knn

October 8, 2022

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

# Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment1/'
FOLDERNAME = 'enpm809K/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 drive/My\ Drive/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/enpm809K/assignments/assignment1/cs231n/datasets
/content/drive/My Drive/enpm809K/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.

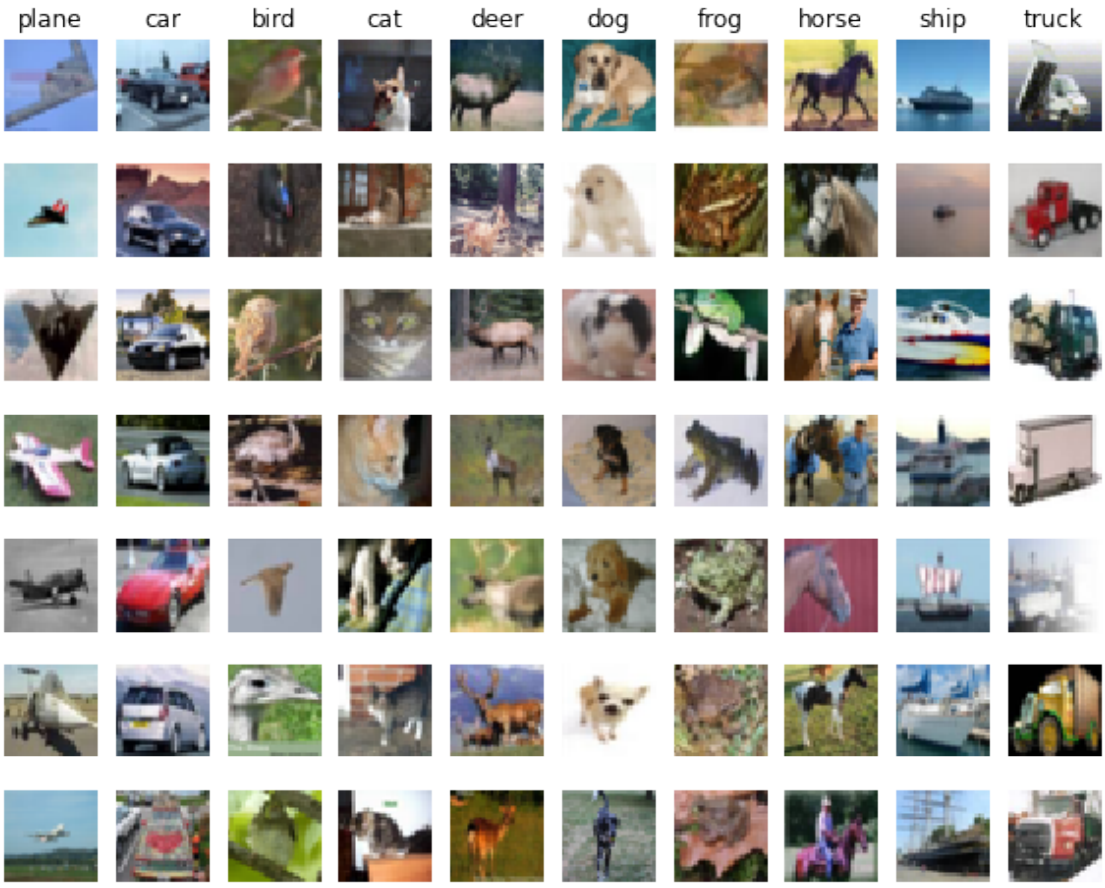
[3]:
```
# Run some setup code for this notebook.

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
```

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

[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.
# PLEASE DO NOT MODIFY THE MARKERS
print('||||||||||||||||||||||||||||||||||||||||||||||||')
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)
# PLEASE DO NOT MODIFY THE MARKERS
print('``````````````````````````````````````````````')
```

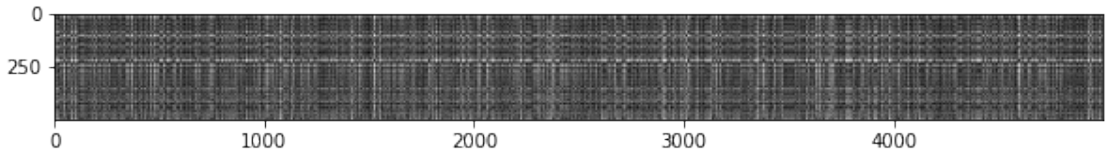
```
||||||||||||||||||||||||||||||||||||||||||||
Training data shape:  (50000, 32, 32, 3)
Training labels shape:  (50000,)
Test data shape:  (10000, 32, 32, 3)
Test labels shape:  (10000,)
``````````````````````````````````````````````
```

[5]:
```
# Visualize some examples from the dataset.
# We show a few examples of training images from each class.
# PLEASE DO NOT MODIFY THE MARKERS
print('||||||||||||||||||||||||||||||||||||||||||||')
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()
print('``')
```

```
||||||||||||||||||||||||||||||||||||||||||||
```
```




```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

[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))
# PLEASE DO NOT MODIFY THE MARKERS
print('||||||||||||||||||||||||||||||||||||||||||||||||||')
print(X_train.shape, X_test.shape)
```

```
print('``````````````````````````````````````````````')
```

```
||||||||||||||||||||||||||||||||||||||||||||
(5000, 3072) (500, 3072)
``````````````````````````````````````````````
```

[7]:
```
from cs231n.classifiers import KNearestNeighbor
# PLEASE DO NOT MODIFY THE MARKERS
print('||||||||||||||||||||||||||||||||||||||||||||||')
# 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)
print('``')
```

```
||||||||||||||||||||||||||||||||||||||||||||||
``````````````````````````````````````````````
```

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.
# PLEASE DO NOT MODIFY THE MARKERS
print('||||||||||||||||||||||||||||||||||||||||||||||||')

# Test your implementation:
dists = classifier.compute_distances_two_loops(X_test)
print(dists.shape)
# PLEASE DO NOT MODIFY THE MARKERS
print('``````````````````````````````````````````````````')
```

```
||||||||||||||||||||||||||||||||||||||||||||||
(500, 5000)
```

```
```

[9]:
```
# We can visualize the distance matrix: each row is a single test example and
# its distances to training examples
# PLEASE DO NOT MODIFY THE MARKERS
print('||||||||||||||||||||||||||||||||||||||||||||||||')
plt.imshow(dists, interpolation='none')
plt.show()
# PLEASE DO NOT MODIFY THE MARKERS
print('``````````````````````````````````````````````')
```

```
||||||||||||||||||||||||||||||||||||||||||||||||
```




```
``````````````````````````````````````````````
```

**Inline Question 1**

Notice the structured patterns in the distance matrix, where some rows or columns are visible 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 : Distance is highly related to the brighness of this figure. when the pixel is brighter the distance is higher. Based on this information, the cause behind the distinctly rows means the data is not close to any other training data. The columns means the comparision between itself and testing data.*

[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
# PLEASE DO NOT MODIFY THE MARKERS
print('||||||||||||||||||||||||||||||||||||||||||||||||')
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
# PLEASE DO NOT MODIFY THE MARKERS
print('``')
```
```

```
||||||||||||||||||||||||||||||||||||||||||||||||
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]:
```
# PLEASE DO NOT MODIFY THE MARKERS
print('||||||||||||||||||||||||||||||||||||||||||||||||')
y_test_pred = classifier.predict_labels(dists, k=5)
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
# PLEASE DO NOT MODIFY THE MARKERS
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
print('``')
```

```
||||||||||||||||||||||||||||||||||||||||||||||||
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 $\mu$ 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 $\mu_{ij}$ across all images is

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

The general standard deviation $\sigma$ and pixel-wise standard deviation $\sigma_{ij}$ is defined similarly.

Which of the following preprocessing steps will not change the performance of a Nearest Neighbor classifier that uses L1 distance? Select all that apply. 1. Subtracting the mean $\mu$ ($\tilde{p}_{ij}^{(k)} = p_{ij}^{(k)} - \mu$.) 2. Subtracting the per pixel mean $\mu_{ij}$ ($\tilde{p}_{ij}^{(k)} = p_{ij}^{(k)} - \mu_{ij}$.) 3. Subtracting the mean $\mu$ and dividing by the standard deviation $\sigma$. 4. Subtracting the pixel-wise mean $\mu_{ij}$ and dividing by the pixel-wise standard deviation $\sigma_{ij}$. 5. Rotating the coordinate axes of the data.

*YourAnswer* : 1. 3. 5. *YourExplanation* :

Ans: 1 and 3.: because every pixels subtracting the same value, so the manhantan distance is still same, a linear combination will not effect the performance in L1.
Ans: 5.: because every data is rotated so it will not cause any difference.

Ans 2 and 4.: will cause a difference of KNN performance because every data are not doing the same linear combination.

[12]:
```
# PLEASE DO NOT MODIFY THE MARKERS
print('||||||||||||||||||||||||||||||||||||||||||||||||')
# 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')
# PLEASE DO NOT MODIFY THE MARKERS
print('``````````````````````````````````````````````````')
```

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

[13]:
```
# PLEASE DO NOT MODIFY THE MARKERS
print('||||||||||||||||||||||||||||||||||||||||||||||||')
# 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')
# PLEASE DO NOT MODIFY THE MARKERS
print('``')
```

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

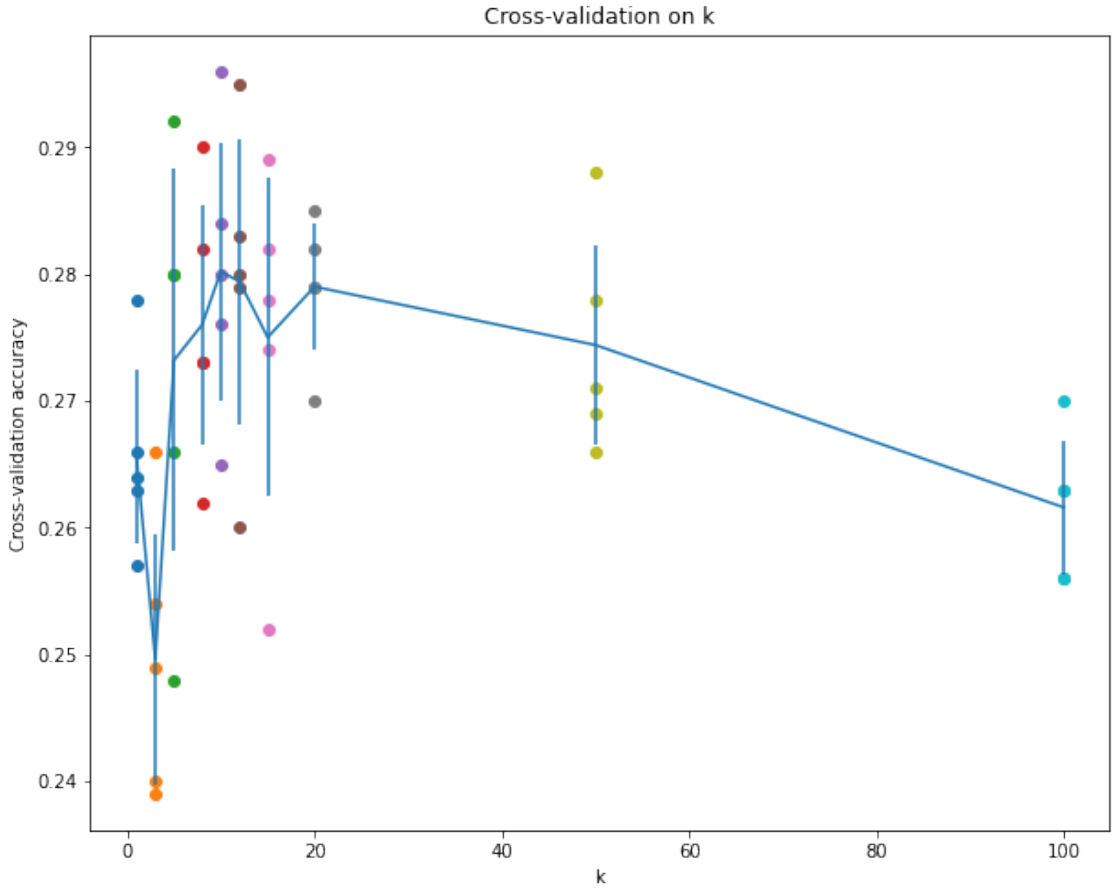
```
````````````````````````````````````````````````
```

[14]:
```
# PLEASE DO NOT MODIFY THE MARKERS
print('||||||||||||||||||||||||||||||||||||||||||||||||')
# 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.
# PLEASE DO NOT MODIFY THE MARKERS
print('``')
```

```
||||||||||||||||||||||||||||||||||||||||||||||||
Two loop version took 35.969367 seconds
One loop version took 45.354480 seconds
No loop version took 0.597947 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]:
```
# PLEASE DO NOT MODIFY THE MARKERS
print('||||||||||||||||||||||||||||||||||||||||||||||||')
```

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

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

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

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


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

    current_X_test_folds = np.array(current_X_train_folds.pop(num))
    current_Y_test_folds = np.array(current_y_train_folds.pop(num))

    #print(np.concatenate(current_X_train_folds).shape)
    classifier.train(np.concatenate(current_X_train_folds), np.
 ↪concatenate(current_y_train_folds))
```

```
    dists = classifier.compute_distances_no_loops(current_X_test_folds)
    y_test_pred = classifier.predict_labels(dists, k=k_c)

    # Compute and print the fraction of correctly predicted examples
    num_correct = np.sum(y_test_pred == current_Y_test_folds)
    accuracy = float(num_correct) / len(current_Y_test_folds)
    # PLEASE DO NOT MODIFY THE MARKERS
    print('||||||||||||||||||||||||||||||||||||||||||||||||')
    print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test,␣
 ↪accuracy))
    k_list.append(accuracy)
  k_to_accuracies[k_c] = k_list

print(k_to_accuracies)
# pass

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

# Print out the computed accuracies
for k in sorted(k_to_accuracies):
    for accuracy in k_to_accuracies[k]:
        print('k = %d, accuracy = %f' % (k, accuracy))
# PLEASE DO NOT MODIFY THE MARKERS
print('````````````````````````````````````````````````')
```

```
||||||||||||||||||||||||||||||||||||||||||||||||
||||||||||||||||||||||||||||||||||||||||||||||||
Got 263 / 500 correct => accuracy: 0.263000
||||||||||||||||||||||||||||||||||||||||||||||||
Got 257 / 500 correct => accuracy: 0.257000
||||||||||||||||||||||||||||||||||||||||||||||||
Got 264 / 500 correct => accuracy: 0.264000
||||||||||||||||||||||||||||||||||||||||||||||||
Got 278 / 500 correct => accuracy: 0.278000
||||||||||||||||||||||||||||||||||||||||||||||||
Got 266 / 500 correct => accuracy: 0.266000
||||||||||||||||||||||||||||||||||||||||||||||||
Got 239 / 500 correct => accuracy: 0.239000
||||||||||||||||||||||||||||||||||||||||||||||||
Got 249 / 500 correct => accuracy: 0.249000
||||||||||||||||||||||||||||||||||||||||||||||||
Got 240 / 500 correct => accuracy: 0.240000
||||||||||||||||||||||||||||||||||||||||||||||||
Got 266 / 500 correct => accuracy: 0.266000
||||||||||||||||||||||||||||||||||||||||||||||||
Got 254 / 500 correct => accuracy: 0.254000
||||||||||||||||||||||||||||||||||||||||||||||||
```

```
Got 248 / 500 correct => accuracy: 0.248000
|||||||||||||||||||||||||||||||||||||||||||||||||
Got 266 / 500 correct => accuracy: 0.266000
|||||||||||||||||||||||||||||||||||||||||||||||||||
Got 280 / 500 correct => accuracy: 0.280000
|||||||||||||||||||||||||||||||||||||||||||||||||||
Got 292 / 500 correct => accuracy: 0.292000
|||||||||||||||||||||||||||||||||||||||||||||||||||
Got 280 / 500 correct => accuracy: 0.280000
|||||||||||||||||||||||||||||||||||||||||||||||||||
Got 262 / 500 correct => accuracy: 0.262000
|||||||||||||||||||||||||||||||||||||||||||||||||||
Got 282 / 500 correct => accuracy: 0.282000
|||||||||||||||||||||||||||||||||||||||||||||||||||
Got 273 / 500 correct => accuracy: 0.273000
|||||||||||||||||||||||||||||||||||||||||||||||||||
Got 290 / 500 correct => accuracy: 0.290000
|||||||||||||||||||||||||||||||||||||||||||||||||||
Got 273 / 500 correct => accuracy: 0.273000
|||||||||||||||||||||||||||||||||||||||||||||||||||
Got 265 / 500 correct => accuracy: 0.265000
|||||||||||||||||||||||||||||||||||||||||||||||||||
Got 296 / 500 correct => accuracy: 0.296000
|||||||||||||||||||||||||||||||||||||||||||||||||||
Got 276 / 500 correct => accuracy: 0.276000
|||||||||||||||||||||||||||||||||||||||||||||||||||
Got 284 / 500 correct => accuracy: 0.284000
|||||||||||||||||||||||||||||||||||||||||||||||||||
Got 280 / 500 correct => accuracy: 0.280000
|||||||||||||||||||||||||||||||||||||||||||||||||||
Got 260 / 500 correct => accuracy: 0.260000
|||||||||||||||||||||||||||||||||||||||||||||||||||
Got 295 / 500 correct => accuracy: 0.295000
|||||||||||||||||||||||||||||||||||||||||||||||||||
Got 279 / 500 correct => accuracy: 0.279000
|||||||||||||||||||||||||||||||||||||||||||||||||||
Got 283 / 500 correct => accuracy: 0.283000
|||||||||||||||||||||||||||||||||||||||||||||||||||
Got 280 / 500 correct => accuracy: 0.280000
|||||||||||||||||||||||||||||||||||||||||||||||||||
Got 252 / 500 correct => accuracy: 0.252000
|||||||||||||||||||||||||||||||||||||||||||||||||||
Got 289 / 500 correct => accuracy: 0.289000
|||||||||||||||||||||||||||||||||||||||||||||||||||
Got 278 / 500 correct => accuracy: 0.278000
|||||||||||||||||||||||||||||||||||||||||||||||||||
Got 282 / 500 correct => accuracy: 0.282000
|||||||||||||||||||||||||||||||||||||||||||||||||||
```

```
Got 274 / 500 correct => accuracy: 0.274000
|||||||||||||||||||||||||||||||||||||||||||||||||
Got 270 / 500 correct => accuracy: 0.270000
|||||||||||||||||||||||||||||||||||||||||||||||||
Got 279 / 500 correct => accuracy: 0.279000
|||||||||||||||||||||||||||||||||||||||||||||||||
Got 279 / 500 correct => accuracy: 0.279000
|||||||||||||||||||||||||||||||||||||||||||||||||
Got 282 / 500 correct => accuracy: 0.282000
|||||||||||||||||||||||||||||||||||||||||||||||||
Got 285 / 500 correct => accuracy: 0.285000
|||||||||||||||||||||||||||||||||||||||||||||||||
Got 271 / 500 correct => accuracy: 0.271000
|||||||||||||||||||||||||||||||||||||||||||||||||
Got 288 / 500 correct => accuracy: 0.288000
|||||||||||||||||||||||||||||||||||||||||||||||||
Got 278 / 500 correct => accuracy: 0.278000
|||||||||||||||||||||||||||||||||||||||||||||||||
Got 269 / 500 correct => accuracy: 0.269000
|||||||||||||||||||||||||||||||||||||||||||||||||
Got 266 / 500 correct => accuracy: 0.266000
|||||||||||||||||||||||||||||||||||||||||||||||||
Got 256 / 500 correct => accuracy: 0.256000
|||||||||||||||||||||||||||||||||||||||||||||||||
Got 270 / 500 correct => accuracy: 0.270000
|||||||||||||||||||||||||||||||||||||||||||||||||
Got 263 / 500 correct => accuracy: 0.263000
|||||||||||||||||||||||||||||||||||||||||||||||||
Got 256 / 500 correct => accuracy: 0.256000
|||||||||||||||||||||||||||||||||||||||||||||||||
Got 263 / 500 correct => accuracy: 0.263000
{1: [0.263, 0.257, 0.264, 0.278, 0.266], 3: [0.239, 0.249, 0.24, 0.266, 0.254],
5: [0.248, 0.266, 0.28, 0.292, 0.28], 8: [0.262, 0.282, 0.273, 0.29, 0.273], 10:
[0.265, 0.296, 0.276, 0.284, 0.28], 12: [0.26, 0.295, 0.279, 0.283, 0.28], 15:
[0.252, 0.289, 0.278, 0.282, 0.274], 20: [0.27, 0.279, 0.279, 0.282, 0.285], 50:
[0.271, 0.288, 0.278, 0.269, 0.266], 100: [0.256, 0.27, 0.263, 0.256, 0.263]}
k = 1, accuracy = 0.263000
k = 1, accuracy = 0.257000
k = 1, accuracy = 0.264000
k = 1, accuracy = 0.278000
k = 1, accuracy = 0.266000
k = 3, accuracy = 0.239000
k = 3, accuracy = 0.249000
k = 3, accuracy = 0.240000
k = 3, accuracy = 0.266000
k = 3, accuracy = 0.254000
k = 5, accuracy = 0.248000
k = 5, accuracy = 0.266000
```

```
k = 5, accuracy = 0.280000
k = 5, accuracy = 0.292000
k = 5, accuracy = 0.280000
k = 8, accuracy = 0.262000
k = 8, accuracy = 0.282000
k = 8, accuracy = 0.273000
k = 8, accuracy = 0.290000
k = 8, accuracy = 0.273000
k = 10, accuracy = 0.265000
k = 10, accuracy = 0.296000
k = 10, accuracy = 0.276000
k = 10, accuracy = 0.284000
k = 10, accuracy = 0.280000
k = 12, accuracy = 0.260000
k = 12, accuracy = 0.295000
k = 12, accuracy = 0.279000
k = 12, accuracy = 0.283000
k = 12, accuracy = 0.280000
k = 15, accuracy = 0.252000
k = 15, accuracy = 0.289000
k = 15, accuracy = 0.278000
k = 15, accuracy = 0.282000
k = 15, accuracy = 0.274000
k = 20, accuracy = 0.270000
k = 20, accuracy = 0.279000
k = 20, accuracy = 0.279000
k = 20, accuracy = 0.282000
k = 20, accuracy = 0.285000
k = 50, accuracy = 0.271000
k = 50, accuracy = 0.288000
k = 50, accuracy = 0.278000
k = 50, accuracy = 0.269000
k = 50, accuracy = 0.266000
k = 100, accuracy = 0.256000
k = 100, accuracy = 0.270000
k = 100, accuracy = 0.263000
k = 100, accuracy = 0.256000
k = 100, accuracy = 0.263000
```

