: 1.738684
(Iteration 361 / 9800) loss: 1.508773
(Iteration 371 / 9800) loss: 1.798692
(Iteration 381 / 9800) loss: 1.843585
(Iteration 391 / 9800) loss: 1.734888
(Iteration 401 / 9800) loss: 1.660123
(Iteration 411 / 9800) loss: 1.828536
(Iteration 421 / 9800) loss: 1.750412
(Iteration 431 / 9800) loss: 1.628551
(Iteration 441 / 9800) loss: 1.620598
(Iteration 451 / 9800) loss: 1.557729
(Iteration 461 / 9800) loss: 1.735471
(Iteration 471 / 9800) loss: 1.646725
(Iteration 481 / 9800) loss: 1.652919
```

```
(Epoch 1 / 20) train acc: 0.456000; val_acc: 0.420000
(Iteration 491 / 9800) loss: 1.370107
(Iteration 501 / 9800) loss: 1.869789
(Iteration 511 / 9800) loss: 1.829618
(Iteration 521 / 9800) loss: 1.570901
(Iteration 531 / 9800) loss: 1.781365
(Iteration 541 / 9800) loss: 1.598264
(Iteration 551 / 9800) loss: 1.586572
(Iteration 561 / 9800) loss: 1.751210
(Iteration 571 / 9800) loss: 1.610640
(Iteration 581 / 9800) loss: 1.776170
(Iteration 591 / 9800) loss: 1.804493
(Iteration 601 / 9800) loss: 1.900805
(Iteration 611 / 9800) loss: 1.701503
(Iteration 621 / 9800) loss: 1.562425
(Iteration 631 / 9800) loss: 1.660580
(Iteration 641 / 9800) loss: 1.905722
(Iteration 651 / 9800) loss: 1.557917
(Iteration 661 / 9800) loss: 1.803876
(Iteration 671 / 9800) loss: 1.786783
(Iteration 681 / 9800) loss: 1.535483
(Iteration 691 / 9800) loss: 1.542364
(Iteration 701 / 9800) loss: 1.673252
(Iteration 711 / 9800) loss: 1.757053
(Iteration 721 / 9800) loss: 1.651885
(Iteration 731 / 9800) loss: 1.563210
(Iteration 741 / 9800) loss: 1.816457
(Iteration 751 / 9800) loss: 1.512118
(Iteration 761 / 9800) loss: 1.685284
(Iteration 771 / 9800) loss: 1.651988
(Iteration 781 / 9800) loss: 1.435018
(Iteration 791 / 9800) loss: 1.435064
(Iteration 801 / 9800) loss: 1.515999
(Iteration 811 / 9800) loss: 1.548277
(Iteration 821 / 9800) loss: 1.694358
(Iteration 831 / 9800) loss: 1.667586
(Iteration 841 / 9800) loss: 1.559241
(Iteration 851 / 9800) loss: 1.557093
(Iteration 861 / 9800) loss: 1.535857
(Iteration 871 / 9800) loss: 1.565386
(Iteration 881 / 9800) loss: 1.446615
(Iteration 891 / 9800) loss: 1.570549
(Iteration 901 / 9800) loss: 1.530649
(Iteration 911 / 9800) loss: 1.684882
(Iteration 921 / 9800) loss: 1.741265
(Iteration 931 / 9800) loss: 1.502188
(Iteration 941 / 9800) loss: 1.583268
(Iteration 951 / 9800) loss: 1.404246
```

```
(Iteration 961 / 9800) loss: 1.628070
(Iteration 971 / 9800) loss: 1.618095
(Epoch 2 / 20) train acc: 0.454000; val_acc: 0.449000
(Iteration 981 / 9800) loss: 1.826326
(Iteration 991 / 9800) loss: 1.469442
(Iteration 1001 / 9800) loss: 1.664503
(Iteration 1011 / 9800) loss: 1.643646
(Iteration 1021 / 9800) loss: 1.677737
(Iteration 1031 / 9800) loss: 1.725351
(Iteration 1041 / 9800) loss: 1.390619
(Iteration 1051 / 9800) loss: 1.422764
(Iteration 1061 / 9800) loss: 1.581725
(Iteration 1071 / 9800) loss: 1.690440
(Iteration 1081 / 9800) loss: 1.583127
(Iteration 1091 / 9800) loss: 1.515659
(Iteration 1101 / 9800) loss: 1.510582
(Iteration 1111 / 9800) loss: 1.746585
(Iteration 1121 / 9800) loss: 1.637699
(Iteration 1131 / 9800) loss: 1.373276
(Iteration 1141 / 9800) loss: 1.576935
(Iteration 1151 / 9800) loss: 1.608252
(Iteration 1161 / 9800) loss: 1.425272
(Iteration 1171 / 9800) loss: 1.553534
(Iteration 1181 / 9800) loss: 1.437127
(Iteration 1191 / 9800) loss: 1.618157
(Iteration 1201 / 9800) loss: 1.653258
(Iteration 1211 / 9800) loss: 1.497887
(Iteration 1221 / 9800) loss: 1.398385
(Iteration 1231 / 9800) loss: 1.618917
(Iteration 1241 / 9800) loss: 1.524493
(Iteration 1251 / 9800) loss: 1.459025
(Iteration 1261 / 9800) loss: 1.594800
(Iteration 1271 / 9800) loss: 1.520178
(Iteration 1281 / 9800) loss: 1.604202
(Iteration 1291 / 9800) loss: 1.554538
(Iteration 1301 / 9800) loss: 1.647717
(Iteration 1311 / 9800) loss: 1.547966
(Iteration 1321 / 9800) loss: 1.508802
(Iteration 1331 / 9800) loss: 1.649882
(Iteration 1341 / 9800) loss: 1.772500
(Iteration 1351 / 9800) loss: 1.549050
(Iteration 1361 / 9800) loss: 1.614081
(Iteration 1371 / 9800) loss: 1.590640
(Iteration 1381 / 9800) loss: 1.558985
(Iteration 1391 / 9800) loss: 1.394368
(Iteration 1401 / 9800) loss: 1.359291
(Iteration 1411 / 9800) loss: 2.013854
(Iteration 1421 / 9800) loss: 1.506824
```

```
(Iteration 1431 / 9800) loss: 1.735413
(Iteration 1441 / 9800) loss: 1.469241
(Iteration 1451 / 9800) loss: 1.414111
(Iteration 1461 / 9800) loss: 1.393158
(Epoch 3 / 20) train acc: 0.453000; val_acc: 0.437000
(Iteration 1471 / 9800) loss: 1.501004
(Iteration 1481 / 9800) loss: 1.584716
(Iteration 1491 / 9800) loss: 1.373167
(Iteration 1501 / 9800) loss: 1.530010
(Iteration 1511 / 9800) loss: 1.599978
(Iteration 1521 / 9800) loss: 1.597697
(Iteration 1531 / 9800) loss: 1.670274
(Iteration 1541 / 9800) loss: 1.705155
(Iteration 1551 / 9800) loss: 1.595101
(Iteration 1561 / 9800) loss: 1.611113
(Iteration 1571 / 9800) loss: 1.473739
(Iteration 1581 / 9800) loss: 1.492200
(Iteration 1591 / 9800) loss: 1.563889
(Iteration 1601 / 9800) loss: 1.431841
(Iteration 1611 / 9800) loss: 1.701735
(Iteration 1621 / 9800) loss: 1.655145
(Iteration 1631 / 9800) loss: 1.610999
(Iteration 1641 / 9800) loss: 1.695816
(Iteration 1651 / 9800) loss: 1.384341
(Iteration 1661 / 9800) loss: 1.511661
(Iteration 1671 / 9800) loss: 1.683427
(Iteration 1681 / 9800) loss: 1.458215
(Iteration 1691 / 9800) loss: 1.598441
(Iteration 1701 / 9800) loss: 1.358948
(Iteration 1711 / 9800) loss: 1.437827
(Iteration 1721 / 9800) loss: 1.577698
(Iteration 1731 / 9800) loss: 1.388009
(Iteration 1741 / 9800) loss: 1.303368
(Iteration 1751 / 9800) loss: 1.534651
(Iteration 1761 / 9800) loss: 1.605842
(Iteration 1771 / 9800) loss: 1.454320
(Iteration 1781 / 9800) loss: 1.534691
(Iteration 1791 / 9800) loss: 1.509491
(Iteration 1801 / 9800) loss: 1.626276
(Iteration 1811 / 9800) loss: 1.426476
(Iteration 1821 / 9800) loss: 1.657605
(Iteration 1831 / 9800) loss: 1.502699
(Iteration 1841 / 9800) loss: 1.362436
(Iteration 1851 / 9800) loss: 1.612098
(Iteration 1861 / 9800) loss: 1.685364
(Iteration 1871 / 9800) loss: 1.450791
(Iteration 1881 / 9800) loss: 1.729251
(Iteration 1891 / 9800) loss: 1.537035
```

```
(Iteration 1901 / 9800) loss: 1.721008
(Iteration 1911 / 9800) loss: 1.473980
(Iteration 1921 / 9800) loss: 1.349105
(Iteration 1931 / 9800) loss: 1.303560
(Iteration 1941 / 9800) loss: 1.598018
(Iteration 1951 / 9800) loss: 1.884374
(Epoch 4 / 20) train acc: 0.471000; val_acc: 0.460000
(Iteration 1961 / 9800) loss: 1.434475
(Iteration 1971 / 9800) loss: 1.632057
(Iteration 1981 / 9800) loss: 1.457135
(Iteration 1991 / 9800) loss: 1.646991
(Iteration 2001 / 9800) loss: 1.710491
(Iteration 2011 / 9800) loss: 1.481138
(Iteration 2021 / 9800) loss: 1.456058
(Iteration 2031 / 9800) loss: 1.404150
(Iteration 2041 / 9800) loss: 1.657377
(Iteration 2051 / 9800) loss: 1.364648
(Iteration 2061 / 9800) loss: 1.633620
(Iteration 2071 / 9800) loss: 1.518522
(Iteration 2081 / 9800) loss: 1.614360
(Iteration 2091 / 9800) loss: 1.590014
(Iteration 2101 / 9800) loss: 1.716510
(Iteration 2111 / 9800) loss: 1.427444
(Iteration 2121 / 9800) loss: 1.567925
(Iteration 2131 / 9800) loss: 1.587693
(Iteration 2141 / 9800) loss: 1.546268
(Iteration 2151 / 9800) loss: 1.557700
(Iteration 2161 / 9800) loss: 1.319207
(Iteration 2171 / 9800) loss: 1.492978
(Iteration 2181 / 9800) loss: 1.782602
(Iteration 2191 / 9800) loss: 1.535560
(Iteration 2201 / 9800) loss: 1.560633
(Iteration 2211 / 9800) loss: 1.582380
(Iteration 2221 / 9800) loss: 1.509830
(Iteration 2231 / 9800) loss: 1.483854
(Iteration 2241 / 9800) loss: 1.716010
(Iteration 2251 / 9800) loss: 1.450506
(Iteration 2261 / 9800) loss: 1.527102
(Iteration 2271 / 9800) loss: 1.427153
(Iteration 2281 / 9800) loss: 1.456745
(Iteration 2291 / 9800) loss: 1.432583
(Iteration 2301 / 9800) loss: 1.893346
(Iteration 2311 / 9800) loss: 1.690767
(Iteration 2321 / 9800) loss: 1.622424
(Iteration 2331 / 9800) loss: 1.349697
(Iteration 2341 / 9800) loss: 1.457128
(Iteration 2351 / 9800) loss: 1.482689
(Iteration 2361 / 9800) loss: 1.387805
```

```
(Iteration 2371 / 9800) loss: 1.532053
(Iteration 2381 / 9800) loss: 1.513698
(Iteration 2391 / 9800) loss: 1.467122
(Iteration 2401 / 9800) loss: 1.643177
(Iteration 2411 / 9800) loss: 1.440269
(Iteration 2421 / 9800) loss: 1.605676
(Iteration 2431 / 9800) loss: 1.613243
(Iteration 2441 / 9800) loss: 1.759586
(Epoch 5 / 20) train acc: 0.502000; val_acc: 0.467000
(Iteration 2451 / 9800) loss: 2.003325
(Iteration 2461 / 9800) loss: 1.690529
(Iteration 2471 / 9800) loss: 1.521266
(Iteration 2481 / 9800) loss: 1.826848
(Iteration 2491 / 9800) loss: 1.572516
(Iteration 2501 / 9800) loss: 1.442833
(Iteration 2511 / 9800) loss: 1.637181
(Iteration 2521 / 9800) loss: 1.621761
(Iteration 2531 / 9800) loss: 1.847959
(Iteration 2541 / 9800) loss: 1.456064
(Iteration 2551 / 9800) loss: 1.577065
(Iteration 2561 / 9800) loss: 1.534172
(Iteration 2571 / 9800) loss: 1.364422
(Iteration 2581 / 9800) loss: 1.515700
(Iteration 2591 / 9800) loss: 1.647626
(Iteration 2601 / 9800) loss: 1.504406
(Iteration 2611 / 9800) loss: 1.424410
(Iteration 2621 / 9800) loss: 1.471005
(Iteration 2631 / 9800) loss: 1.593346
(Iteration 2641 / 9800) loss: 1.475415
(Iteration 2651 / 9800) loss: 1.538010
(Iteration 2661 / 9800) loss: 1.520248
(Iteration 2671 / 9800) loss: 1.554219
(Iteration 2681 / 9800) loss: 1.603098
(Iteration 2691 / 9800) loss: 1.313210
(Iteration 2701 / 9800) loss: 1.470614
(Iteration 2711 / 9800) loss: 1.638999
(Iteration 2721 / 9800) loss: 1.823385
(Iteration 2731 / 9800) loss: 1.476795
(Iteration 2741 / 9800) loss: 1.508448
(Iteration 2751 / 9800) loss: 1.422711
(Iteration 2761 / 9800) loss: 1.442429
(Iteration 2771 / 9800) loss: 1.403757
(Iteration 2781 / 9800) loss: 1.539232
(Iteration 2791 / 9800) loss: 1.453793
(Iteration 2801 / 9800) loss: 1.479042
(Iteration 2811 / 9800) loss: 1.438349
(Iteration 2821 / 9800) loss: 1.493979
(Iteration 2831 / 9800) loss: 1.339909
```