[16]:
```
# PLEASE DO NOT MODIFY THE MARKERS
print('||||||||||||||||||||||||||||||||||||||||||||||||')
# 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()
# PLEASE DO NOT MODIFY THE MARKERS
print('`````````````````````````````````````````````````')
```

```
|||||||||||||||||||||||||||||||||||||||||||||||
```




```
`````````````````````````````````````````````````
```

[18]:
```
# PLEASE DO NOT MODIFY THE MARKERS
print('|||||||||||||||||||||||||||||||||||||||||||||||')
```
```

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

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

# Compute and display the accuracy
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
# PLEASE DO NOT MODIFY THE MARKERS
print('~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~')
```

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

Inline Question 3

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

YourAnswer : 5.

YourExplanation : 1. The decision boundary will not always linear, it might be compose by some line segment but it is not linear in general. 2. No, it depnd on case, every cases features are different. 3. No, it depnd on case, every cases features are different. 4. we have already get a function when we are training the model, when we are testing we just uses that function, so the time should be same.

svm

October 8, 2022

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

# Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment1/'
FOLDERNAME = 'enpm809K/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 drive/My\ Drive/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/enpm809K/assignments/assignment1/cs231n/datasets
/content/drive/My Drive/enpm809K/assignments/assignment1
```

1 Multiclass Support Vector Machine exercise

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

In this exercise you will:

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

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

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

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

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

1.1 CIFAR-10 Data Loading and Preprocessing

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

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

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

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

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

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

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

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

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

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

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

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

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

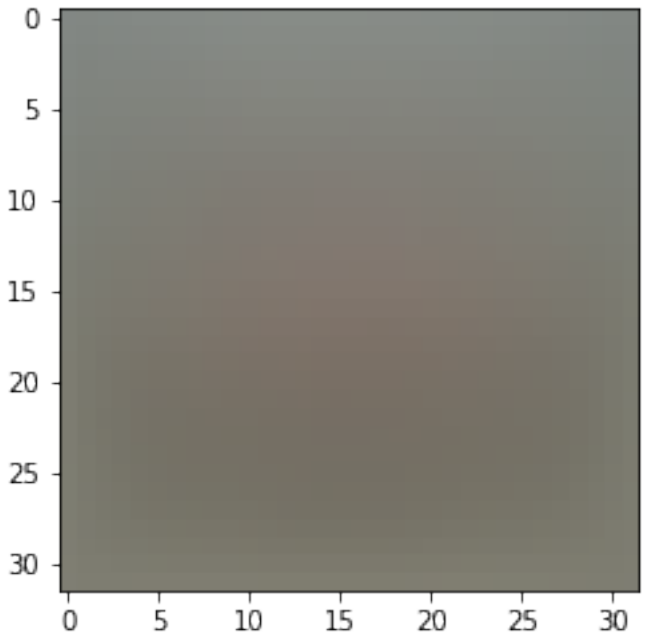
```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

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

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

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

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

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

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

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

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




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

1.2 SVM Classifier

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

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

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

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

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

```
loss: 8.904136
[[-3.27333931e+00 -1.93353930e+01  6.78589257e+00 … -4.67187825e+00
  -6.59472873e+00 -3.96185533e+01]
 [-1.12135798e+01 -1.74266027e+01  7.82395429e+00 … -7.96840184e+00
  -1.75689955e+01 -4.63583745e+01]
 [-2.67006882e+01 -1.63841488e+01  2.49370789e+01 … -8.65357380e+00
```

```
 -4.41642774e+01 -5.96143906e+01]
…
[-1.36205941e+01 -1.57803244e+01  1.26053943e+00 … -4.98303690e+00
  2.19953613e+01 -1.31179453e+01]
[-2.44986963e+01 -1.94643203e+01  8.50675200e+00 …  5.65279057e+00
  7.75529714e-01 -1.73080557e+01]
[ 2.41999999e-01  1.85999999e-01 -1.12000000e-01 … -2.20000007e-02
 -1.64000000e-01 -1.30000000e-01]]
```

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

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

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

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

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

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

```
numerical: -27.707089 analytic: -27.707089, relative error: 1.809176e-12
numerical: -12.525700 analytic: -12.525700, relative error: 9.546409e-12
numerical: -0.762110 analytic: -0.762110, relative error: 2.579298e-10
numerical: 8.340969 analytic: 8.340969, relative error: 9.869029e-12
numerical: -13.482087 analytic: -13.482087, relative error: 6.544625e-12
numerical: 18.453835 analytic: 18.453835, relative error: 4.387219e-12
numerical: -16.449289 analytic: -16.449289, relative error: 1.896950e-12
numerical: -13.677078 analytic: -13.677078, relative error: 2.141402e-11
numerical: 21.299096 analytic: 21.299096, relative error: 9.754531e-12
numerical: 35.660904 analytic: 35.660904, relative error: 5.930172e-12
numerical: -25.659383 analytic: -25.662839, relative error: 6.734074e-05
numerical: -7.195793 analytic: -7.191164, relative error: 3.217579e-04
```

```
numerical: 23.195763 analytic: 23.201134, relative error: 1.157751e-04
numerical: 14.481051 analytic: 14.479396, relative error: 5.717286e-05
numerical: 14.100737 analytic: 14.097847, relative error: 1.024678e-04
numerical: 2.290892 analytic: 2.293784, relative error: 6.308029e-04
numerical: -6.960186 analytic: -6.955122, relative error: 3.638836e-04
numerical: -4.557876 analytic: -4.563017, relative error: 5.636427e-04
numerical: 5.409528 analytic: 5.413148, relative error: 3.344659e-04
numerical: 13.714254 analytic: 13.711304, relative error: 1.075772e-04
```

Inline Question 1

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

YourAnswer : it may be caused by the non differentiable part of the loss function. In the SVM loss function, there have a non-differential point which is in the boundary in the maximum function max(0, sj - syi + 1).

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

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

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

```
Naive loss: 8.904136e+00 computed in 0.189036s
Vectorized loss: 8.904136e+00 computed in 0.014735s
difference: -0.000000
```

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

# The naive implementation and the vectorized implementation should match, but
# the vectorized version should still be much faster.
tic = time.time()
_, grad_naive = svm_loss_naive(W, X_dev, y_dev, 0.000005)
```

```
toc = time.time()
print('Naive loss and gradient: computed in %fs' % (toc - tic))

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

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

```
Naive loss and gradient: computed in 0.211936s
Vectorized loss and gradient: computed in 0.013513s
difference: 0.000000
```

1.2.1 Stochastic Gradient Descent

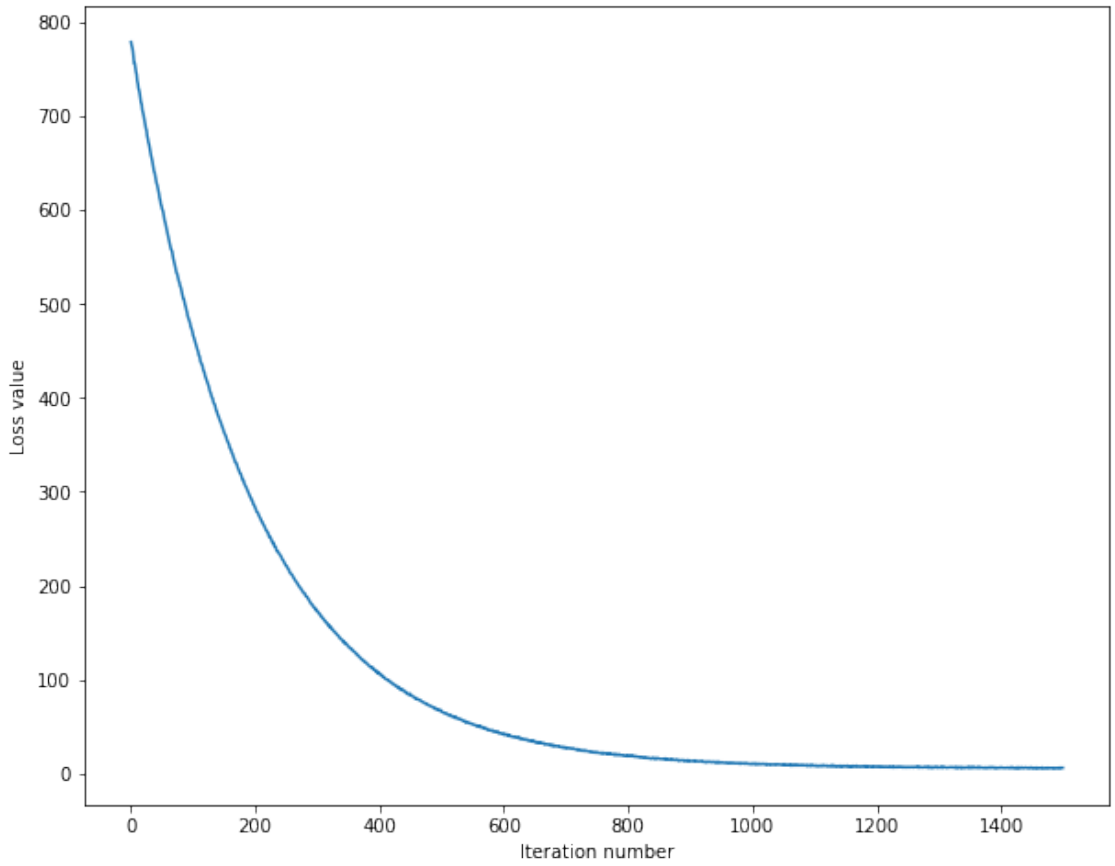
We now have vectorized and efficient expressions for the loss, the gradient and our gradient matches the numerical gradient. We are therefore ready to do SGD to minimize the loss. Your code for this part will be written inside `cs231n/classifiers/linear_classifier.py`.

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

```
iteration 0 / 1500: loss 778.936917
iteration 100 / 1500: loss 466.187691
iteration 200 / 1500: loss 282.658076
iteration 300 / 1500: loss 172.367355
iteration 400 / 1500: loss 106.372559
iteration 500 / 1500: loss 65.800651
iteration 600 / 1500: loss 41.124220
iteration 700 / 1500: loss 26.781205
iteration 800 / 1500: loss 18.813832
iteration 900 / 1500: loss 13.270599
iteration 1000 / 1500: loss 10.413194
iteration 1100 / 1500: loss 8.458446
iteration 1200 / 1500: loss 6.880468
```

```
iteration 1300 / 1500: loss 6.434385
iteration 1400 / 1500: loss 6.485557
That took 11.290267s
```

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



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

```
training accuracy: 0.384469
validation accuracy: 0.397000
```

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

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

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

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

# In this part, I am trying to tune the parameters which are learning rate and␣
 ↪regularization strength.
# The process of validation is describe as follows: First, randomly select the␣
 ↪learning rate and regularization strength.
# Use SVM to train this model, get the predict by using the trained weight,
# one thing I found in this validation is when the learning rate are too large␣
 ↪like 5e-5 the loss will not converge well.
# Find out the best value (which means that it has the highest validation␣
 ↪accuracy)

# Provided as a reference. You may or may not want to change these␣
 ↪hyperparameters
learning_rates = [1e-9, 5e-9, 1e-8, 5e-8, 1e-7, 5e-7]
regularization_strengths = [2.5e4, 3.5e4, 4.5e4, 5.5e4, 6.5e4, 7.5e4]
```

```
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
svm = LinearSVM()
# Use for loop to go throught every parameters

for learning_rate in learning_rates:
  for regularization_strength in regularization_strengths:

    #tic = time.time()
    loss_hist = svm.train(X_train, y_train, learning_rate=learning_rate,␣
 ↪reg=regularization_strength,
                          num_iters=1500, verbose=True)
    y_train_pred = svm.predict(X_train)
    # print('training accuracy: %f' % (np.mean(y_train == y_train_pred), ))
    training_accuracy = np.mean(y_train == y_train_pred)
    y_val_pred = svm.predict(X_val)
    # print('validation accuracy: %f' % (np.mean(y_val == y_val_pred), ))
    validation_accuracy = np.mean(y_val == y_val_pred)
    results[(learning_rate, regularization_strength)] = [training_accuracy,␣
 ↪validation_accuracy]

    if(validation_accuracy > best_val):
      best_val = validation_accuracy
      best_svm = svm


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

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

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

```
iteration 0 / 1500: loss 787.659376
iteration 100 / 1500: loss 782.960090
iteration 200 / 1500: loss 777.195623
iteration 300 / 1500: loss 770.620772
iteration 400 / 1500: loss 765.096143
iteration 500 / 1500: loss 761.594985
iteration 600 / 1500: loss 758.516538
iteration 700 / 1500: loss 753.344505
iteration 800 / 1500: loss 748.095006
iteration 900 / 1500: loss 745.329548
iteration 1000 / 1500: loss 738.527858
```

```
iteration 1100 / 1500: loss 734.391371
iteration 1200 / 1500: loss 732.836657
iteration 1300 / 1500: loss 725.932968
iteration 1400 / 1500: loss 725.467680
iteration 0 / 1500: loss 1002.916888
iteration 100 / 1500: loss 995.703170
iteration 200 / 1500: loss 988.878237
iteration 300 / 1500: loss 979.337439
iteration 400 / 1500: loss 972.354197
iteration 500 / 1500: loss 965.108496
iteration 600 / 1500: loss 960.535582
iteration 700 / 1500: loss 953.793857
iteration 800 / 1500: loss 945.628820
iteration 900 / 1500: loss 940.382642
iteration 1000 / 1500: loss 934.158440
iteration 1100 / 1500: loss 926.823694
iteration 1200 / 1500: loss 918.932034
iteration 1300 / 1500: loss 912.231023
iteration 1400 / 1500: loss 906.525818
iteration 0 / 1500: loss 1154.925579
iteration 100 / 1500: loss 1143.270138
iteration 200 / 1500: loss 1132.716519
iteration 300 / 1500: loss 1121.776997
iteration 400 / 1500: loss 1113.616764
iteration 500 / 1500: loss 1101.981161
iteration 600 / 1500: loss 1092.963401
iteration 700 / 1500: loss 1083.425322
iteration 800 / 1500: loss 1073.689761
iteration 900 / 1500: loss 1063.132671
iteration 1000 / 1500: loss 1052.565692
iteration 1100 / 1500: loss 1045.838540
iteration 1200 / 1500: loss 1033.981891
iteration 1300 / 1500: loss 1025.927699
iteration 1400 / 1500: loss 1015.478653
iteration 0 / 1500: loss 1227.234748
iteration 100 / 1500: loss 1214.789499
iteration 200 / 1500: loss 1202.739429
iteration 300 / 1500: loss 1186.840668
iteration 400 / 1500: loss 1175.174747
iteration 500 / 1500: loss 1162.320810
iteration 600 / 1500: loss 1149.142291
iteration 700 / 1500: loss 1137.336424
iteration 800 / 1500: loss 1123.872236
iteration 900 / 1500: loss 1112.124159
iteration 1000 / 1500: loss 1100.040778
iteration 1100 / 1500: loss 1087.524593
iteration 1200 / 1500: loss 1076.987233
iteration 1300 / 1500: loss 1064.986209
```

```
iteration 1400 / 1500: loss 1052.873249
iteration 0 / 1500: loss 1229.380004
iteration 100 / 1500: loss 1212.038492
iteration 200 / 1500: loss 1196.890819
iteration 300 / 1500: loss 1181.583619
iteration 400 / 1500: loss 1167.328797
iteration 500 / 1500: loss 1150.832685
iteration 600 / 1500: loss 1136.305615
iteration 700 / 1500: loss 1121.118986
iteration 800 / 1500: loss 1107.813421
iteration 900 / 1500: loss 1092.127152
iteration 1000 / 1500: loss 1078.848693
iteration 1100 / 1500: loss 1063.920782
iteration 1200 / 1500: loss 1052.884781
iteration 1300 / 1500: loss 1037.084920
iteration 1400 / 1500: loss 1024.181488
iteration 0 / 1500: loss 1167.125143
iteration 100 / 1500: loss 1148.301332
iteration 200 / 1500: loss 1130.747751
iteration 300 / 1500: loss 1113.810496
iteration 400 / 1500: loss 1097.882729
iteration 500 / 1500: loss 1080.902224
iteration 600 / 1500: loss 1064.967240
iteration 700 / 1500: loss 1049.198727
iteration 800 / 1500: loss 1032.842005
iteration 900 / 1500: loss 1017.152095
iteration 1000 / 1500: loss 1002.967217
iteration 1100 / 1500: loss 988.337354
iteration 1200 / 1500: loss 972.576635
iteration 1300 / 1500: loss 957.534049
iteration 1400 / 1500: loss 944.378113
iteration 0 / 1500: loss 316.204831
iteration 100 / 1500: loss 308.593630
iteration 200 / 1500: loss 300.059803
iteration 300 / 1500: loss 291.948911
iteration 400 / 1500: loss 284.632428
iteration 500 / 1500: loss 278.112792
iteration 600 / 1500: loss 272.007039
iteration 700 / 1500: loss 264.269870
iteration 800 / 1500: loss 257.945224
iteration 900 / 1500: loss 252.366765
iteration 1000 / 1500: loss 246.444298
iteration 1100 / 1500: loss 239.601041
iteration 1200 / 1500: loss 233.739566
iteration 1300 / 1500: loss 227.868098
iteration 1400 / 1500: loss 222.442940
iteration 0 / 1500: loss 301.272797
iteration 100 / 1500: loss 290.982382
```

```
iteration 200 / 1500: loss 281.269721
iteration 300 / 1500: loss 271.241116
iteration 400 / 1500: loss 262.089384
iteration 500 / 1500: loss 253.662557
iteration 600 / 1500: loss 245.187412
iteration 700 / 1500: loss 236.623867
iteration 800 / 1500: loss 228.357027
iteration 900 / 1500: loss 221.734860
iteration 1000 / 1500: loss 213.721161
iteration 1100 / 1500: loss 206.420189
iteration 1200 / 1500: loss 199.511805
iteration 1300 / 1500: loss 192.409539
iteration 1400 / 1500: loss 185.157549
iteration 0 / 1500: loss 229.177628
iteration 100 / 1500: loss 219.444411
iteration 200 / 1500: loss 209.654405
iteration 300 / 1500: loss 200.383950
iteration 400 / 1500: loss 191.349285
iteration 500 / 1500: loss 184.442695
iteration 600 / 1500: loss 175.826986
iteration 700 / 1500: loss 168.101487
iteration 800 / 1500: loss 160.603415
iteration 900 / 1500: loss 153.878074
iteration 1000 / 1500: loss 146.984221
iteration 1100 / 1500: loss 141.066775
iteration 1200 / 1500: loss 135.417382
iteration 1300 / 1500: loss 129.256368
iteration 1400 / 1500: loss 124.391894
iteration 0 / 1500: loss 143.604462
iteration 100 / 1500: loss 136.354496
iteration 200 / 1500: loss 129.365465
iteration 300 / 1500: loss 122.729608
iteration 400 / 1500: loss 116.035096
iteration 500 / 1500: loss 110.189729
iteration 600 / 1500: loss 104.547802
iteration 700 / 1500: loss 99.368400
iteration 800 / 1500: loss 94.075168
iteration 900 / 1500: loss 89.159181
iteration 1000 / 1500: loss 85.481606
iteration 1100 / 1500: loss 80.347739
iteration 1200 / 1500: loss 76.403944
iteration 1300 / 1500: loss 73.017791
iteration 1400 / 1500: loss 69.502389
iteration 0 / 1500: loss 77.268072
iteration 100 / 1500: loss 72.334388
iteration 200 / 1500: loss 68.554778
iteration 300 / 1500: loss 64.736676
iteration 400 / 1500: loss 60.301044
```

```
iteration 500 / 1500: loss 57.676722
iteration 600 / 1500: loss 53.720104
iteration 700 / 1500: loss 50.763615
iteration 800 / 1500: loss 47.964306
iteration 900 / 1500: loss 45.306591
iteration 1000 / 1500: loss 43.472377
iteration 1100 / 1500: loss 40.525105
iteration 1200 / 1500: loss 38.122046
iteration 1300 / 1500: loss 36.003317
iteration 1400 / 1500: loss 34.461188
iteration 0 / 1500: loss 36.428176
iteration 100 / 1500: loss 34.161655
iteration 200 / 1500: loss 32.665580
iteration 300 / 1500: loss 30.930044
iteration 400 / 1500: loss 29.034955
iteration 500 / 1500: loss 27.189206
iteration 600 / 1500: loss 25.243840
iteration 700 / 1500: loss 23.940326
iteration 800 / 1500: loss 22.695123
iteration 900 / 1500: loss 22.037246
iteration 1000 / 1500: loss 20.435262
iteration 1100 / 1500: loss 19.119231
iteration 1200 / 1500: loss 18.189332
iteration 1300 / 1500: loss 17.578475
iteration 1400 / 1500: loss 17.085492
iteration 0 / 1500: loss 8.160489
iteration 100 / 1500: loss 8.474968
iteration 200 / 1500: loss 8.288829
iteration 300 / 1500: loss 7.846597
iteration 400 / 1500: loss 7.983377
iteration 500 / 1500: loss 7.495720
iteration 600 / 1500: loss 7.643050
iteration 700 / 1500: loss 8.036247
iteration 800 / 1500: loss 7.281273
iteration 900 / 1500: loss 7.431820
iteration 1000 / 1500: loss 7.328205
iteration 1100 / 1500: loss 7.164886
iteration 1200 / 1500: loss 7.725279
iteration 1300 / 1500: loss 7.128634
iteration 1400 / 1500: loss 7.642412
iteration 0 / 1500: loss 7.480844
iteration 100 / 1500: loss 7.298648
iteration 200 / 1500: loss 6.917718
iteration 300 / 1500: loss 7.368821
iteration 400 / 1500: loss 7.383925
iteration 500 / 1500: loss 7.049615
iteration 600 / 1500: loss 7.002390
iteration 700 / 1500: loss 7.465329
```

```
iteration 800 / 1500: loss 6.923547
iteration 900 / 1500: loss 6.599746
iteration 1000 / 1500: loss 6.638041
iteration 1100 / 1500: loss 6.510086
iteration 1200 / 1500: loss 6.742422
iteration 1300 / 1500: loss 6.337948
iteration 1400 / 1500: loss 6.161507
iteration 0 / 1500: loss 6.898917
iteration 100 / 1500: loss 6.415628
iteration 200 / 1500: loss 6.231588
iteration 300 / 1500: loss 6.928227
iteration 400 / 1500: loss 5.832561
iteration 500 / 1500: loss 5.859897
iteration 600 / 1500: loss 6.496042
iteration 700 / 1500: loss 6.256090
iteration 800 / 1500: loss 6.158716
iteration 900 / 1500: loss 5.714514
iteration 1000 / 1500: loss 6.107710
iteration 1100 / 1500: loss 6.276277
iteration 1200 / 1500: loss 5.826104
iteration 1300 / 1500: loss 5.886191
iteration 1400 / 1500: loss 6.406161
iteration 0 / 1500: loss 5.915974
iteration 100 / 1500: loss 6.598230
iteration 200 / 1500: loss 5.675544
iteration 300 / 1500: loss 5.484614
iteration 400 / 1500: loss 6.225405
iteration 500 / 1500: loss 6.077644
iteration 600 / 1500: loss 5.835489
iteration 700 / 1500: loss 5.793406
iteration 800 / 1500: loss 6.659486
iteration 900 / 1500: loss 6.046747
iteration 1000 / 1500: loss 6.320469
iteration 1100 / 1500: loss 6.164161
iteration 1200 / 1500: loss 6.068745
iteration 1300 / 1500: loss 5.508424
iteration 1400 / 1500: loss 5.757660
iteration 0 / 1500: loss 6.099243
iteration 100 / 1500: loss 5.772218
iteration 200 / 1500: loss 5.350363
iteration 300 / 1500: loss 5.664013
iteration 400 / 1500: loss 6.082802
iteration 500 / 1500: loss 6.354056
iteration 600 / 1500: loss 5.705770
iteration 700 / 1500: loss 5.473440
iteration 800 / 1500: loss 5.613191
iteration 900 / 1500: loss 5.432830
iteration 1000 / 1500: loss 6.210414
```