```
(Iteration 2841 / 9800) loss: 1.491637
(Iteration 2851 / 9800) loss: 1.552372
(Iteration 2861 / 9800) loss: 1.672683
(Iteration 2871 / 9800) loss: 1.530515
(Iteration 2881 / 9800) loss: 1.526793
(Iteration 2891 / 9800) loss: 1.664991
(Iteration 2901 / 9800) loss: 1.674134
(Iteration 2911 / 9800) loss: 1.585840
(Iteration 2921 / 9800) loss: 1.528777
(Iteration 2931 / 9800) loss: 1.692830
(Epoch 6 / 20) train acc: 0.511000; val_acc: 0.492000
(Iteration 2941 / 9800) loss: 1.333696
(Iteration 2951 / 9800) loss: 1.382740
(Iteration 2961 / 9800) loss: 1.455284
(Iteration 2971 / 9800) loss: 1.463580
(Iteration 2981 / 9800) loss: 1.422194
(Iteration 2991 / 9800) loss: 1.647625
(Iteration 3001 / 9800) loss: 1.489058
(Iteration 3011 / 9800) loss: 1.367272
(Iteration 3021 / 9800) loss: 1.427893
(Iteration 3031 / 9800) loss: 1.469229
(Iteration 3041 / 9800) loss: 1.482496
(Iteration 3051 / 9800) loss: 1.497225
(Iteration 3061 / 9800) loss: 1.664655
(Iteration 3071 / 9800) loss: 1.418491
(Iteration 3081 / 9800) loss: 1.436584
(Iteration 3091 / 9800) loss: 1.552656
(Iteration 3101 / 9800) loss: 1.485860
(Iteration 3111 / 9800) loss: 1.554221
(Iteration 3121 / 9800) loss: 1.611537
(Iteration 3131 / 9800) loss: 1.438060
(Iteration 3141 / 9800) loss: 1.660174
(Iteration 3151 / 9800) loss: 1.616682
(Iteration 3161 / 9800) loss: 1.615640
(Iteration 3171 / 9800) loss: 1.434198
(Iteration 3181 / 9800) loss: 1.762572
(Iteration 3191 / 9800) loss: 1.629507
(Iteration 3201 / 9800) loss: 1.544435
(Iteration 3211 / 9800) loss: 1.697633
(Iteration 3221 / 9800) loss: 1.582714
(Iteration 3231 / 9800) loss: 1.468654
(Iteration 3241 / 9800) loss: 1.360931
(Iteration 3251 / 9800) loss: 1.541853
(Iteration 3261 / 9800) loss: 1.430238
(Iteration 3271 / 9800) loss: 1.349767
(Iteration 3281 / 9800) loss: 1.512732
(Iteration 3291 / 9800) loss: 1.528898
(Iteration 3301 / 9800) loss: 1.462601
```

```
(Iteration 3311 / 9800) loss: 1.648595
(Iteration 3321 / 9800) loss: 1.499418
(Iteration 3331 / 9800) loss: 1.404068
(Iteration 3341 / 9800) loss: 1.559388
(Iteration 3351 / 9800) loss: 1.449792
(Iteration 3361 / 9800) loss: 1.738178
(Iteration 3371 / 9800) loss: 1.486988
(Iteration 3381 / 9800) loss: 1.736909
(Iteration 3391 / 9800) loss: 1.554045
(Iteration 3401 / 9800) loss: 1.499752
(Iteration 3411 / 9800) loss: 1.441145
(Iteration 3421 / 9800) loss: 1.545015
(Epoch 7 / 20) train acc: 0.539000; val_acc: 0.465000
(Iteration 3431 / 9800) loss: 1.696986
(Iteration 3441 / 9800) loss: 1.531764
(Iteration 3451 / 9800) loss: 1.530834
(Iteration 3461 / 9800) loss: 1.390618
(Iteration 3471 / 9800) loss: 1.527041
(Iteration 3481 / 9800) loss: 1.491361
(Iteration 3491 / 9800) loss: 1.497140
(Iteration 3501 / 9800) loss: 1.451717
(Iteration 3511 / 9800) loss: 1.495753
(Iteration 3521 / 9800) loss: 1.391241
(Iteration 3531 / 9800) loss: 1.453332
(Iteration 3541 / 9800) loss: 1.534321
(Iteration 3551 / 9800) loss: 1.322531
(Iteration 3561 / 9800) loss: 1.518433
(Iteration 3571 / 9800) loss: 1.389161
(Iteration 3581 / 9800) loss: 1.528467
(Iteration 3591 / 9800) loss: 1.372166
(Iteration 3601 / 9800) loss: 1.672007
(Iteration 3611 / 9800) loss: 1.223237
(Iteration 3621 / 9800) loss: 1.637575
(Iteration 3631 / 9800) loss: 1.537446
(Iteration 3641 / 9800) loss: 1.631127
(Iteration 3651 / 9800) loss: 1.477498
(Iteration 3661 / 9800) loss: 1.402028
(Iteration 3671 / 9800) loss: 1.382691
(Iteration 3681 / 9800) loss: 1.385972
(Iteration 3691 / 9800) loss: 1.447316
(Iteration 3701 / 9800) loss: 1.529404
(Iteration 3711 / 9800) loss: 1.519317
(Iteration 3721 / 9800) loss: 1.732351
(Iteration 3731 / 9800) loss: 1.461846
(Iteration 3741 / 9800) loss: 1.503543
(Iteration 3751 / 9800) loss: 1.430760
(Iteration 3761 / 9800) loss: 1.472614
(Iteration 3771 / 9800) loss: 1.230966
```