```
iteration 1100 / 1500: loss 5.796281
iteration 1200 / 1500: loss 5.467240
iteration 1300 / 1500: loss 5.365167
iteration 1400 / 1500: loss 5.852032
iteration 0 / 1500: loss 6.350132
iteration 100 / 1500: loss 5.891378
iteration 200 / 1500: loss 6.377298
iteration 300 / 1500: loss 5.764241
iteration 400 / 1500: loss 6.372149
iteration 500 / 1500: loss 6.040058
iteration 600 / 1500: loss 5.926021
iteration 700 / 1500: loss 5.790834
iteration 800 / 1500: loss 5.729251
iteration 900 / 1500: loss 6.132419
iteration 1000 / 1500: loss 6.172461
iteration 1100 / 1500: loss 5.961906
iteration 1200 / 1500: loss 6.017709
iteration 1300 / 1500: loss 5.805294
iteration 1400 / 1500: loss 6.086078
iteration 0 / 1500: loss 5.112174
iteration 100 / 1500: loss 5.033915
iteration 200 / 1500: loss 5.336169
iteration 300 / 1500: loss 5.240643
iteration 400 / 1500: loss 5.150757
iteration 500 / 1500: loss 5.067821
iteration 600 / 1500: loss 4.941894
iteration 700 / 1500: loss 5.250671
iteration 800 / 1500: loss 5.234870
iteration 900 / 1500: loss 5.166161
iteration 1000 / 1500: loss 4.915432
iteration 1100 / 1500: loss 5.144922
iteration 1200 / 1500: loss 5.993435
iteration 1300 / 1500: loss 5.246103
iteration 1400 / 1500: loss 5.158891
iteration 0 / 1500: loss 5.248001
iteration 100 / 1500: loss 5.816553
iteration 200 / 1500: loss 5.723832
iteration 300 / 1500: loss 5.603524
iteration 400 / 1500: loss 5.718415
iteration 500 / 1500: loss 5.353221
iteration 600 / 1500: loss 5.656932
iteration 700 / 1500: loss 5.382375
iteration 800 / 1500: loss 5.389795
iteration 900 / 1500: loss 5.068310
iteration 1000 / 1500: loss 5.674361
iteration 1100 / 1500: loss 5.606440
iteration 1200 / 1500: loss 5.993516
iteration 1300 / 1500: loss 5.010754
```

```
iteration 1400 / 1500: loss 5.417732
iteration 0 / 1500: loss 5.405369
iteration 100 / 1500: loss 5.740318
iteration 200 / 1500: loss 5.271312
iteration 300 / 1500: loss 5.575479
iteration 400 / 1500: loss 5.915573
iteration 500 / 1500: loss 5.805289
iteration 600 / 1500: loss 6.105365
iteration 700 / 1500: loss 4.982567
iteration 800 / 1500: loss 5.714959
iteration 900 / 1500: loss 5.848690
iteration 1000 / 1500: loss 6.140554
iteration 1100 / 1500: loss 6.055007
iteration 1200 / 1500: loss 5.319155
iteration 1300 / 1500: loss 5.580278
iteration 1400 / 1500: loss 5.552724
iteration 0 / 1500: loss 6.032905
iteration 100 / 1500: loss 5.754847
iteration 200 / 1500: loss 5.664005
iteration 300 / 1500: loss 5.758336
iteration 400 / 1500: loss 5.609686
iteration 500 / 1500: loss 5.631267
iteration 600 / 1500: loss 6.133343
iteration 700 / 1500: loss 5.606622
iteration 800 / 1500: loss 6.115944
iteration 900 / 1500: loss 5.449351
iteration 1000 / 1500: loss 5.842723
iteration 1100 / 1500: loss 6.021578
iteration 1200 / 1500: loss 5.525585
iteration 1300 / 1500: loss 5.373303
iteration 1400 / 1500: loss 5.268236
iteration 0 / 1500: loss 6.303463
iteration 100 / 1500: loss 5.831152
iteration 200 / 1500: loss 5.929351
iteration 300 / 1500: loss 5.712893
iteration 400 / 1500: loss 5.719580
iteration 500 / 1500: loss 6.120919
iteration 600 / 1500: loss 5.791369
iteration 700 / 1500: loss 6.163642
iteration 800 / 1500: loss 6.438767
iteration 900 / 1500: loss 5.699244
iteration 1000 / 1500: loss 6.355747
iteration 1100 / 1500: loss 5.747299
iteration 1200 / 1500: loss 6.007136
iteration 1300 / 1500: loss 6.262420
iteration 1400 / 1500: loss 5.448335
iteration 0 / 1500: loss 6.041170
iteration 100 / 1500: loss 6.561115
```

```
iteration 200 / 1500: loss 5.809983
iteration 300 / 1500: loss 6.176453
iteration 400 / 1500: loss 6.032488
iteration 500 / 1500: loss 5.880078
iteration 600 / 1500: loss 6.143577
iteration 700 / 1500: loss 5.886784
iteration 800 / 1500: loss 5.905080
iteration 900 / 1500: loss 5.748801
iteration 1000 / 1500: loss 6.044068
iteration 1100 / 1500: loss 5.713444
iteration 1200 / 1500: loss 5.959733
iteration 1300 / 1500: loss 6.066072
iteration 1400 / 1500: loss 6.592383
iteration 0 / 1500: loss 5.596748
iteration 100 / 1500: loss 5.179291
iteration 200 / 1500: loss 5.103406
iteration 300 / 1500: loss 5.664805
iteration 400 / 1500: loss 5.596860
iteration 500 / 1500: loss 5.507707
iteration 600 / 1500: loss 5.447977
iteration 700 / 1500: loss 5.473208
iteration 800 / 1500: loss 5.863411
iteration 900 / 1500: loss 5.259177
iteration 1000 / 1500: loss 5.389342
iteration 1100 / 1500: loss 5.073044
iteration 1200 / 1500: loss 5.030245
iteration 1300 / 1500: loss 5.272606
iteration 1400 / 1500: loss 5.155934
iteration 0 / 1500: loss 5.224181
iteration 100 / 1500: loss 5.587951
iteration 200 / 1500: loss 5.576656
iteration 300 / 1500: loss 5.829596
iteration 400 / 1500: loss 5.027698
iteration 500 / 1500: loss 5.949093
iteration 600 / 1500: loss 5.428047
iteration 700 / 1500: loss 5.457145
iteration 800 / 1500: loss 5.593372
iteration 900 / 1500: loss 5.487601
iteration 1000 / 1500: loss 5.783559
iteration 1100 / 1500: loss 6.078228
iteration 1200 / 1500: loss 5.701381
iteration 1300 / 1500: loss 5.275495
iteration 1400 / 1500: loss 5.950936
iteration 0 / 1500: loss 5.852539
iteration 100 / 1500: loss 5.551624
iteration 200 / 1500: loss 5.663105
iteration 300 / 1500: loss 6.081620
iteration 400 / 1500: loss 5.538521
```

```
iteration 500 / 1500: loss 5.340450
iteration 600 / 1500: loss 5.590391
iteration 700 / 1500: loss 5.586404
iteration 800 / 1500: loss 5.510365
iteration 900 / 1500: loss 5.567697
iteration 1000 / 1500: loss 5.866998
iteration 1100 / 1500: loss 5.598649
iteration 1200 / 1500: loss 5.105848
iteration 1300 / 1500: loss 5.744317
iteration 1400 / 1500: loss 5.630303
iteration 0 / 1500: loss 5.574017
iteration 100 / 1500: loss 5.860414
iteration 200 / 1500: loss 6.019987
iteration 300 / 1500: loss 6.221375
iteration 400 / 1500: loss 5.810257
iteration 500 / 1500: loss 5.865122
iteration 600 / 1500: loss 5.520234
iteration 700 / 1500: loss 6.038840
iteration 800 / 1500: loss 6.117215
iteration 900 / 1500: loss 5.016873
iteration 1000 / 1500: loss 6.122325
iteration 1100 / 1500: loss 5.840970
iteration 1200 / 1500: loss 6.367596
iteration 1300 / 1500: loss 5.957641
iteration 1400 / 1500: loss 5.993522
iteration 0 / 1500: loss 5.705213
iteration 100 / 1500: loss 5.960042
iteration 200 / 1500: loss 6.180754
iteration 300 / 1500: loss 6.209606
iteration 400 / 1500: loss 6.132268
iteration 500 / 1500: loss 5.796621
iteration 600 / 1500: loss 5.769372
iteration 700 / 1500: loss 6.337530
iteration 800 / 1500: loss 5.533560
iteration 900 / 1500: loss 6.110537
iteration 1000 / 1500: loss 6.080002
iteration 1100 / 1500: loss 6.044992
iteration 1200 / 1500: loss 5.583205
iteration 1300 / 1500: loss 5.547355
iteration 1400 / 1500: loss 6.061466
iteration 0 / 1500: loss 6.161080
iteration 100 / 1500: loss 6.084375
iteration 200 / 1500: loss 6.454549
iteration 300 / 1500: loss 5.936678
iteration 400 / 1500: loss 5.782026
iteration 500 / 1500: loss 6.477850
iteration 600 / 1500: loss 6.278933
iteration 700 / 1500: loss 6.182928
```

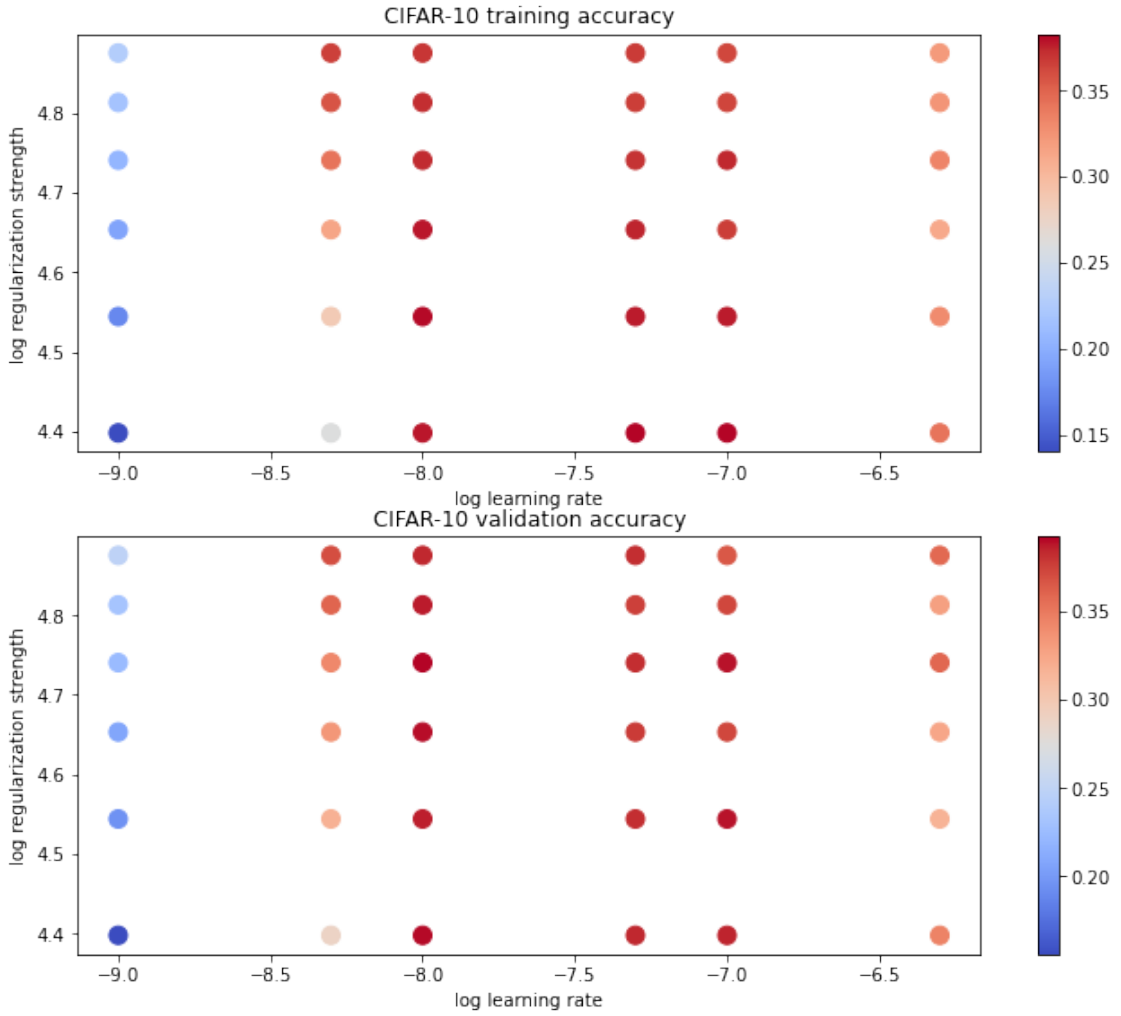
```
iteration 800 / 1500: loss 5.911648
iteration 900 / 1500: loss 6.154627
iteration 1000 / 1500: loss 6.254027
iteration 1100 / 1500: loss 5.757256
iteration 1200 / 1500: loss 5.825580
iteration 1300 / 1500: loss 5.392234
iteration 1400 / 1500: loss 6.127904
iteration 0 / 1500: loss 5.057480
iteration 100 / 1500: loss 6.019369
iteration 200 / 1500: loss 5.564521
iteration 300 / 1500: loss 5.595051
iteration 400 / 1500: loss 6.342313
iteration 500 / 1500: loss 6.015494
iteration 600 / 1500: loss 5.926138
iteration 700 / 1500: loss 6.148504
iteration 800 / 1500: loss 6.026622
iteration 900 / 1500: loss 5.675284
iteration 1000 / 1500: loss 5.566850
iteration 1100 / 1500: loss 6.388514
iteration 1200 / 1500: loss 6.222696
iteration 1300 / 1500: loss 5.260691
iteration 1400 / 1500: loss 5.644814
iteration 0 / 1500: loss 6.114030
iteration 100 / 1500: loss 5.603176
iteration 200 / 1500: loss 6.695318
iteration 300 / 1500: loss 6.704455
iteration 400 / 1500: loss 6.023516
iteration 500 / 1500: loss 6.455630
iteration 600 / 1500: loss 5.622246
iteration 700 / 1500: loss 6.432193
iteration 800 / 1500: loss 6.268696
iteration 900 / 1500: loss 6.306666
iteration 1000 / 1500: loss 6.389107
iteration 1100 / 1500: loss 5.730780
iteration 1200 / 1500: loss 5.773958
iteration 1300 / 1500: loss 5.626641
iteration 1400 / 1500: loss 5.043934
iteration 0 / 1500: loss 6.162228
iteration 100 / 1500: loss 5.833731
iteration 200 / 1500: loss 5.837942
iteration 300 / 1500: loss 6.174423
iteration 400 / 1500: loss 6.275722
iteration 500 / 1500: loss 6.499743
iteration 600 / 1500: loss 6.203391
iteration 700 / 1500: loss 6.149074
iteration 800 / 1500: loss 6.044197
iteration 900 / 1500: loss 6.455055
iteration 1000 / 1500: loss 6.258653
```

```
iteration 1100 / 1500: loss 6.422010
iteration 1200 / 1500: loss 6.581107
iteration 1300 / 1500: loss 6.220246
iteration 1400 / 1500: loss 6.361550
iteration 0 / 1500: loss 6.385035
iteration 100 / 1500: loss 6.431194
iteration 200 / 1500: loss 5.795418
iteration 300 / 1500: loss 6.943221
iteration 400 / 1500: loss 6.062030
iteration 500 / 1500: loss 6.618889
iteration 600 / 1500: loss 6.582032
iteration 700 / 1500: loss 6.401175
iteration 800 / 1500: loss 5.948019
iteration 900 / 1500: loss 6.393462
iteration 1000 / 1500: loss 6.228562
iteration 1100 / 1500: loss 7.227506
iteration 1200 / 1500: loss 6.732370
iteration 1300 / 1500: loss 6.293243
iteration 1400 / 1500: loss 7.003625
iteration 0 / 1500: loss 6.429633
iteration 100 / 1500: loss 6.167103
iteration 200 / 1500: loss 6.626209
iteration 300 / 1500: loss 6.146097
iteration 400 / 1500: loss 6.755705
iteration 500 / 1500: loss 5.783188
iteration 600 / 1500: loss 6.287666
iteration 700 / 1500: loss 6.816501
iteration 800 / 1500: loss 7.234640
iteration 900 / 1500: loss 6.438616
iteration 1000 / 1500: loss 7.381626
iteration 1100 / 1500: loss 6.610672
iteration 1200 / 1500: loss 6.239345
iteration 1300 / 1500: loss 6.490204
iteration 1400 / 1500: loss 6.453124
iteration 0 / 1500: loss 6.586760
iteration 100 / 1500: loss 6.247700
iteration 200 / 1500: loss 7.259163
iteration 300 / 1500: loss 7.084655
iteration 400 / 1500: loss 6.451802
iteration 500 / 1500: loss 7.017799
iteration 600 / 1500: loss 6.126013
iteration 700 / 1500: loss 6.221536
iteration 800 / 1500: loss 6.147605
iteration 900 / 1500: loss 7.039432
iteration 1000 / 1500: loss 6.046547
iteration 1100 / 1500: loss 6.365562
iteration 1200 / 1500: loss 6.869846
iteration 1300 / 1500: loss 6.348618
```

```
iteration 1400 / 1500: loss 6.680755
lr 1.000000e-09 reg 2.500000e+04 train accuracy: 0.140367 val accuracy: 0.156000
lr 1.000000e-09 reg 3.500000e+04 train accuracy: 0.175347 val accuracy: 0.195000
lr 1.000000e-09 reg 4.500000e+04 train accuracy: 0.193327 val accuracy: 0.208000
lr 1.000000e-09 reg 5.500000e+04 train accuracy: 0.206327 val accuracy: 0.224000
lr 1.000000e-09 reg 6.500000e+04 train accuracy: 0.217163 val accuracy: 0.232000
lr 1.000000e-09 reg 7.500000e+04 train accuracy: 0.227735 val accuracy: 0.249000
lr 5.000000e-09 reg 2.500000e+04 train accuracy: 0.260857 val accuracy: 0.289000
lr 5.000000e-09 reg 3.500000e+04 train accuracy: 0.286449 val accuracy: 0.318000
lr 5.000000e-09 reg 4.500000e+04 train accuracy: 0.314857 val accuracy: 0.334000
lr 5.000000e-09 reg 5.500000e+04 train accuracy: 0.341980 val accuracy: 0.343000
lr 5.000000e-09 reg 6.500000e+04 train accuracy: 0.357367 val accuracy: 0.360000
lr 5.000000e-09 reg 7.500000e+04 train accuracy: 0.365633 val accuracy: 0.370000
lr 1.000000e-08 reg 2.500000e+04 train accuracy: 0.378592 val accuracy: 0.391000
lr 1.000000e-08 reg 3.500000e+04 train accuracy: 0.381490 val accuracy: 0.386000
lr 1.000000e-08 reg 4.500000e+04 train accuracy: 0.378020 val accuracy: 0.390000
lr 1.000000e-08 reg 5.500000e+04 train accuracy: 0.373592 val accuracy: 0.392000
lr 1.000000e-08 reg 6.500000e+04 train accuracy: 0.373020 val accuracy: 0.387000
lr 1.000000e-08 reg 7.500000e+04 train accuracy: 0.369551 val accuracy: 0.384000
lr 5.000000e-08 reg 2.500000e+04 train accuracy: 0.382408 val accuracy: 0.384000
lr 5.000000e-08 reg 3.500000e+04 train accuracy: 0.377469 val accuracy: 0.382000
lr 5.000000e-08 reg 4.500000e+04 train accuracy: 0.375857 val accuracy: 0.378000
lr 5.000000e-08 reg 5.500000e+04 train accuracy: 0.370469 val accuracy: 0.382000
lr 5.000000e-08 reg 6.500000e+04 train accuracy: 0.367204 val accuracy: 0.376000
lr 5.000000e-08 reg 7.500000e+04 train accuracy: 0.368286 val accuracy: 0.382000
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.382612 val accuracy: 0.385000
lr 1.000000e-07 reg 3.500000e+04 train accuracy: 0.377082 val accuracy: 0.389000
lr 1.000000e-07 reg 4.500000e+04 train accuracy: 0.366633 val accuracy: 0.373000
lr 1.000000e-07 reg 5.500000e+04 train accuracy: 0.374347 val accuracy: 0.389000
lr 1.000000e-07 reg 6.500000e+04 train accuracy: 0.364408 val accuracy: 0.373000
lr 1.000000e-07 reg 7.500000e+04 train accuracy: 0.363204 val accuracy: 0.366000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.342143 val accuracy: 0.345000
lr 5.000000e-07 reg 3.500000e+04 train accuracy: 0.329796 val accuracy: 0.317000
lr 5.000000e-07 reg 4.500000e+04 train accuracy: 0.312102 val accuracy: 0.325000
lr 5.000000e-07 reg 5.500000e+04 train accuracy: 0.333531 val accuracy: 0.359000
lr 5.000000e-07 reg 6.500000e+04 train accuracy: 0.324184 val accuracy: 0.329000
lr 5.000000e-07 reg 7.500000e+04 train accuracy: 0.322000 val accuracy: 0.358000
best validation accuracy achieved during cross-validation: 0.392000
```

[ ]:
```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()
```

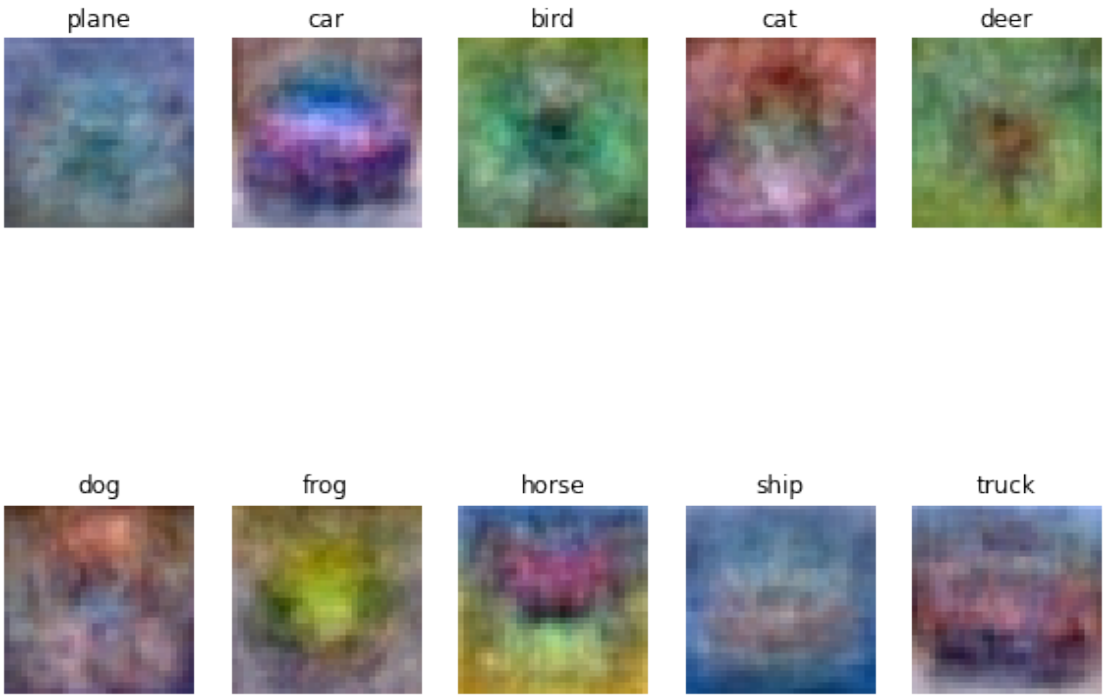


[ ]:
```
# Evaluate the best svm on test set
y_test_pred = best_svm.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
```

```
linear SVM on raw pixels final test set accuracy: 0.338000
```

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

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

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



Inline question 2

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

YourAnswer : The visualize of the weight looks like what the object is but has a little bit blur. The reason of why it looks like the way it does is because we find out the best weight, which means that the weight has the lowest loss (the correct class's score has higher score and the incorrect class has lower score. Lower score means every pixels are more like the object. Second, the reason of why every weight are blur is because every training data are different and the accuracy are not perfect, it just close to 40%.

softmax

October 8, 2022

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

# Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment1/'
FOLDERNAME = 'enpm809K/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 drive/My\ Drive/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/enpm809K/assignments/assignment1/cs231n/datasets
/content/drive/My Drive/enpm809K/assignments/assignment1
```

1 Softmax exercise

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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


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

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

```
dev data shape:  (500, 3073)
dev labels shape:  (500,)
```

1.1 Softmax Classifier

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

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

from cs231n.classifiers.softmax import softmax_loss_naive
import time

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

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

```
loss: 2.469815
sanity check: 2.302585
```

Inline Question 1

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

YourAnswer : First, we know that Loss function in Softmax is possibility function. We do not train the model, and the weights W is a random matrix for now. Therefore, the classification is just like guessing now. Based on above observation, the value inside the -np.log should be 1/(number of classes) which is 10.