```
(Iteration 3781 / 9800) loss: 1.563874
(Iteration 3791 / 9800) loss: 1.633009
(Iteration 3801 / 9800) loss: 1.584100
(Iteration 3811 / 9800) loss: 1.612893
(Iteration 3821 / 9800) loss: 1.547969
(Iteration 3831 / 9800) loss: 1.426047
(Iteration 3841 / 9800) loss: 1.402538
(Iteration 3851 / 9800) loss: 1.478492
(Iteration 3861 / 9800) loss: 1.555995
(Iteration 3871 / 9800) loss: 1.597954
(Iteration 3881 / 9800) loss: 1.352698
(Iteration 3891 / 9800) loss: 1.567664
(Iteration 3901 / 9800) loss: 1.426464
(Iteration 3911 / 9800) loss: 1.372809
(Epoch 8 / 20) train acc: 0.525000; val_acc: 0.482000
(Iteration 3921 / 9800) loss: 1.497799
(Iteration 3931 / 9800) loss: 1.515841
(Iteration 3941 / 9800) loss: 1.594257
(Iteration 3951 / 9800) loss: 1.580598
(Iteration 3961 / 9800) loss: 1.485138
(Iteration 3971 / 9800) loss: 1.490225
(Iteration 3981 / 9800) loss: 1.391344
(Iteration 3991 / 9800) loss: 1.432027
(Iteration 4001 / 9800) loss: 1.442907
(Iteration 4011 / 9800) loss: 1.635717
(Iteration 4021 / 9800) loss: 1.440485
(Iteration 4031 / 9800) loss: 1.564773
(Iteration 4041 / 9800) loss: 1.360365
(Iteration 4051 / 9800) loss: 1.519453
(Iteration 4061 / 9800) loss: 1.572552
(Iteration 4071 / 9800) loss: 1.557301
(Iteration 4081 / 9800) loss: 1.483279
(Iteration 4091 / 9800) loss: 1.585911
(Iteration 4101 / 9800) loss: 1.844662
(Iteration 4111 / 9800) loss: 1.521297
(Iteration 4121 / 9800) loss: 1.629110
(Iteration 4131 / 9800) loss: 1.526750
(Iteration 4141 / 9800) loss: 1.480680
(Iteration 4151 / 9800) loss: 1.417203
(Iteration 4161 / 9800) loss: 1.696376
(Iteration 4171 / 9800) loss: 1.572610
(Iteration 4181 / 9800) loss: 1.423435
(Iteration 4191 / 9800) loss: 1.546187
(Iteration 4201 / 9800) loss: 1.533477
(Iteration 4211 / 9800) loss: 1.676173
(Iteration 4221 / 9800) loss: 1.364870
(Iteration 4231 / 9800) loss: 1.547061
(Iteration 4241 / 9800) loss: 1.457377
```

```
(Iteration 4251 / 9800) loss: 1.402045
(Iteration 4261 / 9800) loss: 1.357053
(Iteration 4271 / 9800) loss: 1.504562
(Iteration 4281 / 9800) loss: 1.509067
(Iteration 4291 / 9800) loss: 1.514857
(Iteration 4301 / 9800) loss: 1.414504
(Iteration 4311 / 9800) loss: 1.251231
(Iteration 4321 / 9800) loss: 1.459914
(Iteration 4331 / 9800) loss: 1.609743
(Iteration 4341 / 9800) loss: 1.580173
(Iteration 4351 / 9800) loss: 1.504884
(Iteration 4361 / 9800) loss: 1.325562
(Iteration 4371 / 9800) loss: 1.378934
(Iteration 4381 / 9800) loss: 1.379053
(Iteration 4391 / 9800) loss: 1.366141
(Iteration 4401 / 9800) loss: 1.487164
(Epoch 9 / 20) train acc: 0.523000; val_acc: 0.513000
(Iteration 4411 / 9800) loss: 1.525928
(Iteration 4421 / 9800) loss: 1.703205
(Iteration 4431 / 9800) loss: 1.395123
(Iteration 4441 / 9800) loss: 1.723525
(Iteration 4451 / 9800) loss: 1.494314
(Iteration 4461 / 9800) loss: 1.402214
(Iteration 4471 / 9800) loss: 1.554910
(Iteration 4481 / 9800) loss: 1.726658
(Iteration 4491 / 9800) loss: 1.348970
(Iteration 4501 / 9800) loss: 1.682975
(Iteration 4511 / 9800) loss: 1.604938
(Iteration 4521 / 9800) loss: 1.357502
(Iteration 4531 / 9800) loss: 1.418039
(Iteration 4541 / 9800) loss: 1.511597
(Iteration 4551 / 9800) loss: 1.343764
(Iteration 4561 / 9800) loss: 1.425949
(Iteration 4571 / 9800) loss: 1.490940
(Iteration 4581 / 9800) loss: 1.508998
(Iteration 4591 / 9800) loss: 1.624346
(Iteration 4601 / 9800) loss: 1.416114
(Iteration 4611 / 9800) loss: 1.309563
(Iteration 4621 / 9800) loss: 1.471554
(Iteration 4631 / 9800) loss: 1.509284
(Iteration 4641 / 9800) loss: 1.248180
(Iteration 4651 / 9800) loss: 1.618576
(Iteration 4661 / 9800) loss: 1.628439
(Iteration 4671 / 9800) loss: 1.417222
(Iteration 4681 / 9800) loss: 1.448192
(Iteration 4691 / 9800) loss: 1.562900
(Iteration 4701 / 9800) loss: 1.532188
(Iteration 4711 / 9800) loss: 1.545789
```

```
(Iteration 4721 / 9800) loss: 1.290591
(Iteration 4731 / 9800) loss: 1.388434
(Iteration 4741 / 9800) loss: 1.651396
(Iteration 4751 / 9800) loss: 1.645137
(Iteration 4761 / 9800) loss: 1.411214
(Iteration 4771 / 9800) loss: 1.543204
(Iteration 4781 / 9800) loss: 1.249714
(Iteration 4791 / 9800) loss: 1.481508
(Iteration 4801 / 9800) loss: 1.479784
(Iteration 4811 / 9800) loss: 1.301002
(Iteration 4821 / 9800) loss: 1.650984
(Iteration 4831 / 9800) loss: 1.462113
(Iteration 4841 / 9800) loss: 1.575181
(Iteration 4851 / 9800) loss: 1.347636
(Iteration 4861 / 9800) loss: 1.603635
(Iteration 4871 / 9800) loss: 1.531189
(Iteration 4881 / 9800) loss: 1.536076
(Iteration 4891 / 9800) loss: 1.645125
(Epoch 10 / 20) train acc: 0.538000; val_acc: 0.484000
(Iteration 4901 / 9800) loss: 1.408333
(Iteration 4911 / 9800) loss: 1.652671
(Iteration 4921 / 9800) loss: 1.542331
(Iteration 4931 / 9800) loss: 1.417230
(Iteration 4941 / 9800) loss: 1.693505
(Iteration 4951 / 9800) loss: 1.449799
(Iteration 4961 / 9800) loss: 1.382232
(Iteration 4971 / 9800) loss: 1.764009
(Iteration 4981 / 9800) loss: 1.392873
(Iteration 4991 / 9800) loss: 1.402580
(Iteration 5001 / 9800) loss: 1.498606
(Iteration 5011 / 9800) loss: 1.599066
(Iteration 5021 / 9800) loss: 1.614038
(Iteration 5031 / 9800) loss: 1.436788
(Iteration 5041 / 9800) loss: 1.399939
(Iteration 5051 / 9800) loss: 1.377574
(Iteration 5061 / 9800) loss: 1.693143
(Iteration 5071 / 9800) loss: 1.392517
(Iteration 5081 / 9800) loss: 1.190464
(Iteration 5091 / 9800) loss: 1.338405
(Iteration 5101 / 9800) loss: 1.420279
(Iteration 5111 / 9800) loss: 1.469973
(Iteration 5121 / 9800) loss: 1.643326
(Iteration 5131 / 9800) loss: 1.371987
(Iteration 5141 / 9800) loss: 1.453234
(Iteration 5151 / 9800) loss: 1.335812
(Iteration 5161 / 9800) loss: 1.437008
(Iteration 5171 / 9800) loss: 1.470808
(Iteration 5181 / 9800) loss: 1.481528
```