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

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

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

```
numerical: -3.250531 analytic: -3.250531, relative error: 3.351042e-09
numerical: -1.574255 analytic: -1.574255, relative error: 3.354255e-09
numerical: -0.995299 analytic: -0.995299, relative error: 3.309711e-08
numerical: 2.792144 analytic: 2.792144, relative error: 7.590450e-09
numerical: 0.260011 analytic: 0.260011, relative error: 2.023113e-07
numerical: -0.988330 analytic: -0.988330, relative error: 4.836705e-08
numerical: 1.378052 analytic: 1.378052, relative error: 1.457192e-08
numerical: 1.125141 analytic: 1.125141, relative error: 8.785996e-09
numerical: 3.623642 analytic: 3.623642, relative error: 2.536629e-08
numerical: 6.661819 analytic: 6.661819, relative error: 1.208727e-08
(3073, 10)
numerical: -1.731881 analytic: -1.730719, relative error: 3.357562e-04
numerical: -3.042197 analytic: -3.044863, relative error: 4.380664e-04
numerical: -2.028764 analytic: -2.026918, relative error: 4.552748e-04
numerical: 2.044328 analytic: 2.046750, relative error: 5.919414e-04
numerical: -0.844539 analytic: -0.843651, relative error: 5.262406e-04
numerical: 0.821608 analytic: 0.826001, relative error: 2.666179e-03
numerical: 1.909018 analytic: 1.900276, relative error: 2.294930e-03
numerical: -1.330162 analytic: -1.323141, relative error: 2.646181e-03
numerical: -0.975505 analytic: -0.977410, relative error: 9.752263e-04
numerical: 2.870515 analytic: 2.869067, relative error: 2.522714e-04
```

[6]:
```python
# Now that we have a naive implementation of the softmax loss function and its␣
↪gradient,
# implement a vectorized version in softmax_loss_vectorized.
# The two versions should compute the same results, but the vectorized version␣
↪should be
# much faster.
tic = time.time()
loss_naive, grad_naive = softmax_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('naive loss: %e computed in %fs' % (loss_naive, toc - tic))

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

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

```
naive loss: 2.469815e+00 computed in 0.138875s
vectorized loss: 2.469815e+00 computed in 0.023427s
2.469814584150278 2.46981458415028
Loss difference: 0.000000
Gradient difference: 0.000000
```

[7]:
```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of over 0.35 on the validation set.

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

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

# Provided as a reference. You may or may not want to change these␣
 ↪hyperparameters
learning_rates = [1e-9, 5e-9, 1e-8, 5e-8, 1e-7, 5e-7]
regularization_strengths = [2.5e4, 3.5e4, 4.5e4, 5.5e4, 6.5e4, 7.5e4]

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

# In this part, I am trying to tune the parameters which are learning rate and␣
 ↪regularization strength.
# The process of validation is describe as follows: First, randomly select the␣
 ↪learning rate and regularization strength.
# Use for loop to go throught every parameters
# Use softmax to train this model, get the predict by using the trained weight,
# Find out the best value (which means that it has the highest validation␣
 ↪accuracy)


softmax = Softmax()
for learning_rate in learning_rates:
  for regularization_strength in regularization_strengths:

    #tic = time.time()
    loss_hist = softmax.train(X_train, y_train, learning_rate=learning_rate,␣
 ↪reg=regularization_strength,
```

```
                          num_iters=1500)
    y_train_pred = softmax.predict(X_train)
    # print('training accuracy: %f' % (np.mean(y_train == y_train_pred), ))
    training_accuracy = 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), ))
    validation_accuracy = np.mean(y_val == y_val_pred)
    results[(learning_rate, regularization_strength)] = [training_accuracy,␣
 ↪validation_accuracy]

    if(validation_accuracy > best_val):
      best_val = validation_accuracy
      best_softmax = softmax

#pass

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

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

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

```
lr 1.000000e-09 reg 2.500000e+04 train accuracy: 0.125082 val accuracy: 0.131000
lr 1.000000e-09 reg 3.500000e+04 train accuracy: 0.133816 val accuracy: 0.138000
lr 1.000000e-09 reg 4.500000e+04 train accuracy: 0.141755 val accuracy: 0.149000
lr 1.000000e-09 reg 5.500000e+04 train accuracy: 0.148673 val accuracy: 0.161000
lr 1.000000e-09 reg 6.500000e+04 train accuracy: 0.154592 val accuracy: 0.173000
lr 1.000000e-09 reg 7.500000e+04 train accuracy: 0.159449 val accuracy: 0.176000
lr 5.000000e-09 reg 2.500000e+04 train accuracy: 0.183694 val accuracy: 0.220000
lr 5.000000e-09 reg 3.500000e+04 train accuracy: 0.211184 val accuracy: 0.250000
lr 5.000000e-09 reg 4.500000e+04 train accuracy: 0.240592 val accuracy: 0.266000
lr 5.000000e-09 reg 5.500000e+04 train accuracy: 0.270714 val accuracy: 0.291000
lr 5.000000e-09 reg 6.500000e+04 train accuracy: 0.294796 val accuracy: 0.306000
lr 5.000000e-09 reg 7.500000e+04 train accuracy: 0.310735 val accuracy: 0.327000
lr 1.000000e-08 reg 2.500000e+04 train accuracy: 0.331653 val accuracy: 0.344000
lr 1.000000e-08 reg 3.500000e+04 train accuracy: 0.339449 val accuracy: 0.354000
lr 1.000000e-08 reg 4.500000e+04 train accuracy: 0.337143 val accuracy: 0.354000
lr 1.000000e-08 reg 5.500000e+04 train accuracy: 0.331224 val accuracy: 0.341000
lr 1.000000e-08 reg 6.500000e+04 train accuracy: 0.324592 val accuracy: 0.339000
lr 1.000000e-08 reg 7.500000e+04 train accuracy: 0.321143 val accuracy: 0.331000
lr 5.000000e-08 reg 2.500000e+04 train accuracy: 0.349510 val accuracy: 0.359000
lr 5.000000e-08 reg 3.500000e+04 train accuracy: 0.344143 val accuracy: 0.355000
```

```
lr 5.000000e-08 reg 4.500000e+04 train accuracy: 0.336878 val accuracy: 0.349000
lr 5.000000e-08 reg 5.500000e+04 train accuracy: 0.324694 val accuracy: 0.344000
lr 5.000000e-08 reg 6.500000e+04 train accuracy: 0.316510 val accuracy: 0.335000
lr 5.000000e-08 reg 7.500000e+04 train accuracy: 0.316551 val accuracy: 0.325000
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.353102 val accuracy: 0.361000
lr 1.000000e-07 reg 3.500000e+04 train accuracy: 0.338959 val accuracy: 0.353000
lr 1.000000e-07 reg 4.500000e+04 train accuracy: 0.336714 val accuracy: 0.355000
lr 1.000000e-07 reg 5.500000e+04 train accuracy: 0.322776 val accuracy: 0.335000
lr 1.000000e-07 reg 6.500000e+04 train accuracy: 0.322245 val accuracy: 0.332000
lr 1.000000e-07 reg 7.500000e+04 train accuracy: 0.324449 val accuracy: 0.342000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.351816 val accuracy: 0.359000
lr 5.000000e-07 reg 3.500000e+04 train accuracy: 0.336531 val accuracy: 0.356000
lr 5.000000e-07 reg 4.500000e+04 train accuracy: 0.325714 val accuracy: 0.334000
lr 5.000000e-07 reg 5.500000e+04 train accuracy: 0.322939 val accuracy: 0.332000
lr 5.000000e-07 reg 6.500000e+04 train accuracy: 0.319122 val accuracy: 0.338000
lr 5.000000e-07 reg 7.500000e+04 train accuracy: 0.318490 val accuracy: 0.325000
best validation accuracy achieved during cross-validation: 0.361000
```

[8]:
```
# evaluate on test set
# Evaluate the best softmax on test set
y_test_pred = best_softmax.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('softmax on raw pixels final test set accuracy: %f' % (test_accuracy, ))
```

```
softmax on raw pixels final test set accuracy: 0.327000
```

Inline Question 2 - *True or False*

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

YourAnswer : False

YourExplanation : The Softmax classifier loss function is the sum of the correct class possibility, and the possibility can be 1, and then the log of possibility of the correct score will be zero. Therefore, if this happen in the new data, then it will not change when we add a new data.

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

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

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

    # Rescale the weights to be between 0 and 255
```

```
    wimg = 255.0 * (w[:, :, :, i].squeeze() - w_min) / (w_max - w_min)
    plt.imshow(wimg.astype('uint8'))
    plt.axis('off')
    plt.title(classes[i])
```

[9]:

two_layer_net

October 8, 2022

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

# Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment1/'
FOLDERNAME = 'enpm809K/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 drive/My\ Drive/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/enpm809K/assignments/assignment1/cs231n/datasets
/content/drive/My Drive/enpm809K/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.

[2]:
```
# As usual, a bit of setup
from __future__ import print_function
import time
import numpy as np
import matplotlib.pyplot as plt
from cs231n.classifiers.fc_net import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient,␣
 ↪eval_numerical_gradient_array
from cs231n.solver import Solver

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

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

def rel_error(x, y):
```

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

[3]:
```
# Load the (preprocessed) CIFAR10 data.

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

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

2 Affine layer: forward

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

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

[4]:
```
# Test the affine_forward function

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

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

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

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

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

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

3 Affine layer: backward

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

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

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

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

# 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:

[6]:
```
# Test the relu_forward function

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

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

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

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

5 ReLU activation: backward

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

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

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

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

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

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

5.1 Inline Question 1:

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

5.2 Answer:

1. The equation of the backpropagation of Sigmoid function is dx = (1/(1+e^x)*(1-1/(1+e^x))), when 1-1/(1+e^x) becomes zero then the dx will becomes 0, on the other hand when the x is too small the slope is also close to zero. So when x is too large and too small the dx will approach 0.

2. The backpropagation of ReLU function is different when x > 0. When x is larger than 0 the dx is a constant, and when the x is smaller than 0 the dx is 0.

6 "Sandwich" layers

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

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

[8]:
```
from cs231n.layer_utils import affine_relu_forward, affine_relu_backward
np.random.seed(231)
x = np.random.randn(2, 3, 4)
w = np.random.randn(12, 10)
b = np.random.randn(10)
dout = np.random.randn(2, 10)

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

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

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

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

7 Loss layers: Softmax and SVM

Now implement the loss and gradient for softmax and SVM in the `softmax_loss` and `svm_loss` function in `cs231n/layers.py`. These should be similar to what you implemented in `cs231n/classifiers/softmax.py` and `cs231n/classifiers/linear_svm.py`.

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

[9]:
```
np.random.seed(231)
num_classes, num_inputs = 10, 50
x = 0.001 * np.random.randn(num_inputs, num_classes)
y = np.random.randint(num_classes, size=num_inputs)

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

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

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

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

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

Testing softmax_loss:
loss:  2.302545844500738
dx error:  1.0948706063428788e-08
```

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.

[10]:
```
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: 3.61e-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. You also need to imeplement the `sgd` function in `cs231n/optim.py`. After doing so, use a `Solver` instance to train a `TwoLayerNet` that achieves about `36%` accuracy on the validation set.

[11]:
```
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
model = TwoLayerNet(input_size, hidden_size, num_classes)
solver = None

################################################################################
# TODO: Use a Solver instance to train a TwoLayerNet that achieves about 36% #
# accuracy on the validation set.                                              #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

# use the Solver function based on the comment in the solver.py
# for some unknown reason this accuracy is above 50%
solver = Solver(model, data,
                update_rule='sgd',
                optim_config={
                'learning_rate': 1e-3,
                },
                lr_decay=0.95,
                num_epochs=10, batch_size=100,
                print_every=100)
```

```
solver.train()

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

```
(Iteration 1 / 4900) loss: 2.300089
(Epoch 0 / 10) train acc: 0.171000; val_acc: 0.170000
(Iteration 101 / 4900) loss: 1.782419
(Iteration 201 / 4900) loss: 1.803466
(Iteration 301 / 4900) loss: 1.712676
(Iteration 401 / 4900) loss: 1.693946
(Epoch 1 / 10) train acc: 0.399000; val_acc: 0.428000
(Iteration 501 / 4900) loss: 1.711237
(Iteration 601 / 4900) loss: 1.443683
(Iteration 701 / 4900) loss: 1.574904
(Iteration 801 / 4900) loss: 1.526751
(Iteration 901 / 4900) loss: 1.352554
(Epoch 2 / 10) train acc: 0.463000; val_acc: 0.443000
(Iteration 1001 / 4900) loss: 1.340071
(Iteration 1101 / 4900) loss: 1.386843
(Iteration 1201 / 4900) loss: 1.489919
(Iteration 1301 / 4900) loss: 1.353077
(Iteration 1401 / 4900) loss: 1.467951
(Epoch 3 / 10) train acc: 0.477000; val_acc: 0.430000
(Iteration 1501 / 4900) loss: 1.337133
(Iteration 1601 / 4900) loss: 1.426819
(Iteration 1701 / 4900) loss: 1.348675
(Iteration 1801 / 4900) loss: 1.412626
(Iteration 1901 / 4900) loss: 1.354764
(Epoch 4 / 10) train acc: 0.497000; val_acc: 0.467000
(Iteration 2001 / 4900) loss: 1.422221
(Iteration 2101 / 4900) loss: 1.360665
(Iteration 2201 / 4900) loss: 1.546539
(Iteration 2301 / 4900) loss: 1.196704
(Iteration 2401 / 4900) loss: 1.401958
(Epoch 5 / 10) train acc: 0.508000; val_acc: 0.483000
(Iteration 2501 / 4900) loss: 1.243567
(Iteration 2601 / 4900) loss: 1.513808
(Iteration 2701 / 4900) loss: 1.266416
(Iteration 2801 / 4900) loss: 1.485956
(Iteration 2901 / 4900) loss: 1.049706
(Epoch 6 / 10) train acc: 0.533000; val_acc: 0.492000
(Iteration 3001 / 4900) loss: 1.264002
(Iteration 3101 / 4900) loss: 1.184786
(Iteration 3201 / 4900) loss: 1.231162
```

```
(Iteration 3301 / 4900) loss: 1.353725
(Iteration 3401 / 4900) loss: 1.217830
(Epoch 7 / 10) train acc: 0.545000; val_acc: 0.489000
(Iteration 3501 / 4900) loss: 1.446003
(Iteration 3601 / 4900) loss: 1.152495
(Iteration 3701 / 4900) loss: 1.421970
(Iteration 3801 / 4900) loss: 1.222752
(Iteration 3901 / 4900) loss: 1.213468
(Epoch 8 / 10) train acc: 0.546000; val_acc: 0.474000
(Iteration 4001 / 4900) loss: 1.187435
(Iteration 4101 / 4900) loss: 1.284799
(Iteration 4201 / 4900) loss: 1.135251
(Iteration 4301 / 4900) loss: 1.212217
(Iteration 4401 / 4900) loss: 1.213544
(Epoch 9 / 10) train acc: 0.586000; val_acc: 0.486000
(Iteration 4501 / 4900) loss: 1.306174
(Iteration 4601 / 4900) loss: 1.213528
(Iteration 4701 / 4900) loss: 1.220260
(Iteration 4801 / 4900) loss: 1.233231
(Epoch 10 / 10) train acc: 0.545000; val_acc: 0.483000
```

10 Debug the training

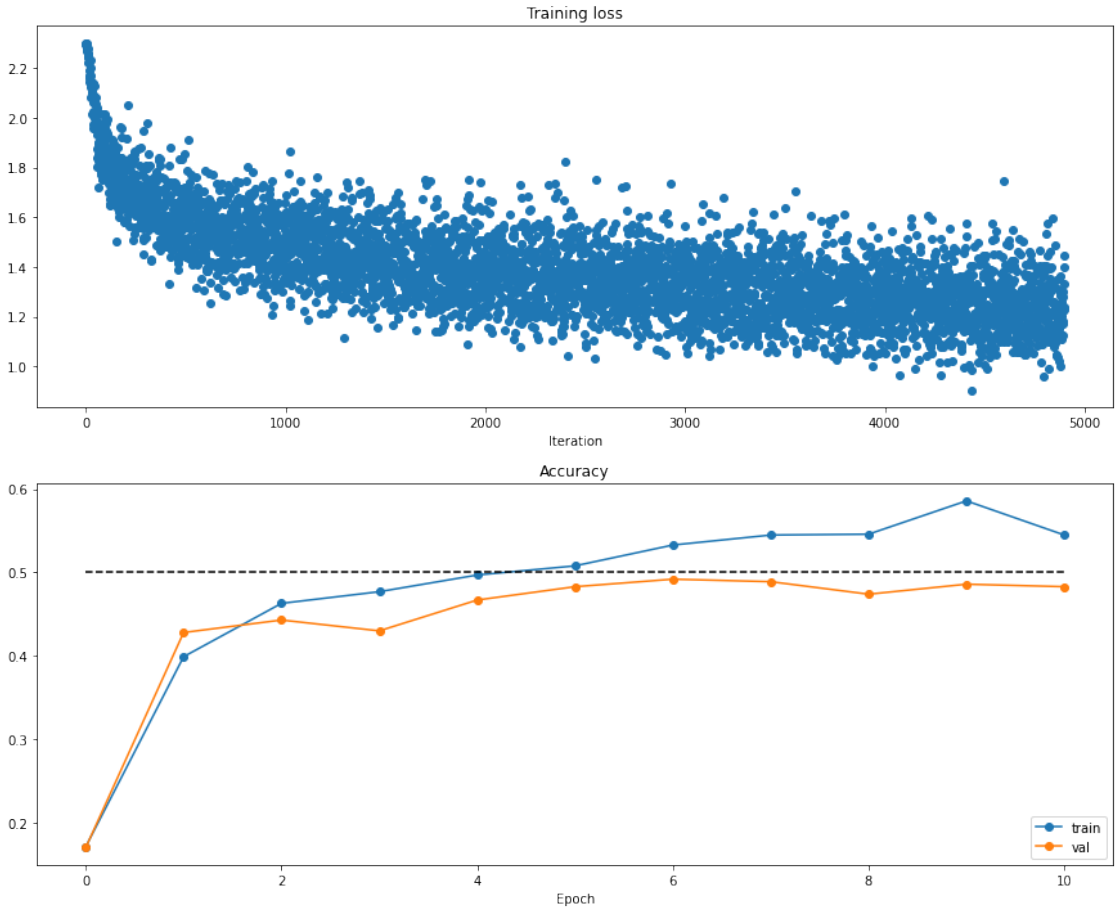
With the default parameters we provided above, you should get a validation accuracy of about 0.36 on the validation set. This isn't very good.

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.

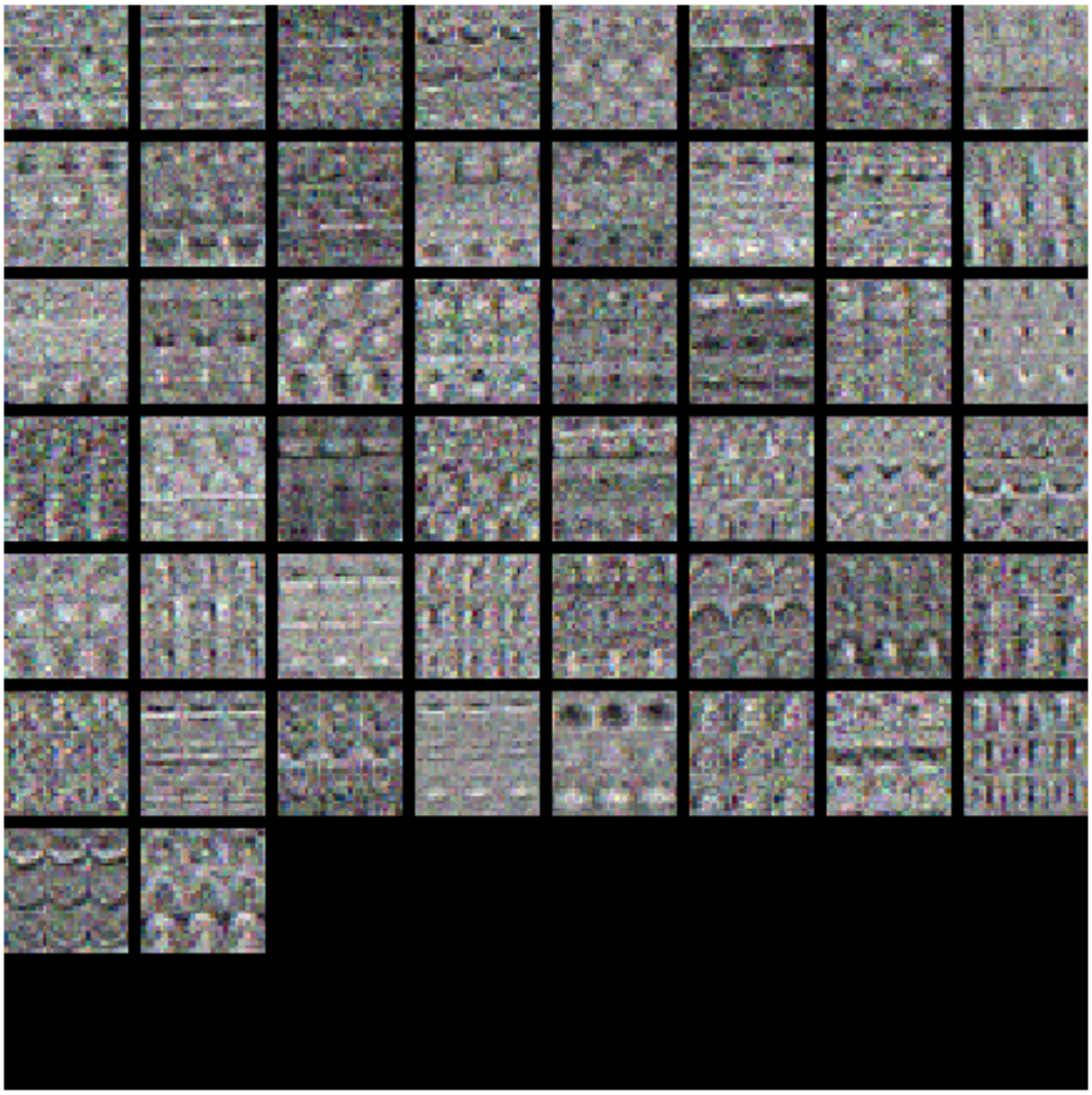
Another strategy is to visualize the weights that were learned in the first layer of the network. In most neural networks trained on visual data, the first layer weights typically show some visible structure when visualized.

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



[13]:
```
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(32, 32, 3, -1).transpose(3, 0, 1, 2)
    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.).