```
(Iteration 5191 / 9800) loss: 1.754513
(Iteration 5201 / 9800) loss: 1.579893
(Iteration 5211 / 9800) loss: 1.261409
(Iteration 5221 / 9800) loss: 1.551735
(Iteration 5231 / 9800) loss: 1.458882
(Iteration 5241 / 9800) loss: 1.528471
(Iteration 5251 / 9800) loss: 1.556954
(Iteration 5261 / 9800) loss: 1.417230
(Iteration 5271 / 9800) loss: 1.652678
(Iteration 5281 / 9800) loss: 1.445809
(Iteration 5291 / 9800) loss: 1.361981
(Iteration 5301 / 9800) loss: 1.327656
(Iteration 5311 / 9800) loss: 1.567706
(Iteration 5321 / 9800) loss: 1.261019
(Iteration 5331 / 9800) loss: 1.496712
(Iteration 5341 / 9800) loss: 1.268512
(Iteration 5351 / 9800) loss: 1.480076
(Iteration 5361 / 9800) loss: 1.522993
(Iteration 5371 / 9800) loss: 1.447436
(Iteration 5381 / 9800) loss: 1.703682
(Epoch 11 / 20) train acc: 0.504000; val_acc: 0.488000
(Iteration 5391 / 9800) loss: 1.528453
(Iteration 5401 / 9800) loss: 1.724343
(Iteration 5411 / 9800) loss: 1.423530
(Iteration 5421 / 9800) loss: 1.602672
(Iteration 5431 / 9800) loss: 1.557760
(Iteration 5441 / 9800) loss: 1.394885
(Iteration 5451 / 9800) loss: 1.432927
(Iteration 5461 / 9800) loss: 1.578938
(Iteration 5471 / 9800) loss: 1.255586
(Iteration 5481 / 9800) loss: 1.400767
(Iteration 5491 / 9800) loss: 1.481712
(Iteration 5501 / 9800) loss: 1.524644
(Iteration 5511 / 9800) loss: 1.429997
(Iteration 5521 / 9800) loss: 1.436551
(Iteration 5531 / 9800) loss: 1.377820
(Iteration 5541 / 9800) loss: 1.580399
(Iteration 5551 / 9800) loss: 1.529769
(Iteration 5561 / 9800) loss: 1.594720
(Iteration 5571 / 9800) loss: 1.283374
(Iteration 5581 / 9800) loss: 1.505067
(Iteration 5591 / 9800) loss: 1.279820
(Iteration 5601 / 9800) loss: 1.568307
(Iteration 5611 / 9800) loss: 1.420498
(Iteration 5621 / 9800) loss: 1.679780
(Iteration 5631 / 9800) loss: 1.305272
(Iteration 5641 / 9800) loss: 1.446127
(Iteration 5651 / 9800) loss: 1.276549
```

```
(Iteration 5661 / 9800) loss: 1.573962
(Iteration 5671 / 9800) loss: 1.385835
(Iteration 5681 / 9800) loss: 1.403393
(Iteration 5691 / 9800) loss: 1.503540
(Iteration 5701 / 9800) loss: 1.327735
(Iteration 5711 / 9800) loss: 1.350248
(Iteration 5721 / 9800) loss: 1.476000
(Iteration 5731 / 9800) loss: 1.517045
(Iteration 5741 / 9800) loss: 1.321680
(Iteration 5751 / 9800) loss: 1.308953
(Iteration 5761 / 9800) loss: 1.828018
(Iteration 5771 / 9800) loss: 1.568223
(Iteration 5781 / 9800) loss: 1.542334
(Iteration 5791 / 9800) loss: 1.376787
(Iteration 5801 / 9800) loss: 1.392604
(Iteration 5811 / 9800) loss: 1.383767
(Iteration 5821 / 9800) loss: 1.451939
(Iteration 5831 / 9800) loss: 1.567204
(Iteration 5841 / 9800) loss: 1.516617
(Iteration 5851 / 9800) loss: 1.256301
(Iteration 5861 / 9800) loss: 1.303938
(Iteration 5871 / 9800) loss: 1.559992
(Epoch 12 / 20) train acc: 0.551000; val_acc: 0.493000
(Iteration 5881 / 9800) loss: 1.347673
(Iteration 5891 / 9800) loss: 1.386421
(Iteration 5901 / 9800) loss: 1.653612
(Iteration 5911 / 9800) loss: 1.494828
(Iteration 5921 / 9800) loss: 1.461342
(Iteration 5931 / 9800) loss: 1.368917
(Iteration 5941 / 9800) loss: 1.304509
(Iteration 5951 / 9800) loss: 1.472545
(Iteration 5961 / 9800) loss: 1.207941
(Iteration 5971 / 9800) loss: 1.469593
(Iteration 5981 / 9800) loss: 1.375389
(Iteration 5991 / 9800) loss: 1.276855
(Iteration 6001 / 9800) loss: 1.586354
(Iteration 6011 / 9800) loss: 1.479047
(Iteration 6021 / 9800) loss: 1.467469
(Iteration 6031 / 9800) loss: 1.237826
(Iteration 6041 / 9800) loss: 1.575204
(Iteration 6051 / 9800) loss: 1.487614
(Iteration 6061 / 9800) loss: 1.377267
(Iteration 6071 / 9800) loss: 1.392077
(Iteration 6081 / 9800) loss: 1.627320
(Iteration 6091 / 9800) loss: 1.310652
(Iteration 6101 / 9800) loss: 1.625362
(Iteration 6111 / 9800) loss: 1.527619
(Iteration 6121 / 9800) loss: 1.664006
```