[14]:
```
best_model = None


################################################################################
# TODO: Tune hyperparameters using the validation set. Store your best trained␣
↪#
# model in best_model.                                                         ␣
↪#
#                                                                              ␣
↪#
# To help debug your network, it may help to use visualizations similar to the␣
↪#
# ones we used above; these visualizations will have significant qualitative   ␣
↪#
# differences from the ones we saw above for the poorly tuned network.         ␣
↪#
#                                                                              ␣
↪#
# Tweaking hyperparameters by hand can be fun, but you might find it useful to␣
↪#
# write code to sweep through possible combinations of hyperparameters         ␣
↪#
# automatically like we did on thexs previous exercises.                       ␣
↪  #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
# finding good parameters
hss = [50, 75, 100]
lrs = [1e-5, 1e-4, 1e-3]
lr_decs = [1, 0.75]
# define necessary parameters
input_size = 32 * 32 * 3
num_classes = 10
best = 0

# Find the best parameters by go through every pairs of parameters
```

```
for hs in hss:
  for lr in lrs:
    for lr_dec in lr_decs:
      model = TwoLayerNet(input_size, hs, num_classes)
      solver = Solver(model, data,
                      update_rule='sgd',
                      optim_config={
                      'learning_rate': lr,
                      },
                      lr_decay=lr_dec,
                      num_epochs=10, batch_size=100,
                      print_every=100)
      # train model
      solver.train()
      # find best model
      if solver.best_val_acc > best:
        best = solver.best_val_acc
        best_model = model
#pass
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
################################################################################
#                              END OF YOUR CODE                                #
################################################################################
```

```
(Iteration 1 / 4900) loss: 2.303135
(Epoch 0 / 10) train acc: 0.083000; val_acc: 0.073000
(Iteration 101 / 4900) loss: 2.301192
(Iteration 201 / 4900) loss: 2.298792
(Iteration 301 / 4900) loss: 2.295644
(Iteration 401 / 4900) loss: 2.291433
(Epoch 1 / 10) train acc: 0.195000; val_acc: 0.172000
(Iteration 501 / 4900) loss: 2.290656
(Iteration 601 / 4900) loss: 2.286551
(Iteration 701 / 4900) loss: 2.279994
(Iteration 801 / 4900) loss: 2.279667
(Iteration 901 / 4900) loss: 2.272517
(Epoch 2 / 10) train acc: 0.206000; val_acc: 0.197000
(Iteration 1001 / 4900) loss: 2.269366
(Iteration 1101 / 4900) loss: 2.253118
(Iteration 1201 / 4900) loss: 2.235787
(Iteration 1301 / 4900) loss: 2.234800
(Iteration 1401 / 4900) loss: 2.196863
(Epoch 3 / 10) train acc: 0.225000; val_acc: 0.196000
(Iteration 1501 / 4900) loss: 2.210367
(Iteration 1601 / 4900) loss: 2.152881
(Iteration 1701 / 4900) loss: 2.180424
(Iteration 1801 / 4900) loss: 2.209705
```

```
(Iteration 1901 / 4900) loss: 2.125786
(Epoch 4 / 10) train acc: 0.218000; val_acc: 0.219000
(Iteration 2001 / 4900) loss: 2.186521
(Iteration 2101 / 4900) loss: 2.102440
(Iteration 2201 / 4900) loss: 2.142097
(Iteration 2301 / 4900) loss: 2.214205
(Iteration 2401 / 4900) loss: 2.165366
(Epoch 5 / 10) train acc: 0.241000; val_acc: 0.252000
(Iteration 2501 / 4900) loss: 2.122819
(Iteration 2601 / 4900) loss: 2.067654
(Iteration 2701 / 4900) loss: 2.125412
(Iteration 2801 / 4900) loss: 2.114933
(Iteration 2901 / 4900) loss: 2.073151
(Epoch 6 / 10) train acc: 0.243000; val_acc: 0.272000
(Iteration 3001 / 4900) loss: 2.094084
(Iteration 3101 / 4900) loss: 2.094512
(Iteration 3201 / 4900) loss: 2.085144
(Iteration 3301 / 4900) loss: 2.106553
(Iteration 3401 / 4900) loss: 2.123425
(Epoch 7 / 10) train acc: 0.289000; val_acc: 0.276000
(Iteration 3501 / 4900) loss: 2.031932
(Iteration 3601 / 4900) loss: 2.037255
(Iteration 3701 / 4900) loss: 1.997135
(Iteration 3801 / 4900) loss: 2.040605
(Iteration 3901 / 4900) loss: 2.098358
(Epoch 8 / 10) train acc: 0.284000; val_acc: 0.291000
(Iteration 4001 / 4900) loss: 2.068198
(Iteration 4101 / 4900) loss: 1.917346
(Iteration 4201 / 4900) loss: 1.812494
(Iteration 4301 / 4900) loss: 2.076228
(Iteration 4401 / 4900) loss: 2.033967
(Epoch 9 / 10) train acc: 0.297000; val_acc: 0.292000
(Iteration 4501 / 4900) loss: 1.943162
(Iteration 4601 / 4900) loss: 1.940023
(Iteration 4701 / 4900) loss: 1.882549
(Iteration 4801 / 4900) loss: 1.934029
(Epoch 10 / 10) train acc: 0.295000; val_acc: 0.292000
(Iteration 1 / 4900) loss: 2.302277
(Epoch 0 / 10) train acc: 0.097000; val_acc: 0.098000
(Iteration 101 / 4900) loss: 2.299535
(Iteration 201 / 4900) loss: 2.297895
(Iteration 301 / 4900) loss: 2.295901
(Iteration 401 / 4900) loss: 2.292283
(Epoch 1 / 10) train acc: 0.185000; val_acc: 0.173000
(Iteration 501 / 4900) loss: 2.287644
(Iteration 601 / 4900) loss: 2.284352
(Iteration 701 / 4900) loss: 2.282445
(Iteration 801 / 4900) loss: 2.285294
```

```
(Iteration 901 / 4900) loss: 2.265436
(Epoch 2 / 10) train acc: 0.188000; val_acc: 0.212000
(Iteration 1001 / 4900) loss: 2.268468
(Iteration 1101 / 4900) loss: 2.265923
(Iteration 1201 / 4900) loss: 2.259522
(Iteration 1301 / 4900) loss: 2.265115
(Iteration 1401 / 4900) loss: 2.253925
(Epoch 3 / 10) train acc: 0.214000; val_acc: 0.218000
(Iteration 1501 / 4900) loss: 2.239574
(Iteration 1601 / 4900) loss: 2.230586
(Iteration 1701 / 4900) loss: 2.249664
(Iteration 1801 / 4900) loss: 2.209075
(Iteration 1901 / 4900) loss: 2.231665
(Epoch 4 / 10) train acc: 0.211000; val_acc: 0.213000
(Iteration 2001 / 4900) loss: 2.215836
(Iteration 2101 / 4900) loss: 2.199519
(Iteration 2201 / 4900) loss: 2.186741
(Iteration 2301 / 4900) loss: 2.211614
(Iteration 2401 / 4900) loss: 2.180905
(Epoch 5 / 10) train acc: 0.214000; val_acc: 0.221000
(Iteration 2501 / 4900) loss: 2.198437
(Iteration 2601 / 4900) loss: 2.170572
(Iteration 2701 / 4900) loss: 2.222226
(Iteration 2801 / 4900) loss: 2.189285
(Iteration 2901 / 4900) loss: 2.147711
(Epoch 6 / 10) train acc: 0.208000; val_acc: 0.221000
(Iteration 3001 / 4900) loss: 2.216667
(Iteration 3101 / 4900) loss: 2.219913
(Iteration 3201 / 4900) loss: 2.197494
(Iteration 3301 / 4900) loss: 2.206735
(Iteration 3401 / 4900) loss: 2.204764
(Epoch 7 / 10) train acc: 0.202000; val_acc: 0.220000
(Iteration 3501 / 4900) loss: 2.131240
(Iteration 3601 / 4900) loss: 2.193443
(Iteration 3701 / 4900) loss: 2.157426
(Iteration 3801 / 4900) loss: 2.197352
(Iteration 3901 / 4900) loss: 2.193201
(Epoch 8 / 10) train acc: 0.228000; val_acc: 0.223000
(Iteration 4001 / 4900) loss: 2.134444
(Iteration 4101 / 4900) loss: 2.215229
(Iteration 4201 / 4900) loss: 2.194239
(Iteration 4301 / 4900) loss: 2.156017
(Iteration 4401 / 4900) loss: 2.133250
(Epoch 9 / 10) train acc: 0.199000; val_acc: 0.223000
(Iteration 4501 / 4900) loss: 2.112126
(Iteration 4601 / 4900) loss: 2.158050
(Iteration 4701 / 4900) loss: 2.147763
(Iteration 4801 / 4900) loss: 2.184325
```

```
(Epoch 10 / 10) train acc: 0.225000; val_acc: 0.226000
(Iteration 1 / 4900) loss: 2.303642
(Epoch 0 / 10) train acc: 0.090000; val_acc: 0.108000
(Iteration 101 / 4900) loss: 2.271236
(Iteration 201 / 4900) loss: 2.138886
(Iteration 301 / 4900) loss: 2.149988
(Iteration 401 / 4900) loss: 2.004480
(Epoch 1 / 10) train acc: 0.291000; val_acc: 0.301000
(Iteration 501 / 4900) loss: 2.008762
(Iteration 601 / 4900) loss: 1.812849
(Iteration 701 / 4900) loss: 1.854433
(Iteration 801 / 4900) loss: 1.880291
(Iteration 901 / 4900) loss: 1.918236
(Epoch 2 / 10) train acc: 0.343000; val_acc: 0.369000
(Iteration 1001 / 4900) loss: 1.897390
(Iteration 1101 / 4900) loss: 1.787506
(Iteration 1201 / 4900) loss: 1.817863
(Iteration 1301 / 4900) loss: 1.754073
(Iteration 1401 / 4900) loss: 1.655604
(Epoch 3 / 10) train acc: 0.424000; val_acc: 0.408000
(Iteration 1501 / 4900) loss: 1.739480
(Iteration 1601 / 4900) loss: 1.728324
(Iteration 1701 / 4900) loss: 1.894983
(Iteration 1801 / 4900) loss: 1.736069
(Iteration 1901 / 4900) loss: 1.651572
(Epoch 4 / 10) train acc: 0.419000; val_acc: 0.414000
(Iteration 2001 / 4900) loss: 1.631730
(Iteration 2101 / 4900) loss: 1.666616
(Iteration 2201 / 4900) loss: 1.487747
(Iteration 2301 / 4900) loss: 1.502572
(Iteration 2401 / 4900) loss: 1.625814
(Epoch 5 / 10) train acc: 0.438000; val_acc: 0.447000
(Iteration 2501 / 4900) loss: 1.656286
(Iteration 2601 / 4900) loss: 1.740072
(Iteration 2701 / 4900) loss: 1.720000
(Iteration 2801 / 4900) loss: 1.626016
(Iteration 2901 / 4900) loss: 1.677733
(Epoch 6 / 10) train acc: 0.461000; val_acc: 0.441000
(Iteration 3001 / 4900) loss: 1.604723
(Iteration 3101 / 4900) loss: 1.507077
(Iteration 3201 / 4900) loss: 1.383699
(Iteration 3301 / 4900) loss: 1.743330
(Iteration 3401 / 4900) loss: 1.478821
(Epoch 7 / 10) train acc: 0.456000; val_acc: 0.453000
(Iteration 3501 / 4900) loss: 1.434424
(Iteration 3601 / 4900) loss: 1.423199
(Iteration 3701 / 4900) loss: 1.680711
(Iteration 3801 / 4900) loss: 1.449148
```

```
(Iteration 3901 / 4900) loss: 1.399079
(Epoch 8 / 10) train acc: 0.446000; val_acc: 0.459000
(Iteration 4001 / 4900) loss: 1.563450
(Iteration 4101 / 4900) loss: 1.465457
(Iteration 4201 / 4900) loss: 1.540005
(Iteration 4301 / 4900) loss: 1.214625
(Iteration 4401 / 4900) loss: 1.458367
(Epoch 9 / 10) train acc: 0.485000; val_acc: 0.465000
(Iteration 4501 / 4900) loss: 1.548063
(Iteration 4601 / 4900) loss: 1.403109
(Iteration 4701 / 4900) loss: 1.616318
(Iteration 4801 / 4900) loss: 1.597489
(Epoch 10 / 10) train acc: 0.473000; val_acc: 0.462000
(Iteration 1 / 4900) loss: 2.303471
(Epoch 0 / 10) train acc: 0.109000; val_acc: 0.106000
(Iteration 101 / 4900) loss: 2.262982
(Iteration 201 / 4900) loss: 2.227386
(Iteration 301 / 4900) loss: 2.087809
(Iteration 401 / 4900) loss: 2.009308
(Epoch 1 / 10) train acc: 0.307000; val_acc: 0.297000
(Iteration 501 / 4900) loss: 2.075976
(Iteration 601 / 4900) loss: 1.950778
(Iteration 701 / 4900) loss: 1.912723
(Iteration 801 / 4900) loss: 1.921659
(Iteration 901 / 4900) loss: 1.821538
(Epoch 2 / 10) train acc: 0.363000; val_acc: 0.361000
(Iteration 1001 / 4900) loss: 1.841626
(Iteration 1101 / 4900) loss: 1.784906
(Iteration 1201 / 4900) loss: 1.753264
(Iteration 1301 / 4900) loss: 1.823466
(Iteration 1401 / 4900) loss: 1.816009
(Epoch 3 / 10) train acc: 0.360000; val_acc: 0.376000
(Iteration 1501 / 4900) loss: 1.747717
(Iteration 1601 / 4900) loss: 1.804960
(Iteration 1701 / 4900) loss: 1.774429
(Iteration 1801 / 4900) loss: 1.461668
(Iteration 1901 / 4900) loss: 1.739759
(Epoch 4 / 10) train acc: 0.402000; val_acc: 0.389000
(Iteration 2001 / 4900) loss: 1.781121
(Iteration 2101 / 4900) loss: 1.690666
(Iteration 2201 / 4900) loss: 1.666389
(Iteration 2301 / 4900) loss: 1.773360
(Iteration 2401 / 4900) loss: 1.663779
(Epoch 5 / 10) train acc: 0.410000; val_acc: 0.404000
(Iteration 2501 / 4900) loss: 1.757823
(Iteration 2601 / 4900) loss: 1.768775
(Iteration 2701 / 4900) loss: 1.612685
(Iteration 2801 / 4900) loss: 1.693913
```

```
(Iteration 2901 / 4900) loss: 1.654495
(Epoch 6 / 10) train acc: 0.385000; val_acc: 0.413000
(Iteration 3001 / 4900) loss: 1.616625
(Iteration 3101 / 4900) loss: 1.776509
(Iteration 3201 / 4900) loss: 1.672079
(Iteration 3301 / 4900) loss: 1.569846
(Iteration 3401 / 4900) loss: 1.661009
(Epoch 7 / 10) train acc: 0.383000; val_acc: 0.413000
(Iteration 3501 / 4900) loss: 1.602649
(Iteration 3601 / 4900) loss: 1.726197
(Iteration 3701 / 4900) loss: 1.683491
(Iteration 3801 / 4900) loss: 1.725231
(Iteration 3901 / 4900) loss: 1.954722
(Epoch 8 / 10) train acc: 0.390000; val_acc: 0.421000
(Iteration 4001 / 4900) loss: 1.844924
(Iteration 4101 / 4900) loss: 1.582936
(Iteration 4201 / 4900) loss: 1.642752
(Iteration 4301 / 4900) loss: 1.686036
(Iteration 4401 / 4900) loss: 1.730059
(Epoch 9 / 10) train acc: 0.396000; val_acc: 0.427000
(Iteration 4501 / 4900) loss: 1.521767
(Iteration 4601 / 4900) loss: 1.745651
(Iteration 4701 / 4900) loss: 1.674846
(Iteration 4801 / 4900) loss: 1.589742
(Epoch 10 / 10) train acc: 0.438000; val_acc: 0.432000
(Iteration 1 / 4900) loss: 2.303738
(Epoch 0 / 10) train acc: 0.081000; val_acc: 0.081000
(Iteration 101 / 4900) loss: 1.776590
(Iteration 201 / 4900) loss: 1.624259
(Iteration 301 / 4900) loss: 1.512693
(Iteration 401 / 4900) loss: 1.728457
(Epoch 1 / 10) train acc: 0.426000; val_acc: 0.440000
(Iteration 501 / 4900) loss: 1.455381
(Iteration 601 / 4900) loss: 1.492328
(Iteration 701 / 4900) loss: 1.661285
(Iteration 801 / 4900) loss: 1.625128
(Iteration 901 / 4900) loss: 1.537819
(Epoch 2 / 10) train acc: 0.466000; val_acc: 0.446000
(Iteration 1001 / 4900) loss: 1.738284
(Iteration 1101 / 4900) loss: 1.526716
(Iteration 1201 / 4900) loss: 1.454739
(Iteration 1301 / 4900) loss: 1.286785
(Iteration 1401 / 4900) loss: 1.445118
(Epoch 3 / 10) train acc: 0.496000; val_acc: 0.472000
(Iteration 1501 / 4900) loss: 1.694106
(Iteration 1601 / 4900) loss: 1.577496
(Iteration 1701 / 4900) loss: 1.543284
(Iteration 1801 / 4900) loss: 1.346308
```

```
(Iteration 1901 / 4900) loss: 1.470416
(Epoch 4 / 10) train acc: 0.514000; val_acc: 0.498000
(Iteration 2001 / 4900) loss: 1.288914
(Iteration 2101 / 4900) loss: 1.320338
(Iteration 2201 / 4900) loss: 1.285393
(Iteration 2301 / 4900) loss: 1.568407
(Iteration 2401 / 4900) loss: 1.335407
(Epoch 5 / 10) train acc: 0.518000; val_acc: 0.486000
(Iteration 2501 / 4900) loss: 1.500845
(Iteration 2601 / 4900) loss: 1.482741
(Iteration 2701 / 4900) loss: 1.397471
(Iteration 2801 / 4900) loss: 1.254818
(Iteration 2901 / 4900) loss: 1.570639
(Epoch 6 / 10) train acc: 0.555000; val_acc: 0.485000
(Iteration 3001 / 4900) loss: 1.347197
(Iteration 3101 / 4900) loss: 1.346945
(Iteration 3201 / 4900) loss: 1.293326
(Iteration 3301 / 4900) loss: 1.301234
(Iteration 3401 / 4900) loss: 1.264220
(Epoch 7 / 10) train acc: 0.514000; val_acc: 0.474000
(Iteration 3501 / 4900) loss: 1.247572
(Iteration 3601 / 4900) loss: 1.369054
(Iteration 3701 / 4900) loss: 1.324555
(Iteration 3801 / 4900) loss: 1.174047
(Iteration 3901 / 4900) loss: 1.511006
(Epoch 8 / 10) train acc: 0.521000; val_acc: 0.498000
(Iteration 4001 / 4900) loss: 1.477274
(Iteration 4101 / 4900) loss: 1.183175
(Iteration 4201 / 4900) loss: 1.265512
(Iteration 4301 / 4900) loss: 1.440159
(Iteration 4401 / 4900) loss: 1.144707
(Epoch 9 / 10) train acc: 0.554000; val_acc: 0.458000
(Iteration 4501 / 4900) loss: 1.183871
(Iteration 4601 / 4900) loss: 1.450856
(Iteration 4701 / 4900) loss: 1.473342
(Iteration 4801 / 4900) loss: 1.313534
(Epoch 10 / 10) train acc: 0.566000; val_acc: 0.484000
(Iteration 1 / 4900) loss: 2.304534
(Epoch 0 / 10) train acc: 0.125000; val_acc: 0.123000
(Iteration 101 / 4900) loss: 1.748194
(Iteration 201 / 4900) loss: 1.871628
(Iteration 301 / 4900) loss: 1.435446
(Iteration 401 / 4900) loss: 1.726709
(Epoch 1 / 10) train acc: 0.455000; val_acc: 0.457000
(Iteration 501 / 4900) loss: 1.511436
(Iteration 601 / 4900) loss: 1.697467
(Iteration 701 / 4900) loss: 1.486016
(Iteration 801 / 4900) loss: 1.715681
```

```
(Iteration 901 / 4900) loss: 1.473904
(Epoch 2 / 10) train acc: 0.476000; val_acc: 0.479000
(Iteration 1001 / 4900) loss: 1.464975
(Iteration 1101 / 4900) loss: 1.282074
(Iteration 1201 / 4900) loss: 1.673501
(Iteration 1301 / 4900) loss: 1.250401
(Iteration 1401 / 4900) loss: 1.454158
(Epoch 3 / 10) train acc: 0.475000; val_acc: 0.463000
(Iteration 1501 / 4900) loss: 1.423762
(Iteration 1601 / 4900) loss: 1.252824
(Iteration 1701 / 4900) loss: 1.149806
(Iteration 1801 / 4900) loss: 1.324767
(Iteration 1901 / 4900) loss: 1.473352
(Epoch 4 / 10) train acc: 0.528000; val_acc: 0.476000
(Iteration 2001 / 4900) loss: 1.331373
(Iteration 2101 / 4900) loss: 1.245152
(Iteration 2201 / 4900) loss: 1.327973
(Iteration 2301 / 4900) loss: 1.403174
(Iteration 2401 / 4900) loss: 1.265639
(Epoch 5 / 10) train acc: 0.561000; val_acc: 0.515000
(Iteration 2501 / 4900) loss: 1.378190
(Iteration 2601 / 4900) loss: 1.379376
(Iteration 2701 / 4900) loss: 1.410496
(Iteration 2801 / 4900) loss: 1.287688
(Iteration 2901 / 4900) loss: 1.368664
(Epoch 6 / 10) train acc: 0.528000; val_acc: 0.490000
(Iteration 3001 / 4900) loss: 1.308898
(Iteration 3101 / 4900) loss: 1.145268
(Iteration 3201 / 4900) loss: 1.301118
(Iteration 3301 / 4900) loss: 1.278308
(Iteration 3401 / 4900) loss: 1.469732
(Epoch 7 / 10) train acc: 0.553000; val_acc: 0.489000
(Iteration 3501 / 4900) loss: 1.378332
(Iteration 3601 / 4900) loss: 1.207381
(Iteration 3701 / 4900) loss: 1.341003
(Iteration 3801 / 4900) loss: 1.281229
(Iteration 3901 / 4900) loss: 1.121312
(Epoch 8 / 10) train acc: 0.566000; val_acc: 0.502000
(Iteration 4001 / 4900) loss: 1.285847
(Iteration 4101 / 4900) loss: 1.090637
(Iteration 4201 / 4900) loss: 1.120273
(Iteration 4301 / 4900) loss: 1.237613
(Iteration 4401 / 4900) loss: 1.219144
(Epoch 9 / 10) train acc: 0.571000; val_acc: 0.502000
(Iteration 4501 / 4900) loss: 1.214286
(Iteration 4601 / 4900) loss: 1.372042
(Iteration 4701 / 4900) loss: 1.409565
(Iteration 4801 / 4900) loss: 1.154909
```

```
(Epoch 10 / 10) train acc: 0.548000; val_acc: 0.497000
(Iteration 1 / 4900) loss: 2.306000
(Epoch 0 / 10) train acc: 0.099000; val_acc: 0.082000
(Iteration 101 / 4900) loss: 2.299889
(Iteration 201 / 4900) loss: 2.296859
(Iteration 301 / 4900) loss: 2.291309
(Iteration 401 / 4900) loss: 2.290177
(Epoch 1 / 10) train acc: 0.231000; val_acc: 0.235000
(Iteration 501 / 4900) loss: 2.284947
(Iteration 601 / 4900) loss: 2.280248
(Iteration 701 / 4900) loss: 2.277845
(Iteration 801 / 4900) loss: 2.270748
(Iteration 901 / 4900) loss: 2.248017
(Epoch 2 / 10) train acc: 0.240000; val_acc: 0.246000
(Iteration 1001 / 4900) loss: 2.245069
(Iteration 1101 / 4900) loss: 2.228061
(Iteration 1201 / 4900) loss: 2.231108
(Iteration 1301 / 4900) loss: 2.221748
(Iteration 1401 / 4900) loss: 2.191161
(Epoch 3 / 10) train acc: 0.223000; val_acc: 0.246000
(Iteration 1501 / 4900) loss: 2.214161
(Iteration 1601 / 4900) loss: 2.156898
(Iteration 1701 / 4900) loss: 2.191928
(Iteration 1801 / 4900) loss: 2.098251
(Iteration 1901 / 4900) loss: 2.173764
(Epoch 4 / 10) train acc: 0.250000; val_acc: 0.254000
(Iteration 2001 / 4900) loss: 2.124083
(Iteration 2101 / 4900) loss: 2.066515
(Iteration 2201 / 4900) loss: 2.095053
(Iteration 2301 / 4900) loss: 2.125205
(Iteration 2401 / 4900) loss: 2.084268
(Epoch 5 / 10) train acc: 0.263000; val_acc: 0.266000
(Iteration 2501 / 4900) loss: 2.092316
(Iteration 2601 / 4900) loss: 2.055356
(Iteration 2701 / 4900) loss: 2.020230
(Iteration 2801 / 4900) loss: 2.059288
(Iteration 2901 / 4900) loss: 1.984945
(Epoch 6 / 10) train acc: 0.264000; val_acc: 0.278000
(Iteration 3001 / 4900) loss: 2.039536
(Iteration 3101 / 4900) loss: 2.007707
(Iteration 3201 / 4900) loss: 2.105831
(Iteration 3301 / 4900) loss: 2.143806
(Iteration 3401 / 4900) loss: 2.103921
(Epoch 7 / 10) train acc: 0.294000; val_acc: 0.300000
(Iteration 3501 / 4900) loss: 2.037326
(Iteration 3601 / 4900) loss: 2.012278
(Iteration 3701 / 4900) loss: 1.981902
(Iteration 3801 / 4900) loss: 2.017796
```

```
(Iteration 3901 / 4900) loss: 2.065437
(Epoch 8 / 10) train acc: 0.288000; val_acc: 0.299000
(Iteration 4001 / 4900) loss: 1.967375
(Iteration 4101 / 4900) loss: 2.057981
(Iteration 4201 / 4900) loss: 1.941618
(Iteration 4301 / 4900) loss: 1.951748
(Iteration 4401 / 4900) loss: 1.940796
(Epoch 9 / 10) train acc: 0.296000; val_acc: 0.308000
(Iteration 4501 / 4900) loss: 1.890734
(Iteration 4601 / 4900) loss: 1.921161
(Iteration 4701 / 4900) loss: 2.065584
(Iteration 4801 / 4900) loss: 2.009130
(Epoch 10 / 10) train acc: 0.310000; val_acc: 0.318000
(Iteration 1 / 4900) loss: 2.299796
(Epoch 0 / 10) train acc: 0.084000; val_acc: 0.077000
(Iteration 101 / 4900) loss: 2.300447
(Iteration 201 / 4900) loss: 2.297361
(Iteration 301 / 4900) loss: 2.297216
(Iteration 401 / 4900) loss: 2.292715
(Epoch 1 / 10) train acc: 0.166000; val_acc: 0.196000
(Iteration 501 / 4900) loss: 2.285005
(Iteration 601 / 4900) loss: 2.283777
(Iteration 701 / 4900) loss: 2.282387
(Iteration 801 / 4900) loss: 2.287909
(Iteration 901 / 4900) loss: 2.259511
(Epoch 2 / 10) train acc: 0.220000; val_acc: 0.211000
(Iteration 1001 / 4900) loss: 2.273753
(Iteration 1101 / 4900) loss: 2.266860
(Iteration 1201 / 4900) loss: 2.246262
(Iteration 1301 / 4900) loss: 2.267761
(Iteration 1401 / 4900) loss: 2.246879
(Epoch 3 / 10) train acc: 0.209000; val_acc: 0.222000
(Iteration 1501 / 4900) loss: 2.254609
(Iteration 1601 / 4900) loss: 2.251140
(Iteration 1701 / 4900) loss: 2.251241
(Iteration 1801 / 4900) loss: 2.226843
(Iteration 1901 / 4900) loss: 2.226836
(Epoch 4 / 10) train acc: 0.201000; val_acc: 0.225000
(Iteration 2001 / 4900) loss: 2.184539
(Iteration 2101 / 4900) loss: 2.152752
(Iteration 2201 / 4900) loss: 2.218519
(Iteration 2301 / 4900) loss: 2.216809
(Iteration 2401 / 4900) loss: 2.173946
(Epoch 5 / 10) train acc: 0.223000; val_acc: 0.227000
(Iteration 2501 / 4900) loss: 2.184244
(Iteration 2601 / 4900) loss: 2.195188
(Iteration 2701 / 4900) loss: 2.179411
(Iteration 2801 / 4900) loss: 2.201558
```

```
(Iteration 2901 / 4900) loss: 2.169565
(Epoch 6 / 10) train acc: 0.206000; val_acc: 0.231000
(Iteration 3001 / 4900) loss: 2.165765
(Iteration 3101 / 4900) loss: 2.209213
(Iteration 3201 / 4900) loss: 2.197735
(Iteration 3301 / 4900) loss: 2.178776
(Iteration 3401 / 4900) loss: 2.201389
(Epoch 7 / 10) train acc: 0.213000; val_acc: 0.232000
(Iteration 3501 / 4900) loss: 2.169355
(Iteration 3601 / 4900) loss: 2.152643
(Iteration 3701 / 4900) loss: 2.185343
(Iteration 3801 / 4900) loss: 2.181062
(Iteration 3901 / 4900) loss: 2.190953
(Epoch 8 / 10) train acc: 0.221000; val_acc: 0.230000
(Iteration 4001 / 4900) loss: 2.216420
(Iteration 4101 / 4900) loss: 2.170613
(Iteration 4201 / 4900) loss: 2.154836
(Iteration 4301 / 4900) loss: 2.244369
(Iteration 4401 / 4900) loss: 2.259331
(Epoch 9 / 10) train acc: 0.211000; val_acc: 0.232000
(Iteration 4501 / 4900) loss: 2.213027
(Iteration 4601 / 4900) loss: 2.150405
(Iteration 4701 / 4900) loss: 2.148329
(Iteration 4801 / 4900) loss: 2.143803
(Epoch 10 / 10) train acc: 0.204000; val_acc: 0.231000
(Iteration 1 / 4900) loss: 2.306228
(Epoch 0 / 10) train acc: 0.108000; val_acc: 0.107000
(Iteration 101 / 4900) loss: 2.229455
(Iteration 201 / 4900) loss: 2.111482
(Iteration 301 / 4900) loss: 1.925880
(Iteration 401 / 4900) loss: 2.047492
(Epoch 1 / 10) train acc: 0.315000; val_acc: 0.310000
(Iteration 501 / 4900) loss: 1.884078
(Iteration 601 / 4900) loss: 1.839735
(Iteration 701 / 4900) loss: 1.935508
(Iteration 801 / 4900) loss: 1.958707
(Iteration 901 / 4900) loss: 1.797438
(Epoch 2 / 10) train acc: 0.333000; val_acc: 0.370000
(Iteration 1001 / 4900) loss: 1.853332
(Iteration 1101 / 4900) loss: 1.915635
(Iteration 1201 / 4900) loss: 1.876157
(Iteration 1301 / 4900) loss: 1.659963
(Iteration 1401 / 4900) loss: 1.746355
(Epoch 3 / 10) train acc: 0.400000; val_acc: 0.392000
(Iteration 1501 / 4900) loss: 1.620612
(Iteration 1601 / 4900) loss: 1.636658
(Iteration 1701 / 4900) loss: 1.550700
(Iteration 1801 / 4900) loss: 1.803302
```

```
(Iteration 1901 / 4900) loss: 1.625678
(Epoch 4 / 10) train acc: 0.422000; val_acc: 0.430000
(Iteration 2001 / 4900) loss: 1.705856
(Iteration 2101 / 4900) loss: 1.513905
(Iteration 2201 / 4900) loss: 1.568964
(Iteration 2301 / 4900) loss: 1.485267
(Iteration 2401 / 4900) loss: 1.576135
(Epoch 5 / 10) train acc: 0.430000; val_acc: 0.459000
(Iteration 2501 / 4900) loss: 1.683903
(Iteration 2601 / 4900) loss: 1.567725
(Iteration 2701 / 4900) loss: 1.461889
(Iteration 2801 / 4900) loss: 1.604601
(Iteration 2901 / 4900) loss: 1.473137
(Epoch 6 / 10) train acc: 0.454000; val_acc: 0.463000
(Iteration 3001 / 4900) loss: 1.564939
(Iteration 3101 / 4900) loss: 1.674473
(Iteration 3201 / 4900) loss: 1.441439
(Iteration 3301 / 4900) loss: 1.561890
(Iteration 3401 / 4900) loss: 1.603635
(Epoch 7 / 10) train acc: 0.449000; val_acc: 0.464000
(Iteration 3501 / 4900) loss: 1.341160
(Iteration 3601 / 4900) loss: 1.492113
(Iteration 3701 / 4900) loss: 1.423491
(Iteration 3801 / 4900) loss: 1.424816
(Iteration 3901 / 4900) loss: 1.336567
(Epoch 8 / 10) train acc: 0.479000; val_acc: 0.470000
(Iteration 4001 / 4900) loss: 1.375441
(Iteration 4101 / 4900) loss: 1.535867
(Iteration 4201 / 4900) loss: 1.439120
(Iteration 4301 / 4900) loss: 1.437516
(Iteration 4401 / 4900) loss: 1.347025
(Epoch 9 / 10) train acc: 0.470000; val_acc: 0.466000
(Iteration 4501 / 4900) loss: 1.437131
(Iteration 4601 / 4900) loss: 1.343583
(Iteration 4701 / 4900) loss: 1.412569
(Iteration 4801 / 4900) loss: 1.431573
(Epoch 10 / 10) train acc: 0.502000; val_acc: 0.477000
(Iteration 1 / 4900) loss: 2.305062
(Epoch 0 / 10) train acc: 0.135000; val_acc: 0.121000
(Iteration 101 / 4900) loss: 2.249300
(Iteration 201 / 4900) loss: 2.122003
(Iteration 301 / 4900) loss: 2.091080
(Iteration 401 / 4900) loss: 2.012933
(Epoch 1 / 10) train acc: 0.308000; val_acc: 0.311000
(Iteration 501 / 4900) loss: 1.911789
(Iteration 601 / 4900) loss: 1.891960
(Iteration 701 / 4900) loss: 1.899689
(Iteration 801 / 4900) loss: 1.845488
```

```
(Iteration 901 / 4900) loss: 1.840484
(Epoch 2 / 10) train acc: 0.358000; val_acc: 0.353000
(Iteration 1001 / 4900) loss: 1.810849
(Iteration 1101 / 4900) loss: 1.750644
(Iteration 1201 / 4900) loss: 1.792798
(Iteration 1301 / 4900) loss: 1.845098
(Iteration 1401 / 4900) loss: 1.817777
(Epoch 3 / 10) train acc: 0.366000; val_acc: 0.391000
(Iteration 1501 / 4900) loss: 1.701157
(Iteration 1601 / 4900) loss: 1.764765
(Iteration 1701 / 4900) loss: 1.608556
(Iteration 1801 / 4900) loss: 1.737293
(Iteration 1901 / 4900) loss: 1.633861
(Epoch 4 / 10) train acc: 0.394000; val_acc: 0.399000
(Iteration 2001 / 4900) loss: 1.597150
(Iteration 2101 / 4900) loss: 1.685745
(Iteration 2201 / 4900) loss: 1.577013
(Iteration 2301 / 4900) loss: 1.737391
(Iteration 2401 / 4900) loss: 1.797557
(Epoch 5 / 10) train acc: 0.414000; val_acc: 0.397000
(Iteration 2501 / 4900) loss: 1.761845
(Iteration 2601 / 4900) loss: 1.658207
(Iteration 2701 / 4900) loss: 1.728423
(Iteration 2801 / 4900) loss: 1.664903
(Iteration 2901 / 4900) loss: 1.711528
(Epoch 6 / 10) train acc: 0.409000; val_acc: 0.407000
(Iteration 3001 / 4900) loss: 1.741372
(Iteration 3101 / 4900) loss: 1.763378
(Iteration 3201 / 4900) loss: 1.717538
(Iteration 3301 / 4900) loss: 1.744028
(Iteration 3401 / 4900) loss: 1.781439
(Epoch 7 / 10) train acc: 0.406000; val_acc: 0.415000
(Iteration 3501 / 4900) loss: 1.487450
(Iteration 3601 / 4900) loss: 1.567041
(Iteration 3701 / 4900) loss: 1.738243
(Iteration 3801 / 4900) loss: 1.660306
(Iteration 3901 / 4900) loss: 1.714692
(Epoch 8 / 10) train acc: 0.429000; val_acc: 0.421000
(Iteration 4001 / 4900) loss: 1.606516
(Iteration 4101 / 4900) loss: 1.662700
(Iteration 4201 / 4900) loss: 1.622945
(Iteration 4301 / 4900) loss: 1.698195
(Iteration 4401 / 4900) loss: 1.671225
(Epoch 9 / 10) train acc: 0.414000; val_acc: 0.424000
(Iteration 4501 / 4900) loss: 1.683348
(Iteration 4601 / 4900) loss: 1.569639
(Iteration 4701 / 4900) loss: 1.510263
(Iteration 4801 / 4900) loss: 1.663173
```

```
(Epoch 10 / 10) train acc: 0.431000; val_acc: 0.420000
(Iteration 1 / 4900) loss: 2.306113
(Epoch 0 / 10) train acc: 0.104000; val_acc: 0.120000
(Iteration 101 / 4900) loss: 1.745336
(Iteration 201 / 4900) loss: 1.757943
(Iteration 301 / 4900) loss: 1.733483
(Iteration 401 / 4900) loss: 1.647210
(Epoch 1 / 10) train acc: 0.465000; val_acc: 0.441000
(Iteration 501 / 4900) loss: 1.552111
(Iteration 601 / 4900) loss: 1.451791
(Iteration 701 / 4900) loss: 1.726711
(Iteration 801 / 4900) loss: 1.404434
(Iteration 901 / 4900) loss: 1.381414
(Epoch 2 / 10) train acc: 0.480000; val_acc: 0.449000
(Iteration 1001 / 4900) loss: 1.320950
(Iteration 1101 / 4900) loss: 1.313397
(Iteration 1201 / 4900) loss: 1.403283
(Iteration 1301 / 4900) loss: 1.529019
(Iteration 1401 / 4900) loss: 1.730811
(Epoch 3 / 10) train acc: 0.506000; val_acc: 0.449000
(Iteration 1501 / 4900) loss: 1.401560
(Iteration 1601 / 4900) loss: 1.324542
(Iteration 1701 / 4900) loss: 1.512984
(Iteration 1801 / 4900) loss: 1.431365
(Iteration 1901 / 4900) loss: 1.201026
(Epoch 4 / 10) train acc: 0.511000; val_acc: 0.479000
(Iteration 2001 / 4900) loss: 1.407292
(Iteration 2101 / 4900) loss: 1.289127
(Iteration 2201 / 4900) loss: 1.407390
(Iteration 2301 / 4900) loss: 1.341770
(Iteration 2401 / 4900) loss: 1.236038
(Epoch 5 / 10) train acc: 0.531000; val_acc: 0.487000
(Iteration 2501 / 4900) loss: 1.201444
(Iteration 2601 / 4900) loss: 1.300359
(Iteration 2701 / 4900) loss: 1.486642
(Iteration 2801 / 4900) loss: 1.301498
(Iteration 2901 / 4900) loss: 1.305431
(Epoch 6 / 10) train acc: 0.579000; val_acc: 0.519000
(Iteration 3001 / 4900) loss: 1.109481
(Iteration 3101 / 4900) loss: 1.212928
(Iteration 3201 / 4900) loss: 1.274058
(Iteration 3301 / 4900) loss: 1.358510
(Iteration 3401 / 4900) loss: 1.347689
(Epoch 7 / 10) train acc: 0.549000; val_acc: 0.498000
(Iteration 3501 / 4900) loss: 1.166758
(Iteration 3601 / 4900) loss: 1.201540
(Iteration 3701 / 4900) loss: 1.230301
(Iteration 3801 / 4900) loss: 1.288240
```

```
(Iteration 3901 / 4900) loss: 1.508024
(Epoch 8 / 10) train acc: 0.568000; val_acc: 0.489000
(Iteration 4001 / 4900) loss: 1.038700
(Iteration 4101 / 4900) loss: 1.207167
(Iteration 4201 / 4900) loss: 1.411221
(Iteration 4301 / 4900) loss: 1.105594
(Iteration 4401 / 4900) loss: 1.432934
(Epoch 9 / 10) train acc: 0.560000; val_acc: 0.503000
(Iteration 4501 / 4900) loss: 1.173974
(Iteration 4601 / 4900) loss: 1.607968
(Iteration 4701 / 4900) loss: 1.052972
(Iteration 4801 / 4900) loss: 1.227282
(Epoch 10 / 10) train acc: 0.507000; val_acc: 0.445000
(Iteration 1 / 4900) loss: 2.299677
(Epoch 0 / 10) train acc: 0.145000; val_acc: 0.151000
(Iteration 101 / 4900) loss: 1.922200
(Iteration 201 / 4900) loss: 1.578848
(Iteration 301 / 4900) loss: 1.527884
(Iteration 401 / 4900) loss: 1.657380
(Epoch 1 / 10) train acc: 0.438000; val_acc: 0.437000
(Iteration 501 / 4900) loss: 1.480051
(Iteration 601 / 4900) loss: 1.432291
(Iteration 701 / 4900) loss: 1.662131
(Iteration 801 / 4900) loss: 1.381582
(Iteration 901 / 4900) loss: 1.488700
(Epoch 2 / 10) train acc: 0.464000; val_acc: 0.470000
(Iteration 1001 / 4900) loss: 1.100140
(Iteration 1101 / 4900) loss: 1.525602
(Iteration 1201 / 4900) loss: 1.624132
(Iteration 1301 / 4900) loss: 1.609781
(Iteration 1401 / 4900) loss: 1.459342
(Epoch 3 / 10) train acc: 0.498000; val_acc: 0.460000
(Iteration 1501 / 4900) loss: 1.196595
(Iteration 1601 / 4900) loss: 1.298905
(Iteration 1701 / 4900) loss: 1.152505
(Iteration 1801 / 4900) loss: 1.571702
(Iteration 1901 / 4900) loss: 1.033902
(Epoch 4 / 10) train acc: 0.529000; val_acc: 0.493000
(Iteration 2001 / 4900) loss: 1.245403
(Iteration 2101 / 4900) loss: 1.338198
(Iteration 2201 / 4900) loss: 1.384518
(Iteration 2301 / 4900) loss: 1.202082
(Iteration 2401 / 4900) loss: 1.417495
(Epoch 5 / 10) train acc: 0.555000; val_acc: 0.504000
(Iteration 2501 / 4900) loss: 1.337666
(Iteration 2601 / 4900) loss: 1.258548
(Iteration 2701 / 4900) loss: 1.303501
(Iteration 2801 / 4900) loss: 1.159265
```

```
(Iteration 2901 / 4900) loss: 1.133871
(Epoch 6 / 10) train acc: 0.604000; val_acc: 0.499000
(Iteration 3001 / 4900) loss: 1.283895
(Iteration 3101 / 4900) loss: 1.305139
(Iteration 3201 / 4900) loss: 1.199517
(Iteration 3301 / 4900) loss: 1.045683
(Iteration 3401 / 4900) loss: 1.121629
(Epoch 7 / 10) train acc: 0.583000; val_acc: 0.516000