```
(Iteration 6131 / 9800) loss: 1.543798
(Iteration 6141 / 9800) loss: 1.367192
(Iteration 6151 / 9800) loss: 1.213354
(Iteration 6161 / 9800) loss: 1.383130
(Iteration 6171 / 9800) loss: 1.341550
(Iteration 6181 / 9800) loss: 1.506521
(Iteration 6191 / 9800) loss: 1.465947
(Iteration 6201 / 9800) loss: 1.538561
(Iteration 6211 / 9800) loss: 1.695471
(Iteration 6221 / 9800) loss: 1.422370
(Iteration 6231 / 9800) loss: 1.775234
(Iteration 6241 / 9800) loss: 1.590871
(Iteration 6251 / 9800) loss: 1.383773
(Iteration 6261 / 9800) loss: 1.434053
(Iteration 6271 / 9800) loss: 1.432273
(Iteration 6281 / 9800) loss: 1.645663
(Iteration 6291 / 9800) loss: 1.545531
(Iteration 6301 / 9800) loss: 1.634211
(Iteration 6311 / 9800) loss: 1.366423
(Iteration 6321 / 9800) loss: 1.545642
(Iteration 6331 / 9800) loss: 1.358938
(Iteration 6341 / 9800) loss: 1.329699
(Iteration 6351 / 9800) loss: 1.485661
(Iteration 6361 / 9800) loss: 1.565718
(Epoch 13 / 20) train acc: 0.533000; val_acc: 0.453000
(Iteration 6371 / 9800) loss: 1.345228
(Iteration 6381 / 9800) loss: 1.385432
(Iteration 6391 / 9800) loss: 1.655049
(Iteration 6401 / 9800) loss: 1.469613
(Iteration 6411 / 9800) loss: 1.302424
(Iteration 6421 / 9800) loss: 1.198985
(Iteration 6431 / 9800) loss: 1.526096
(Iteration 6441 / 9800) loss: 1.386061
(Iteration 6451 / 9800) loss: 1.329525
(Iteration 6461 / 9800) loss: 1.408121
(Iteration 6471 / 9800) loss: 1.642860
(Iteration 6481 / 9800) loss: 1.589777
(Iteration 6491 / 9800) loss: 1.402365
(Iteration 6501 / 9800) loss: 1.477863
(Iteration 6511 / 9800) loss: 1.415716
(Iteration 6521 / 9800) loss: 1.424824
(Iteration 6531 / 9800) loss: 1.727986
(Iteration 6541 / 9800) loss: 1.582082
(Iteration 6551 / 9800) loss: 1.359019
(Iteration 6561 / 9800) loss: 1.314504
(Iteration 6571 / 9800) loss: 1.551289
(Iteration 6581 / 9800) loss: 1.381078
(Iteration 6591 / 9800) loss: 1.207247
```

```
(Iteration 6601 / 9800) loss: 1.579060
(Iteration 6611 / 9800) loss: 1.598761
(Iteration 6621 / 9800) loss: 1.569972
(Iteration 6631 / 9800) loss: 1.599652
(Iteration 6641 / 9800) loss: 1.531586
(Iteration 6651 / 9800) loss: 1.426410
(Iteration 6661 / 9800) loss: 1.431387
(Iteration 6671 / 9800) loss: 1.380361
(Iteration 6681 / 9800) loss: 1.635459
(Iteration 6691 / 9800) loss: 1.417210
(Iteration 6701 / 9800) loss: 1.377940
(Iteration 6711 / 9800) loss: 1.388259
(Iteration 6721 / 9800) loss: 1.460442
(Iteration 6731 / 9800) loss: 1.519184
(Iteration 6741 / 9800) loss: 1.347541
(Iteration 6751 / 9800) loss: 1.375582
(Iteration 6761 / 9800) loss: 1.514156
(Iteration 6771 / 9800) loss: 1.483618
(Iteration 6781 / 9800) loss: 1.533758
(Iteration 6791 / 9800) loss: 1.413651
(Iteration 6801 / 9800) loss: 1.393275
(Iteration 6811 / 9800) loss: 1.280467
(Iteration 6821 / 9800) loss: 1.401343
(Iteration 6831 / 9800) loss: 1.543209
(Iteration 6841 / 9800) loss: 1.399974
(Iteration 6851 / 9800) loss: 1.443419
(Epoch 14 / 20) train acc: 0.528000; val_acc: 0.479000
(Iteration 6861 / 9800) loss: 1.312229
(Iteration 6871 / 9800) loss: 1.649718
(Iteration 6881 / 9800) loss: 1.408353
(Iteration 6891 / 9800) loss: 1.511512
(Iteration 6901 / 9800) loss: 1.468498
(Iteration 6911 / 9800) loss: 1.531685
(Iteration 6921 / 9800) loss: 1.297078
(Iteration 6931 / 9800) loss: 1.417385
(Iteration 6941 / 9800) loss: 1.428264
(Iteration 6951 / 9800) loss: 1.265869
(Iteration 6961 / 9800) loss: 1.409231
(Iteration 6971 / 9800) loss: 1.679238
(Iteration 6981 / 9800) loss: 1.515718
(Iteration 6991 / 9800) loss: 1.436940
(Iteration 7001 / 9800) loss: 1.242022
(Iteration 7011 / 9800) loss: 1.307948
(Iteration 7021 / 9800) loss: 1.536091
(Iteration 7031 / 9800) loss: 1.442585
(Iteration 7041 / 9800) loss: 1.347001
(Iteration 7051 / 9800) loss: 1.463050
(Iteration 7061 / 9800) loss: 1.523346
```

```
(Iteration 7071 / 9800) loss: 1.627773
(Iteration 7081 / 9800) loss: 1.326607
(Iteration 7091 / 9800) loss: 1.278926
(Iteration 7101 / 9800) loss: 1.580000
(Iteration 7111 / 9800) loss: 1.370527
(Iteration 7121 / 9800) loss: 1.434732
(Iteration 7131 / 9800) loss: 1.457447
(Iteration 7141 / 9800) loss: 1.667412
(Iteration 7151 / 9800) loss: 1.537978
(Iteration 7161 / 9800) loss: 1.366083
(Iteration 7171 / 9800) loss: 1.401110
(Iteration 7181 / 9800) loss: 1.308210
(Iteration 7191 / 9800) loss: 1.390263
(Iteration 7201 / 9800) loss: 1.673033
(Iteration 7211 / 9800) loss: 1.347204
(Iteration 7221 / 9800) loss: 1.393550
(Iteration 7231 / 9800) loss: 1.301388
(Iteration 7241 / 9800) loss: 1.357073
(Iteration 7251 / 9800) loss: 1.332571
(Iteration 7261 / 9800) loss: 1.538966
(Iteration 7271 / 9800) loss: 1.389765
(Iteration 7281 / 9800) loss: 1.465888
(Iteration 7291 / 9800) loss: 1.440153
(Iteration 7301 / 9800) loss: 1.501286
(Iteration 7311 / 9800) loss: 1.434549
(Iteration 7321 / 9800) loss: 1.559246
(Iteration 7331 / 9800) loss: 1.645360
(Iteration 7341 / 9800) loss: 1.435822
(Epoch 15 / 20) train acc: 0.547000; val_acc: 0.513000
(Iteration 7351 / 9800) loss: 1.246534
(Iteration 7361 / 9800) loss: 1.286941
(Iteration 7371 / 9800) loss: 1.484595
(Iteration 7381 / 9800) loss: 1.338905
(Iteration 7391 / 9800) loss: 1.490881
(Iteration 7401 / 9800) loss: 1.350771
(Iteration 7411 / 9800) loss: 1.390256
(Iteration 7421 / 9800) loss: 1.317192
(Iteration 7431 / 9800) loss: 1.371558
(Iteration 7441 / 9800) loss: 1.356942
(Iteration 7451 / 9800) loss: 1.431572
(Iteration 7461 / 9800) loss: 1.451786
(Iteration 7471 / 9800) loss: 1.604030
(Iteration 7481 / 9800) loss: 1.640113
(Iteration 7491 / 9800) loss: 1.376466
(Iteration 7501 / 9800) loss: 1.559902
(Iteration 7511 / 9800) loss: 1.596243
(Iteration 7521 / 9800) loss: 1.493945
(Iteration 7531 / 9800) loss: 1.699541
```