(Iteration 3501 / 4900) loss: 1.194922
(Iteration 3601 / 4900) loss: 1.516545
(Iteration 3701 / 4900) loss: 1.159288
(Iteration 3801 / 4900) loss: 1.228429
(Iteration 3901 / 4900) loss: 1.081823
(Epoch 8 / 10) train acc: 0.573000; val_acc: 0.519000
(Iteration 4001 / 4900) loss: 1.420857
(Iteration 4101 / 4900) loss: 1.347124
(Iteration 4201 / 4900) loss: 1.128454
(Iteration 4301 / 4900) loss: 1.172579
(Iteration 4401 / 4900) loss: 1.321357
(Epoch 9 / 10) train acc: 0.593000; val_acc: 0.526000
(Iteration 4501 / 4900) loss: 1.310129
(Iteration 4601 / 4900) loss: 1.212299
(Iteration 4701 / 4900) loss: 1.279539
(Iteration 4801 / 4900) loss: 1.039521
(Epoch 10 / 10) train acc: 0.585000; val_acc: 0.522000
(Iteration 1 / 4900) loss: 2.301995
(Epoch 0 / 10) train acc: 0.096000; val_acc: 0.113000
(Iteration 101 / 4900) loss: 2.298114
(Iteration 201 / 4900) loss: 2.296703
(Iteration 301 / 4900) loss: 2.292318
(Iteration 401 / 4900) loss: 2.285221
(Epoch 1 / 10) train acc: 0.244000; val_acc: 0.248000
(Iteration 501 / 4900) loss: 2.286731
(Iteration 601 / 4900) loss: 2.275560
(Iteration 701 / 4900) loss: 2.262082
(Iteration 801 / 4900) loss: 2.246649
(Iteration 901 / 4900) loss: 2.244737
(Epoch 2 / 10) train acc: 0.236000; val_acc: 0.247000
(Iteration 1001 / 4900) loss: 2.240676
(Iteration 1101 / 4900) loss: 2.223271
(Iteration 1201 / 4900) loss: 2.227983
(Iteration 1301 / 4900) loss: 2.211689
(Iteration 1401 / 4900) loss: 2.200222
(Epoch 3 / 10) train acc: 0.204000; val_acc: 0.242000
(Iteration 1501 / 4900) loss: 2.147731
(Iteration 1601 / 4900) loss: 2.194619
(Iteration 1701 / 4900) loss: 2.121347
(Iteration 1801 / 4900) loss: 2.155288
```

```
(Iteration 1901 / 4900) loss: 2.234986
(Epoch 4 / 10) train acc: 0.244000; val_acc: 0.254000
(Iteration 2001 / 4900) loss: 2.100540
(Iteration 2101 / 4900) loss: 2.110350
(Iteration 2201 / 4900) loss: 2.087345
(Iteration 2301 / 4900) loss: 2.112593
(Iteration 2401 / 4900) loss: 2.090787
(Epoch 5 / 10) train acc: 0.237000; val_acc: 0.264000
(Iteration 2501 / 4900) loss: 2.030008
(Iteration 2601 / 4900) loss: 2.074922
(Iteration 2701 / 4900) loss: 2.033879
(Iteration 2801 / 4900) loss: 2.004088
(Iteration 2901 / 4900) loss: 2.038889
(Epoch 6 / 10) train acc: 0.265000; val_acc: 0.288000
(Iteration 3001 / 4900) loss: 2.078373
(Iteration 3101 / 4900) loss: 2.006591
(Iteration 3201 / 4900) loss: 1.951942
(Iteration 3301 / 4900) loss: 1.997849
(Iteration 3401 / 4900) loss: 1.939200
(Epoch 7 / 10) train acc: 0.267000; val_acc: 0.300000
(Iteration 3501 / 4900) loss: 2.010414
(Iteration 3601 / 4900) loss: 1.991122
(Iteration 3701 / 4900) loss: 2.019886
(Iteration 3801 / 4900) loss: 1.989922
(Iteration 3901 / 4900) loss: 1.925155
(Epoch 8 / 10) train acc: 0.304000; val_acc: 0.307000
(Iteration 4001 / 4900) loss: 1.994990
(Iteration 4101 / 4900) loss: 2.011446
(Iteration 4201 / 4900) loss: 1.863763
(Iteration 4301 / 4900) loss: 1.898242
(Iteration 4401 / 4900) loss: 1.981950
(Epoch 9 / 10) train acc: 0.300000; val_acc: 0.319000
(Iteration 4501 / 4900) loss: 1.946242
(Iteration 4601 / 4900) loss: 1.936488
(Iteration 4701 / 4900) loss: 2.007436
(Iteration 4801 / 4900) loss: 1.949582
(Epoch 10 / 10) train acc: 0.315000; val_acc: 0.325000
(Iteration 1 / 4900) loss: 2.301144
(Epoch 0 / 10) train acc: 0.105000; val_acc: 0.106000
(Iteration 101 / 4900) loss: 2.299245
(Iteration 201 / 4900) loss: 2.294772
(Iteration 301 / 4900) loss: 2.292199
(Iteration 401 / 4900) loss: 2.289860
(Epoch 1 / 10) train acc: 0.214000; val_acc: 0.206000
(Iteration 501 / 4900) loss: 2.273046
(Iteration 601 / 4900) loss: 2.270958
(Iteration 701 / 4900) loss: 2.264966
(Iteration 801 / 4900) loss: 2.272846
```

```
(Iteration 901 / 4900) loss: 2.258183
(Epoch 2 / 10) train acc: 0.220000; val_acc: 0.243000
(Iteration 1001 / 4900) loss: 2.237153
(Iteration 1101 / 4900) loss: 2.233850
(Iteration 1201 / 4900) loss: 2.225548
(Iteration 1301 / 4900) loss: 2.240368
(Iteration 1401 / 4900) loss: 2.220029
(Epoch 3 / 10) train acc: 0.255000; val_acc: 0.260000
(Iteration 1501 / 4900) loss: 2.226993
(Iteration 1601 / 4900) loss: 2.207254
(Iteration 1701 / 4900) loss: 2.224330
(Iteration 1801 / 4900) loss: 2.207458
(Iteration 1901 / 4900) loss: 2.239402
(Epoch 4 / 10) train acc: 0.236000; val_acc: 0.258000
(Iteration 2001 / 4900) loss: 2.201499
(Iteration 2101 / 4900) loss: 2.212820
(Iteration 2201 / 4900) loss: 2.142725
(Iteration 2301 / 4900) loss: 2.152788
(Iteration 2401 / 4900) loss: 2.182176
(Epoch 5 / 10) train acc: 0.254000; val_acc: 0.264000
(Iteration 2501 / 4900) loss: 2.169817
(Iteration 2601 / 4900) loss: 2.145808
(Iteration 2701 / 4900) loss: 2.144646
(Iteration 2801 / 4900) loss: 2.190211
(Iteration 2901 / 4900) loss: 2.126366
(Epoch 6 / 10) train acc: 0.256000; val_acc: 0.264000
(Iteration 3001 / 4900) loss: 2.185806
(Iteration 3101 / 4900) loss: 2.114883
(Iteration 3201 / 4900) loss: 2.154857
(Iteration 3301 / 4900) loss: 2.170654
(Iteration 3401 / 4900) loss: 2.172942
(Epoch 7 / 10) train acc: 0.259000; val_acc: 0.261000
(Iteration 3501 / 4900) loss: 2.153809
(Iteration 3601 / 4900) loss: 2.149442
(Iteration 3701 / 4900) loss: 2.164910
(Iteration 3801 / 4900) loss: 2.131079
(Iteration 3901 / 4900) loss: 2.122168
(Epoch 8 / 10) train acc: 0.246000; val_acc: 0.262000
(Iteration 4001 / 4900) loss: 2.137850
(Iteration 4101 / 4900) loss: 2.142293
(Iteration 4201 / 4900) loss: 2.114758
(Iteration 4301 / 4900) loss: 2.141827
(Iteration 4401 / 4900) loss: 2.114880
(Epoch 9 / 10) train acc: 0.262000; val_acc: 0.263000
(Iteration 4501 / 4900) loss: 2.158960
(Iteration 4601 / 4900) loss: 2.142840
(Iteration 4701 / 4900) loss: 2.097364
(Iteration 4801 / 4900) loss: 2.167882
```

```
(Epoch 10 / 10) train acc: 0.257000; val_acc: 0.263000
(Iteration 1 / 4900) loss: 2.303296
(Epoch 0 / 10) train acc: 0.090000; val_acc: 0.077000
(Iteration 101 / 4900) loss: 2.238563
(Iteration 201 / 4900) loss: 2.092222
(Iteration 301 / 4900) loss: 2.054111
(Iteration 401 / 4900) loss: 1.974015
(Epoch 1 / 10) train acc: 0.318000; val_acc: 0.325000
(Iteration 501 / 4900) loss: 1.913507
(Iteration 601 / 4900) loss: 1.911852
(Iteration 701 / 4900) loss: 1.726525
(Iteration 801 / 4900) loss: 1.751784
(Iteration 901 / 4900) loss: 1.775569
(Epoch 2 / 10) train acc: 0.381000; val_acc: 0.385000
(Iteration 1001 / 4900) loss: 1.683653
(Iteration 1101 / 4900) loss: 1.762722
(Iteration 1201 / 4900) loss: 1.756658
(Iteration 1301 / 4900) loss: 1.792798
(Iteration 1401 / 4900) loss: 1.782841
(Epoch 3 / 10) train acc: 0.400000; val_acc: 0.389000
(Iteration 1501 / 4900) loss: 1.899038
(Iteration 1601 / 4900) loss: 1.534826
(Iteration 1701 / 4900) loss: 1.705666
(Iteration 1801 / 4900) loss: 1.803264
(Iteration 1901 / 4900) loss: 1.793140
(Epoch 4 / 10) train acc: 0.411000; val_acc: 0.433000
(Iteration 2001 / 4900) loss: 1.720339
(Iteration 2101 / 4900) loss: 1.507431
(Iteration 2201 / 4900) loss: 1.609828
(Iteration 2301 / 4900) loss: 1.568631
(Iteration 2401 / 4900) loss: 1.653453
(Epoch 5 / 10) train acc: 0.467000; val_acc: 0.444000
(Iteration 2501 / 4900) loss: 1.571005
(Iteration 2601 / 4900) loss: 1.755173
(Iteration 2701 / 4900) loss: 1.659103
(Iteration 2801 / 4900) loss: 1.522602
(Iteration 2901 / 4900) loss: 1.539019
(Epoch 6 / 10) train acc: 0.441000; val_acc: 0.461000
(Iteration 3001 / 4900) loss: 1.723666
(Iteration 3101 / 4900) loss: 1.597351
(Iteration 3201 / 4900) loss: 1.591864
(Iteration 3301 / 4900) loss: 1.555222
(Iteration 3401 / 4900) loss: 1.619050
(Epoch 7 / 10) train acc: 0.479000; val_acc: 0.460000
(Iteration 3501 / 4900) loss: 1.424437
(Iteration 3601 / 4900) loss: 1.438996
(Iteration 3701 / 4900) loss: 1.409415
(Iteration 3801 / 4900) loss: 1.592414
```

```
(Iteration 3901 / 4900) loss: 1.408979
(Epoch 8 / 10) train acc: 0.499000; val_acc: 0.471000
(Iteration 4001 / 4900) loss: 1.394553
(Iteration 4101 / 4900) loss: 1.523540
(Iteration 4201 / 4900) loss: 1.514619
(Iteration 4301 / 4900) loss: 1.469869
(Iteration 4401 / 4900) loss: 1.316281
(Epoch 9 / 10) train acc: 0.489000; val_acc: 0.489000
(Iteration 4501 / 4900) loss: 1.577714
(Iteration 4601 / 4900) loss: 1.285369
(Iteration 4701 / 4900) loss: 1.455296
(Iteration 4801 / 4900) loss: 1.573562
(Epoch 10 / 10) train acc: 0.491000; val_acc: 0.481000
(Iteration 1 / 4900) loss: 2.301583
(Epoch 0 / 10) train acc: 0.100000; val_acc: 0.111000
(Iteration 101 / 4900) loss: 2.218380
(Iteration 201 / 4900) loss: 2.160635
(Iteration 301 / 4900) loss: 2.020043
(Iteration 401 / 4900) loss: 1.913493
(Epoch 1 / 10) train acc: 0.313000; val_acc: 0.308000
(Iteration 501 / 4900) loss: 1.981236
(Iteration 601 / 4900) loss: 1.858443
(Iteration 701 / 4900) loss: 1.975741
(Iteration 801 / 4900) loss: 1.875973
(Iteration 901 / 4900) loss: 1.849644
(Epoch 2 / 10) train acc: 0.364000; val_acc: 0.360000
(Iteration 1001 / 4900) loss: 1.851444
(Iteration 1101 / 4900) loss: 1.859270
(Iteration 1201 / 4900) loss: 1.665850
(Iteration 1301 / 4900) loss: 1.747614
(Iteration 1401 / 4900) loss: 1.800085
(Epoch 3 / 10) train acc: 0.369000; val_acc: 0.387000
(Iteration 1501 / 4900) loss: 1.769129
(Iteration 1601 / 4900) loss: 1.828312
(Iteration 1701 / 4900) loss: 1.813540
(Iteration 1801 / 4900) loss: 1.788508
(Iteration 1901 / 4900) loss: 1.760392
(Epoch 4 / 10) train acc: 0.380000; val_acc: 0.407000
(Iteration 2001 / 4900) loss: 1.600285
(Iteration 2101 / 4900) loss: 1.676754
(Iteration 2201 / 4900) loss: 1.595967
(Iteration 2301 / 4900) loss: 1.758928
(Iteration 2401 / 4900) loss: 1.669996
(Epoch 5 / 10) train acc: 0.406000; val_acc: 0.407000
(Iteration 2501 / 4900) loss: 1.687110
(Iteration 2601 / 4900) loss: 1.670360
(Iteration 2701 / 4900) loss: 1.601208
(Iteration 2801 / 4900) loss: 1.699902
```

```
(Iteration 2901 / 4900) loss: 1.569555
(Epoch 6 / 10) train acc: 0.390000; val_acc: 0.416000
(Iteration 3001 / 4900) loss: 1.574481
(Iteration 3101 / 4900) loss: 1.794515
(Iteration 3201 / 4900) loss: 1.620405
(Iteration 3301 / 4900) loss: 1.551847
(Iteration 3401 / 4900) loss: 1.672831
(Epoch 7 / 10) train acc: 0.405000; val_acc: 0.421000
(Iteration 3501 / 4900) loss: 1.520961
(Iteration 3601 / 4900) loss: 1.551375
(Iteration 3701 / 4900) loss: 1.594594
(Iteration 3801 / 4900) loss: 1.764591
(Iteration 3901 / 4900) loss: 1.608222
(Epoch 8 / 10) train acc: 0.420000; val_acc: 0.426000
(Iteration 4001 / 4900) loss: 1.684038
(Iteration 4101 / 4900) loss: 1.620177
(Iteration 4201 / 4900) loss: 1.493778
(Iteration 4301 / 4900) loss: 1.928767
(Iteration 4401 / 4900) loss: 1.530115
(Epoch 9 / 10) train acc: 0.421000; val_acc: 0.424000
(Iteration 4501 / 4900) loss: 1.781832
(Iteration 4601 / 4900) loss: 1.749036
(Iteration 4701 / 4900) loss: 1.391129
(Iteration 4801 / 4900) loss: 1.778256
(Epoch 10 / 10) train acc: 0.419000; val_acc: 0.423000
(Iteration 1 / 4900) loss: 2.300894
(Epoch 0 / 10) train acc: 0.126000; val_acc: 0.143000
(Iteration 101 / 4900) loss: 1.809866
(Iteration 201 / 4900) loss: 1.733075
(Iteration 301 / 4900) loss: 1.600049
(Iteration 401 / 4900) loss: 1.641380
(Epoch 1 / 10) train acc: 0.463000; val_acc: 0.423000
(Iteration 501 / 4900) loss: 1.583686
(Iteration 601 / 4900) loss: 1.463395
(Iteration 701 / 4900) loss: 1.329931
(Iteration 801 / 4900) loss: 1.445243
(Iteration 901 / 4900) loss: 1.480026
(Epoch 2 / 10) train acc: 0.524000; val_acc: 0.470000
(Iteration 1001 / 4900) loss: 1.280733
(Iteration 1101 / 4900) loss: 1.342051
(Iteration 1201 / 4900) loss: 1.402517
(Iteration 1301 / 4900) loss: 1.450302
(Iteration 1401 / 4900) loss: 1.522650
(Epoch 3 / 10) train acc: 0.496000; val_acc: 0.496000
(Iteration 1501 / 4900) loss: 1.613534
(Iteration 1601 / 4900) loss: 1.346912
(Iteration 1701 / 4900) loss: 1.458194
(Iteration 1801 / 4900) loss: 1.331275
```

```
(Iteration 1901 / 4900) loss: 1.336134
(Epoch 4 / 10) train acc: 0.541000; val_acc: 0.478000
(Iteration 2001 / 4900) loss: 1.454977
(Iteration 2101 / 4900) loss: 1.382992
(Iteration 2201 / 4900) loss: 1.472722
(Iteration 2301 / 4900) loss: 1.236565
(Iteration 2401 / 4900) loss: 1.137472
(Epoch 5 / 10) train acc: 0.525000; val_acc: 0.484000
(Iteration 2501 / 4900) loss: 1.408234
(Iteration 2601 / 4900) loss: 1.375667
(Iteration 2701 / 4900) loss: 1.341348
(Iteration 2801 / 4900) loss: 1.371225
(Iteration 2901 / 4900) loss: 1.202994
(Epoch 6 / 10) train acc: 0.565000; val_acc: 0.489000
(Iteration 3001 / 4900) loss: 1.205291
(Iteration 3101 / 4900) loss: 1.169704
(Iteration 3201 / 4900) loss: 1.301144
(Iteration 3301 / 4900) loss: 1.249198
(Iteration 3401 / 4900) loss: 1.578334
(Epoch 7 / 10) train acc: 0.596000; val_acc: 0.510000
(Iteration 3501 / 4900) loss: 1.203951
(Iteration 3601 / 4900) loss: 1.450209
(Iteration 3701 / 4900) loss: 1.072341
(Iteration 3801 / 4900) loss: 1.325696
(Iteration 3901 / 4900) loss: 1.250716
(Epoch 8 / 10) train acc: 0.580000; val_acc: 0.502000
(Iteration 4001 / 4900) loss: 1.291551
(Iteration 4101 / 4900) loss: 1.178122
(Iteration 4201 / 4900) loss: 1.141354
(Iteration 4301 / 4900) loss: 1.061436
(Iteration 4401 / 4900) loss: 1.360277
(Epoch 9 / 10) train acc: 0.537000; val_acc: 0.481000
(Iteration 4501 / 4900) loss: 1.528832
(Iteration 4601 / 4900) loss: 1.034450
(Iteration 4701 / 4900) loss: 1.338930
(Iteration 4801 / 4900) loss: 1.174668
(Epoch 10 / 10) train acc: 0.583000; val_acc: 0.505000
(Iteration 1 / 4900) loss: 2.299680
(Epoch 0 / 10) train acc: 0.142000; val_acc: 0.154000
(Iteration 101 / 4900) loss: 1.873483
(Iteration 201 / 4900) loss: 1.751010
(Iteration 301 / 4900) loss: 1.598762
(Iteration 401 / 4900) loss: 1.674976
(Epoch 1 / 10) train acc: 0.454000; val_acc: 0.457000
(Iteration 501 / 4900) loss: 1.509590
(Iteration 601 / 4900) loss: 1.717890
(Iteration 701 / 4900) loss: 1.648706
(Iteration 801 / 4900) loss: 1.535046
```

```
(Iteration 901 / 4900) loss: 1.311613
(Epoch 2 / 10) train acc: 0.486000; val_acc: 0.426000
(Iteration 1001 / 4900) loss: 1.228332
(Iteration 1101 / 4900) loss: 1.257930
(Iteration 1201 / 4900) loss: 1.269632
(Iteration 1301 / 4900) loss: 1.589049
(Iteration 1401 / 4900) loss: 1.393484
(Epoch 3 / 10) train acc: 0.533000; val_acc: 0.479000
(Iteration 1501 / 4900) loss: 1.288857
(Iteration 1601 / 4900) loss: 1.303034
(Iteration 1701 / 4900) loss: 1.391828
(Iteration 1801 / 4900) loss: 1.430958
(Iteration 1901 / 4900) loss: 1.331314
(Epoch 4 / 10) train acc: 0.520000; val_acc: 0.507000
(Iteration 2001 / 4900) loss: 1.304616
(Iteration 2101 / 4900) loss: 1.349300
(Iteration 2201 / 4900) loss: 1.225396
(Iteration 2301 / 4900) loss: 1.258429
(Iteration 2401 / 4900) loss: 1.060144
(Epoch 5 / 10) train acc: 0.568000; val_acc: 0.515000
(Iteration 2501 / 4900) loss: 1.281323
(Iteration 2601 / 4900) loss: 1.342208
(Iteration 2701 / 4900) loss: 1.049700
(Iteration 2801 / 4900) loss: 1.317336
(Iteration 2901 / 4900) loss: 1.202307
(Epoch 6 / 10) train acc: 0.588000; val_acc: 0.536000
(Iteration 3001 / 4900) loss: 1.352771
(Iteration 3101 / 4900) loss: 1.303381
(Iteration 3201 / 4900) loss: 1.322095
(Iteration 3301 / 4900) loss: 1.261666
(Iteration 3401 / 4900) loss: 1.230053
(Epoch 7 / 10) train acc: 0.597000; val_acc: 0.519000
(Iteration 3501 / 4900) loss: 1.036061
(Iteration 3601 / 4900) loss: 1.048811
(Iteration 3701 / 4900) loss: 1.425013
(Iteration 3801 / 4900) loss: 1.239116
(Iteration 3901 / 4900) loss: 1.228352
(Epoch 8 / 10) train acc: 0.592000; val_acc: 0.538000
(Iteration 4001 / 4900) loss: 1.086622
(Iteration 4101 / 4900) loss: 1.226474
(Iteration 4201 / 4900) loss: 1.124573
(Iteration 4301 / 4900) loss: 1.198775
(Iteration 4401 / 4900) loss: 1.126640
(Epoch 9 / 10) train acc: 0.585000; val_acc: 0.526000
(Iteration 4501 / 4900) loss: 1.097248
(Iteration 4601 / 4900) loss: 1.322846
(Iteration 4701 / 4900) loss: 1.192638
(Iteration 4801 / 4900) loss: 1.040969
```

```
(Epoch 10 / 10) train acc: 0.603000; val_acc: 0.542000
```