```
(Iteration 7541 / 9800) loss: 1.268155
(Iteration 7551 / 9800) loss: 1.458684
(Iteration 7561 / 9800) loss: 1.223571
(Iteration 7571 / 9800) loss: 1.403690
(Iteration 7581 / 9800) loss: 1.446652
(Iteration 7591 / 9800) loss: 1.583932
(Iteration 7601 / 9800) loss: 1.534172
(Iteration 7611 / 9800) loss: 1.551562
(Iteration 7621 / 9800) loss: 1.442467
(Iteration 7631 / 9800) loss: 1.445312
(Iteration 7641 / 9800) loss: 1.536613
(Iteration 7651 / 9800) loss: 1.360484
(Iteration 7661 / 9800) loss: 1.335868
(Iteration 7671 / 9800) loss: 1.405937
(Iteration 7681 / 9800) loss: 1.290391
(Iteration 7691 / 9800) loss: 1.559121
(Iteration 7701 / 9800) loss: 1.340304
(Iteration 7711 / 9800) loss: 1.448676
(Iteration 7721 / 9800) loss: 1.519204
(Iteration 7731 / 9800) loss: 1.695587
(Iteration 7741 / 9800) loss: 1.394855
(Iteration 7751 / 9800) loss: 1.309602
(Iteration 7761 / 9800) loss: 1.535230
(Iteration 7771 / 9800) loss: 1.507616
(Iteration 7781 / 9800) loss: 1.516943
(Iteration 7791 / 9800) loss: 1.443150
(Iteration 7801 / 9800) loss: 1.872835
(Iteration 7811 / 9800) loss: 1.214535
(Iteration 7821 / 9800) loss: 1.279015
(Iteration 7831 / 9800) loss: 1.346656
(Epoch 16 / 20) train acc: 0.526000; val_acc: 0.459000
(Iteration 7841 / 9800) loss: 1.511716
(Iteration 7851 / 9800) loss: 1.561761
(Iteration 7861 / 9800) loss: 1.495427
(Iteration 7871 / 9800) loss: 1.241716
(Iteration 7881 / 9800) loss: 1.444474
(Iteration 7891 / 9800) loss: 1.594080
(Iteration 7901 / 9800) loss: 1.350312
(Iteration 7911 / 9800) loss: 1.392910
(Iteration 7921 / 9800) loss: 1.357885
(Iteration 7931 / 9800) loss: 1.324794
(Iteration 7941 / 9800) loss: 1.320132
(Iteration 7951 / 9800) loss: 1.475566
(Iteration 7961 / 9800) loss: 1.468488
(Iteration 7971 / 9800) loss: 1.408101
(Iteration 7981 / 9800) loss: 1.428311
(Iteration 7991 / 9800) loss: 1.476115
(Iteration 8001 / 9800) loss: 1.390082
```

```
(Iteration 8011 / 9800) loss: 1.256199
(Iteration 8021 / 9800) loss: 1.543161
(Iteration 8031 / 9800) loss: 1.527044
(Iteration 8041 / 9800) loss: 1.311414
(Iteration 8051 / 9800) loss: 1.411761
(Iteration 8061 / 9800) loss: 1.561269
(Iteration 8071 / 9800) loss: 1.614447
(Iteration 8081 / 9800) loss: 1.377361
(Iteration 8091 / 9800) loss: 1.592493
(Iteration 8101 / 9800) loss: 1.348131
(Iteration 8111 / 9800) loss: 1.405527
(Iteration 8121 / 9800) loss: 1.352430
(Iteration 8131 / 9800) loss: 1.444509
(Iteration 8141 / 9800) loss: 1.261284
(Iteration 8151 / 9800) loss: 1.516251
(Iteration 8161 / 9800) loss: 1.393442
(Iteration 8171 / 9800) loss: 1.387662
(Iteration 8181 / 9800) loss: 1.619519
(Iteration 8191 / 9800) loss: 1.161108
(Iteration 8201 / 9800) loss: 1.412846
(Iteration 8211 / 9800) loss: 1.327079
(Iteration 8221 / 9800) loss: 1.497561
(Iteration 8231 / 9800) loss: 1.326732
(Iteration 8241 / 9800) loss: 1.240190
(Iteration 8251 / 9800) loss: 1.510455
(Iteration 8261 / 9800) loss: 1.465587
(Iteration 8271 / 9800) loss: 1.393269
(Iteration 8281 / 9800) loss: 1.297412
(Iteration 8291 / 9800) loss: 1.693483
(Iteration 8301 / 9800) loss: 1.508567
(Iteration 8311 / 9800) loss: 1.419232
(Iteration 8321 / 9800) loss: 1.245371
(Epoch 17 / 20) train acc: 0.564000; val_acc: 0.475000
(Iteration 8331 / 9800) loss: 1.604416
(Iteration 8341 / 9800) loss: 1.521123
(Iteration 8351 / 9800) loss: 1.408342
(Iteration 8361 / 9800) loss: 1.477410
(Iteration 8371 / 9800) loss: 1.431844
(Iteration 8381 / 9800) loss: 1.565380
(Iteration 8391 / 9800) loss: 1.610895
(Iteration 8401 / 9800) loss: 1.638133
(Iteration 8411 / 9800) loss: 1.604634
(Iteration 8421 / 9800) loss: 1.387509
(Iteration 8431 / 9800) loss: 1.325959
(Iteration 8441 / 9800) loss: 1.411801
(Iteration 8451 / 9800) loss: 1.360193
(Iteration 8461 / 9800) loss: 1.292322
(Iteration 8471 / 9800) loss: 1.415923
```

```
(Iteration 8481 / 9800) loss: 1.236264
(Iteration 8491 / 9800) loss: 1.369771
(Iteration 8501 / 9800) loss: 1.438280
(Iteration 8511 / 9800) loss: 1.194449
(Iteration 8521 / 9800) loss: 1.659302
(Iteration 8531 / 9800) loss: 1.719297
(Iteration 8541 / 9800) loss: 1.364069
(Iteration 8551 / 9800) loss: 1.305104
(Iteration 8561 / 9800) loss: 1.624369
(Iteration 8571 / 9800) loss: 1.386297
(Iteration 8581 / 9800) loss: 1.328896
(Iteration 8591 / 9800) loss: 1.377681
(Iteration 8601 / 9800) loss: 1.483606
(Iteration 8611 / 9800) loss: 1.500665
(Iteration 8621 / 9800) loss: 1.383454
(Iteration 8631 / 9800) loss: 1.305969
(Iteration 8641 / 9800) loss: 1.432051
(Iteration 8651 / 9800) loss: 1.326778
(Iteration 8661 / 9800) loss: 1.711645
(Iteration 8671 / 9800) loss: 1.332585
(Iteration 8681 / 9800) loss: 1.285507
(Iteration 8691 / 9800) loss: 1.555901
(Iteration 8701 / 9800) loss: 1.442105
(Iteration 8711 / 9800) loss: 1.324235
(Iteration 8721 / 9800) loss: 1.493924
(Iteration 8731 / 9800) loss: 1.255267
(Iteration 8741 / 9800) loss: 1.500624
(Iteration 8751 / 9800) loss: 1.399839
(Iteration 8761 / 9800) loss: 1.351770
(Iteration 8771 / 9800) loss: 1.448903
(Iteration 8781 / 9800) loss: 1.466812
(Iteration 8791 / 9800) loss: 1.309928
(Iteration 8801 / 9800) loss: 1.391549
(Iteration 8811 / 9800) loss: 1.582554
(Epoch 18 / 20) train acc: 0.568000; val_acc: 0.488000
(Iteration 8821 / 9800) loss: 1.194988
(Iteration 8831 / 9800) loss: 1.492285
(Iteration 8841 / 9800) loss: 1.442079
(Iteration 8851 / 9800) loss: 1.628941
(Iteration 8861 / 9800) loss: 1.470106
(Iteration 8871 / 9800) loss: 1.280028
(Iteration 8881 / 9800) loss: 1.278096
(Iteration 8891 / 9800) loss: 1.584093
(Iteration 8901 / 9800) loss: 1.433191
(Iteration 8911 / 9800) loss: 1.642491
(Iteration 8921 / 9800) loss: 1.278854
(Iteration 8931 / 9800) loss: 1.467769
(Iteration 8941 / 9800) loss: 1.476418
```

```
(Iteration 8951 / 9800) loss: 1.569887
(Iteration 8961 / 9800) loss: 1.421801
(Iteration 8971 / 9800) loss: 1.516784
(Iteration 8981 / 9800) loss: 1.525347
(Iteration 8991 / 9800) loss: 1.475767
(Iteration 9001 / 9800) loss: 1.336710
(Iteration 9011 / 9800) loss: 1.448113
(Iteration 9021 / 9800) loss: 1.350371
(Iteration 9031 / 9800) loss: 1.379282
(Iteration 9041 / 9800) loss: 1.466175
(Iteration 9051 / 9800) loss: 1.525038
(Iteration 9061 / 9800) loss: 1.592774
(Iteration 9071 / 9800) loss: 1.412256
(Iteration 9081 / 9800) loss: 1.464268
(Iteration 9091 / 9800) loss: 1.444926
(Iteration 9101 / 9800) loss: 1.451269
(Iteration 9111 / 9800) loss: 1.399292
(Iteration 9121 / 9800) loss: 1.433781
(Iteration 9131 / 9800) loss: 1.440373
(Iteration 9141 / 9800) loss: 1.332975
(Iteration 9151 / 9800) loss: 1.365030
(Iteration 9161 / 9800) loss: 1.485674
(Iteration 9171 / 9800) loss: 1.664748
(Iteration 9181 / 9800) loss: 1.310640
(Iteration 9191 / 9800) loss: 1.490869
(Iteration 9201 / 9800) loss: 1.445392
(Iteration 9211 / 9800) loss: 1.468868
(Iteration 9221 / 9800) loss: 1.697893
(Iteration 9231 / 9800) loss: 1.438082
(Iteration 9241 / 9800) loss: 1.401317
(Iteration 9251 / 9800) loss: 1.332709
(Iteration 9261 / 9800) loss: 1.561992
(Iteration 9271 / 9800) loss: 1.487145
(Iteration 9281 / 9800) loss: 1.364595
(Iteration 9291 / 9800) loss: 1.689735
(Iteration 9301 / 9800) loss: 1.631887
(Epoch 19 / 20) train acc: 0.566000; val_acc: 0.501000
(Iteration 9311 / 9800) loss: 1.498371
(Iteration 9321 / 9800) loss: 1.370813
(Iteration 9331 / 9800) loss: 1.330317
(Iteration 9341 / 9800) loss: 1.328887
(Iteration 9351 / 9800) loss: 1.430692
(Iteration 9361 / 9800) loss: 1.598822
(Iteration 9371 / 9800) loss: 1.405681
(Iteration 9381 / 9800) loss: 1.538734
(Iteration 9391 / 9800) loss: 1.466522
(Iteration 9401 / 9800) loss: 1.341517
(Iteration 9411 / 9800) loss: 1.320962
```

```
(Iteration 9421 / 9800) loss: 1.406886
(Iteration 9431 / 9800) loss: 1.443689
(Iteration 9441 / 9800) loss: 1.313301
(Iteration 9451 / 9800) loss: 1.471692
(Iteration 9461 / 9800) loss: 1.494109
(Iteration 9471 / 9800) loss: 1.482124
(Iteration 9481 / 9800) loss: 1.407160
(Iteration 9491 / 9800) loss: 1.237466
(Iteration 9501 / 9800) loss: 1.230314
(Iteration 9511 / 9800) loss: 1.424442
(Iteration 9521 / 9800) loss: 1.384581
(Iteration 9531 / 9800) loss: 1.620986
(Iteration 9541 / 9800) loss: 1.610797
(Iteration 9551 / 9800) loss: 1.316518
(Iteration 9561 / 9800) loss: 1.339717
(Iteration 9571 / 9800) loss: 1.519730
(Iteration 9581 / 9800) loss: 1.487726
(Iteration 9591 / 9800) loss: 1.377622
(Iteration 9601 / 9800) loss: 1.430297
(Iteration 9611 / 9800) loss: 1.337096
(Iteration 9621 / 9800) loss: 1.377094
(Iteration 9631 / 9800) loss: 1.417364
(Iteration 9641 / 9800) loss: 1.515889
(Iteration 9651 / 9800) loss: 1.359126
(Iteration 9661 / 9800) loss: 1.641270
(Iteration 9671 / 9800) loss: 1.313246
(Iteration 9681 / 9800) loss: 1.423751
(Iteration 9691 / 9800) loss: 1.502911
(Iteration 9701 / 9800) loss: 1.383743
(Iteration 9711 / 9800) loss: 1.269332
(Iteration 9721 / 9800) loss: 1.478456
(Iteration 9731 / 9800) loss: 1.388372
(Iteration 9741 / 9800) loss: 1.279618
(Iteration 9751 / 9800) loss: 1.514683
(Iteration 9761 / 9800) loss: 1.501979
(Iteration 9771 / 9800) loss: 1.509629
(Iteration 9781 / 9800) loss: 1.412340
(Iteration 9791 / 9800) loss: 1.351445
(Epoch 20 / 20) train acc: 0.532000; val_acc: 0.487000
```

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.

[35]:
```
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:  0.513
Test set accuracy:  0.525
```