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.

[15]:
```
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:  0.542
```

[16]:
```
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:  0.537
```

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 :(1), 3

YourExplanation : 1. I would say the first option is not works in all situation. When your training set is too small than train on a larger dataset will help, however, when the dataset is large enough, then it will not affect much. We can see this behavior from the learning curve. 3.The reason of the accuracy difference is mostly happen when its fits too perfect. The only way to prevent it fit too well is add regularization strength.

[16]:
```
```

features

October 8, 2022

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

# Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment1/'
FOLDERNAME = 'enpm809K/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 drive/My\ Drive/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/enpm809K/assignments/assignment1/cs231n/datasets
/content/drive/My Drive/enpm809K/assignments/assignment1
```

1 Image features exercise

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

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

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

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


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

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

1.1 Load data

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

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

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

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

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

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

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

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

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

1.2 Extract Features

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

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

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

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

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

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

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

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

```
X_val_feats = np.hstack([X_val_feats, np.ones((X_val_feats.shape[0], 1))])
X_test_feats = np.hstack([X_test_feats, np.ones((X_test_feats.shape[0], 1))])
```

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

```
Done extracting features for 45000 / 49000 images
Done extracting features for 46000 / 49000 images
Done extracting features for 47000 / 49000 images
Done extracting features for 48000 / 49000 images
Done extracting features for 49000 / 49000 images
```

1.3 Train SVM on features

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

[5]:
```
# Use the validation set to tune the learning rate and regularization strength

from cs231n.classifiers.linear_classifier import LinearSVM

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

results = {}
best_val = -1
best_svm = None

################################################################################
# TODO:                                                                        #
# Use the validation set to set the learning rate and regularization strength. #
# This should be identical to the validation that you did for the SVM; save    #
# the best trained classifer in best_svm. You might also want to play          #
# with different numbers of bins in the color histogram. If you are careful    #
# you should be able to get accuracy of near 0.44 on the validation set.       #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
# define svm
svm = LinearSVM()
# define the parameters that I think it will have good results
my_learning_rates = [1e-8, 2e-8, 3e-8, 4e-8, 5e-8, 6e-8, 7e-8, 8e-8, 9e-8]
my_regularization_strengths = [1.5e4, 2.5e4, 3.5e4, 4.5e4, 5.5e4, 6.5e4, 7.5e4,␣
 ↪8.5e4, 9.5e4]

# go through every pairs of parameters to find out which are best in the␣
 ↪validation accuracy
for learning_rate in my_learning_rates:
  for regularization_strength in my_regularization_strengths:

    #tic = time.time()
    # svm train
```

```
    loss_hist = svm.train(X_train_feats, y_train, learning_rate=learning_rate,␣
↪reg=regularization_strength,
                          num_iters=1500, verbose=True)
    # get training predict class
    y_train_pred = svm.predict(X_train_feats)
    # print('training accuracy: %f' % (np.mean(y_train == y_train_pred), ))

    # calculate training accuracy
    training_accuracy = np.mean(y_train == y_train_pred)

    # get validation predict class
    y_val_pred = svm.predict(X_val_feats)
    # print('validation accuracy: %f' % (np.mean(y_val == y_val_pred), ))

    # calculate training accuracy
    validation_accuracy = np.mean(y_val == y_val_pred)
    # save result to dictionary
    results[(learning_rate, regularization_strength)] = [training_accuracy,␣
↪validation_accuracy]

    # compare to the best model to find out which one is better
    if(validation_accuracy > best_val):
      best_val = validation_accuracy
      best_svm = svm

# pass

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

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

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

```
iteration 0 / 1500: loss 33.152014
iteration 100 / 1500: loss 32.401833
iteration 200 / 1500: loss 31.743884
iteration 300 / 1500: loss 31.061923
iteration 400 / 1500: loss 30.407670
iteration 500 / 1500: loss 29.789022
iteration 600 / 1500: loss 29.162192
iteration 700 / 1500: loss 28.577381
iteration 800 / 1500: loss 27.986720
iteration 900 / 1500: loss 27.430052
```

```
iteration 1000 / 1500: loss 26.877339
iteration 1100 / 1500: loss 26.356202
iteration 1200 / 1500: loss 25.836139
iteration 1300 / 1500: loss 25.353689
iteration 1400 / 1500: loss 24.853475
iteration 0 / 1500: loss 34.661806
iteration 100 / 1500: loss 33.415497
iteration 200 / 1500: loss 32.219905
iteration 300 / 1500: loss 31.095702
iteration 400 / 1500: loss 29.998681
iteration 500 / 1500: loss 28.992983
iteration 600 / 1500: loss 28.000579
iteration 700 / 1500: loss 27.083310
iteration 800 / 1500: loss 26.213502
iteration 900 / 1500: loss 25.367715
iteration 1000 / 1500: loss 24.556422
iteration 1100 / 1500: loss 23.793955
iteration 1200 / 1500: loss 23.082884
iteration 1300 / 1500: loss 22.391908
iteration 1400 / 1500: loss 21.740001
iteration 0 / 1500: loss 25.971636
iteration 100 / 1500: loss 24.827291
iteration 200 / 1500: loss 23.759531
iteration 300 / 1500: loss 22.754350
iteration 400 / 1500: loss 21.828422
iteration 500 / 1500: loss 20.957659
iteration 600 / 1500: loss 20.152127
iteration 700 / 1500: loss 19.400122
iteration 800 / 1500: loss 18.693492
iteration 900 / 1500: loss 18.038782
iteration 1000 / 1500: loss 17.425847
iteration 1100 / 1500: loss 16.859430
iteration 1200 / 1500: loss 16.331263
iteration 1300 / 1500: loss 15.830168
iteration 1400 / 1500: loss 15.375331
iteration 0 / 1500: loss 16.640209
iteration 100 / 1500: loss 15.985353
iteration 200 / 1500: loss 15.382204
iteration 300 / 1500: loss 14.828194
iteration 400 / 1500: loss 14.330080
iteration 500 / 1500: loss 13.871255
iteration 600 / 1500: loss 13.451724
iteration 700 / 1500: loss 13.068335
iteration 800 / 1500: loss 12.716482
iteration 900 / 1500: loss 12.399947
iteration 1000 / 1500: loss 12.105470
iteration 1100 / 1500: loss 11.839425
iteration 1200 / 1500: loss 11.593955
```

```
iteration 1300 / 1500: loss 11.372392
iteration 1400 / 1500: loss 11.168443
iteration 0 / 1500: loss 11.422453
iteration 100 / 1500: loss 11.171942
iteration 200 / 1500: loss 10.941364
iteration 300 / 1500: loss 10.742027
iteration 400 / 1500: loss 10.560960
iteration 500 / 1500: loss 10.397236
iteration 600 / 1500: loss 10.250990
iteration 700 / 1500: loss 10.119910
iteration 800 / 1500: loss 10.004295
iteration 900 / 1500: loss 9.900139
iteration 1000 / 1500: loss 9.805966
iteration 1100 / 1500: loss 9.722235
iteration 1200 / 1500: loss 9.648524
iteration 1300 / 1500: loss 9.579669
iteration 1400 / 1500: loss 9.519384
iteration 0 / 1500: loss 9.550880
iteration 100 / 1500: loss 9.482581
iteration 200 / 1500: loss 9.424789
iteration 300 / 1500: loss 9.372669
iteration 400 / 1500: loss 9.327605
iteration 500 / 1500: loss 9.286720
iteration 600 / 1500: loss 9.252144
iteration 700 / 1500: loss 9.221426
iteration 800 / 1500: loss 9.194766
iteration 900 / 1500: loss 9.170118
iteration 1000 / 1500: loss 9.151032
iteration 1100 / 1500: loss 9.132095
iteration 1200 / 1500: loss 9.115741
iteration 1300 / 1500: loss 9.101516
iteration 1400 / 1500: loss 9.089601
iteration 0 / 1500: loss 9.091187
iteration 100 / 1500: loss 9.078453
iteration 200 / 1500: loss 9.067560
iteration 300 / 1500: loss 9.057593
iteration 400 / 1500: loss 9.049844
iteration 500 / 1500: loss 9.043339
iteration 600 / 1500: loss 9.036809
iteration 700 / 1500: loss 9.032057
iteration 800 / 1500: loss 9.027494
iteration 900 / 1500: loss 9.023890
iteration 1000 / 1500: loss 9.020334
iteration 1100 / 1500: loss 9.017677
iteration 1200 / 1500: loss 9.015105
iteration 1300 / 1500: loss 9.012952
iteration 1400 / 1500: loss 9.011209
iteration 0 / 1500: loss 9.011096
```

```
iteration 100 / 1500: loss 9.009342
iteration 200 / 1500: loss 9.008005
iteration 300 / 1500: loss 9.006750
iteration 400 / 1500: loss 9.005651
iteration 500 / 1500: loss 9.004700
iteration 600 / 1500: loss 9.004058
iteration 700 / 1500: loss 9.003494
iteration 800 / 1500: loss 9.002995
iteration 900 / 1500: loss 9.002417
iteration 1000 / 1500: loss 9.002080
iteration 1100 / 1500: loss 9.001889
iteration 1200 / 1500: loss 9.001592
iteration 1300 / 1500: loss 9.001331
iteration 1400 / 1500: loss 9.001178
iteration 0 / 1500: loss 9.001142
iteration 100 / 1500: loss 9.000959
iteration 200 / 1500: loss 9.000932
iteration 300 / 1500: loss 9.000637
iteration 400 / 1500: loss 9.000546
iteration 500 / 1500: loss 9.000415
iteration 600 / 1500: loss 9.000359
iteration 700 / 1500: loss 9.000444
iteration 800 / 1500: loss 9.000200
iteration 900 / 1500: loss 9.000282
iteration 1000 / 1500: loss 9.000228
iteration 1100 / 1500: loss 9.000183
iteration 1200 / 1500: loss 9.000078
iteration 1300 / 1500: loss 9.000039
iteration 1400 / 1500: loss 8.999961
iteration 0 / 1500: loss 8.999343
iteration 100 / 1500: loss 8.999354
iteration 200 / 1500: loss 8.999138
iteration 300 / 1500: loss 8.999094
iteration 400 / 1500: loss 8.999026
iteration 500 / 1500: loss 8.998973
iteration 600 / 1500: loss 8.998983
iteration 700 / 1500: loss 8.999064
iteration 800 / 1500: loss 8.999039
iteration 900 / 1500: loss 8.998921
iteration 1000 / 1500: loss 8.999170
iteration 1100 / 1500: loss 8.998829
iteration 1200 / 1500: loss 8.998825
iteration 1300 / 1500: loss 8.999092
iteration 1400 / 1500: loss 8.998627
iteration 0 / 1500: loss 8.999526
iteration 100 / 1500: loss 8.999636
iteration 200 / 1500: loss 8.999531
iteration 300 / 1500: loss 8.999658
```

```
iteration 400 / 1500: loss 8.999580
iteration 500 / 1500: loss 8.999393
iteration 600 / 1500: loss 8.999372
iteration 700 / 1500: loss 8.999982
iteration 800 / 1500: loss 8.999681
iteration 900 / 1500: loss 8.999222
iteration 1000 / 1500: loss 8.999533
iteration 1100 / 1500: loss 8.999520
iteration 1200 / 1500: loss 8.999960
iteration 1300 / 1500: loss 8.999654
iteration 1400 / 1500: loss 8.999547
iteration 0 / 1500: loss 9.000790
iteration 100 / 1500: loss 9.000589
iteration 200 / 1500: loss 9.000703
iteration 300 / 1500: loss 9.000486
iteration 400 / 1500: loss 9.000625
iteration 500 / 1500: loss 8.999981
iteration 600 / 1500: loss 9.000191
iteration 700 / 1500: loss 9.000284
iteration 800 / 1500: loss 8.999986
iteration 900 / 1500: loss 9.000333
iteration 1000 / 1500: loss 9.000270
iteration 1100 / 1500: loss 9.000247
iteration 1200 / 1500: loss 9.000386
iteration 1300 / 1500: loss 9.000150
iteration 1400 / 1500: loss 9.000052
iteration 0 / 1500: loss 9.000917
iteration 100 / 1500: loss 9.000858
iteration 200 / 1500: loss 9.000745
iteration 300 / 1500: loss 9.000641
iteration 400 / 1500: loss 9.000448
iteration 500 / 1500: loss 9.000203
iteration 600 / 1500: loss 9.000354
iteration 700 / 1500: loss 9.000539
iteration 800 / 1500: loss 9.000248
iteration 900 / 1500: loss 9.000263
iteration 1000 / 1500: loss 9.000306
iteration 1100 / 1500: loss 9.000239
iteration 1200 / 1500: loss 9.000353
iteration 1300 / 1500: loss 9.000042
iteration 1400 / 1500: loss 9.000297
iteration 0 / 1500: loss 9.000859
iteration 100 / 1500: loss 9.000527
iteration 200 / 1500: loss 9.000700
iteration 300 / 1500: loss 9.000433
iteration 400 / 1500: loss 9.000288
iteration 500 / 1500: loss 9.000230
iteration 600 / 1500: loss 9.000156
```

```
iteration 700 / 1500: loss 9.000147
iteration 800 / 1500: loss 9.000188
iteration 900 / 1500: loss 9.000137
iteration 1000 / 1500: loss 9.000320
iteration 1100 / 1500: loss 9.000269
iteration 1200 / 1500: loss 9.000032
iteration 1300 / 1500: loss 9.000161
iteration 1400 / 1500: loss 8.999955
iteration 0 / 1500: loss 9.000313
iteration 100 / 1500: loss 9.000371
iteration 200 / 1500: loss 9.000166
iteration 300 / 1500: loss 9.000254
iteration 400 / 1500: loss 9.000223
iteration 500 / 1500: loss 9.000204
iteration 600 / 1500: loss 9.000296
iteration 700 / 1500: loss 9.000160
iteration 800 / 1500: loss 8.999894
iteration 900 / 1500: loss 9.000110
iteration 1000 / 1500: loss 9.000116
iteration 1100 / 1500: loss 9.000104
iteration 1200 / 1500: loss 8.999985
iteration 1300 / 1500: loss 9.000101
iteration 1400 / 1500: loss 9.000034
iteration 0 / 1500: loss 9.000188
iteration 100 / 1500: loss 9.000057
iteration 200 / 1500: loss 9.000219
iteration 300 / 1500: loss 9.000219
iteration 400 / 1500: loss 9.000153
iteration 500 / 1500: loss 9.000017
iteration 600 / 1500: loss 9.000075
iteration 700 / 1500: loss 9.000120
iteration 800 / 1500: loss 9.000109
iteration 900 / 1500: loss 8.999861
iteration 1000 / 1500: loss 9.000104
iteration 1100 / 1500: loss 9.000099
iteration 1200 / 1500: loss 9.000045
iteration 1300 / 1500: loss 8.999992
iteration 1400 / 1500: loss 9.000071
iteration 0 / 1500: loss 9.000218
iteration 100 / 1500: loss 9.000039
iteration 200 / 1500: loss 9.000077
iteration 300 / 1500: loss 9.000148
iteration 400 / 1500: loss 9.000065
iteration 500 / 1500: loss 9.000156
iteration 600 / 1500: loss 9.000041
iteration 700 / 1500: loss 9.000030
iteration 800 / 1500: loss 9.000117
iteration 900 / 1500: loss 9.000017
```

```
iteration 1000 / 1500: loss 9.000043
iteration 1100 / 1500: loss 8.999960
iteration 1200 / 1500: loss 9.000112
iteration 1300 / 1500: loss 8.999991
iteration 1400 / 1500: loss 9.000092
iteration 0 / 1500: loss 9.000078
iteration 100 / 1500: loss 9.000126
iteration 200 / 1500: loss 9.000031
iteration 300 / 1500: loss 9.000064
iteration 400 / 1500: loss 9.000001
iteration 500 / 1500: loss 8.999946
iteration 600 / 1500: loss 9.000042
iteration 700 / 1500: loss 9.000040
iteration 800 / 1500: loss 9.000048
iteration 900 / 1500: loss 8.999937
iteration 1000 / 1500: loss 8.999988
iteration 1100 / 1500: loss 8.999976
iteration 1200 / 1500: loss 9.000055
iteration 1300 / 1500: loss 9.000056
iteration 1400 / 1500: loss 8.999955
iteration 0 / 1500: loss 8.999463
iteration 100 / 1500: loss 8.999232
iteration 200 / 1500: loss 8.999136
iteration 300 / 1500: loss 8.999078
iteration 400 / 1500: loss 8.998968
iteration 500 / 1500: loss 8.999056
iteration 600 / 1500: loss 8.999062
iteration 700 / 1500: loss 8.999066
iteration 800 / 1500: loss 8.998921
iteration 900 / 1500: loss 8.998815
iteration 1000 / 1500: loss 8.998889
iteration 1100 / 1500: loss 8.998942
iteration 1200 / 1500: loss 8.998778
iteration 1300 / 1500: loss 8.998518
iteration 1400 / 1500: loss 8.998780
iteration 0 / 1500: loss 9.000180
iteration 100 / 1500: loss 8.999955
iteration 200 / 1500: loss 8.999698
iteration 300 / 1500: loss 9.000005
iteration 400 / 1500: loss 8.999746
iteration 500 / 1500: loss 8.999616
iteration 600 / 1500: loss 8.999597
iteration 700 / 1500: loss 8.999949
iteration 800 / 1500: loss 8.999907
iteration 900 / 1500: loss 8.999821
iteration 1000 / 1500: loss 8.999916
iteration 1100 / 1500: loss 8.999802
iteration 1200 / 1500: loss 8.999778
```

```
iteration 1300 / 1500: loss 8.999985
iteration 1400 / 1500: loss 8.999797
iteration 0 / 1500: loss 9.001071
iteration 100 / 1500: loss 9.000906
iteration 200 / 1500: loss 9.000905
iteration 300 / 1500: loss 9.000549
iteration 400 / 1500: loss 9.000291
iteration 500 / 1500: loss 9.000606
iteration 600 / 1500: loss 9.000487
iteration 700 / 1500: loss 9.000345
iteration 800 / 1500: loss 9.000115
iteration 900 / 1500: loss 9.000480
iteration 1000 / 1500: loss 9.000174
iteration 1100 / 1500: loss 9.000264
iteration 1200 / 1500: loss 9.000457
iteration 1300 / 1500: loss 9.000229
iteration 1400 / 1500: loss 8.999979
iteration 0 / 1500: loss 9.000592
iteration 100 / 1500: loss 9.000496
iteration 200 / 1500: loss 9.000717
iteration 300 / 1500: loss 9.000714
iteration 400 / 1500: loss 9.000523
iteration 500 / 1500: loss 9.000411
iteration 600 / 1500: loss 9.000393
iteration 700 / 1500: loss 9.000178
iteration 800 / 1500: loss 9.000336
iteration 900 / 1500: loss 9.000048
iteration 1000 / 1500: loss 9.000130
iteration 1100 / 1500: loss 9.000242
iteration 1200 / 1500: loss 9.000122
iteration 1300 / 1500: loss 9.000147
iteration 1400 / 1500: loss 8.999850
iteration 0 / 1500: loss 9.000442
iteration 100 / 1500: loss 9.000148
iteration 200 / 1500: loss 9.000473
iteration 300 / 1500: loss 9.000186
iteration 400 / 1500: loss 9.000077
iteration 500 / 1500: loss 9.000166
iteration 600 / 1500: loss 9.000031
iteration 700 / 1500: loss 9.000208
iteration 800 / 1500: loss 9.000146
iteration 900 / 1500: loss 9.000143
iteration 1000 / 1500: loss 9.000073
iteration 1100 / 1500: loss 9.000100
iteration 1200 / 1500: loss 8.999985
iteration 1300 / 1500: loss 9.000092
iteration 1400 / 1500: loss 9.000018
iteration 0 / 1500: loss 9.000226
```

```
iteration 100 / 1500: loss 9.000244
iteration 200 / 1500: loss 9.000126
iteration 300 / 1500: loss 9.000201
iteration 400 / 1500: loss 8.999969
iteration 500 / 1500: loss 9.000118
iteration 600 / 1500: loss 9.000119
iteration 700 / 1500: loss 9.000084
iteration 800 / 1500: loss 9.000074
iteration 900 / 1500: loss 9.000052
iteration 1000 / 1500: loss 9.000041
iteration 1100 / 1500: loss 9.000165
iteration 1200 / 1500: loss 9.000175
iteration 1300 / 1500: loss 8.999975
iteration 1400 / 1500: loss 8.999856
iteration 0 / 1500: loss 9.000083
iteration 100 / 1500: loss 9.000148
iteration 200 / 1500: loss 8.999922
iteration 300 / 1500: loss 9.000063
iteration 400 / 1500: loss 9.000048
iteration 500 / 1500: loss 9.000021
iteration 600 / 1500: loss 8.999985
iteration 700 / 1500: loss 9.000032
iteration 800 / 1500: loss 8.999965
iteration 900 / 1500: loss 8.999977
iteration 1000 / 1500: loss 8.999942
iteration 1100 / 1500: loss 9.000105
iteration 1200 / 1500: loss 9.000041
iteration 1300 / 1500: loss 9.000180
iteration 1400 / 1500: loss 9.000048
iteration 0 / 1500: loss 9.000162
iteration 100 / 1500: loss 9.000065
iteration 200 / 1500: loss 9.000055
iteration 300 / 1500: loss 9.000103
iteration 400 / 1500: loss 9.000118
iteration 500 / 1500: loss 9.000131
iteration 600 / 1500: loss 9.000068
iteration 700 / 1500: loss 9.000026
iteration 800 / 1500: loss 9.000078
iteration 900 / 1500: loss 9.000015
iteration 1000 / 1500: loss 9.000117
iteration 1100 / 1500: loss 8.999946
iteration 1200 / 1500: loss 9.000076
iteration 1300 / 1500: loss 8.999968
iteration 1400 / 1500: loss 9.000036
iteration 0 / 1500: loss 9.000180
iteration 100 / 1500: loss 9.000023
iteration 200 / 1500: loss 9.000078
iteration 300 / 1500: loss 8.999987
```

```
iteration 400 / 1500: loss 9.000137
iteration 500 / 1500: loss 8.999970
iteration 600 / 1500: loss 9.000073
iteration 700 / 1500: loss 9.000065
iteration 800 / 1500: loss 9.000062
iteration 900 / 1500: loss 8.999985
iteration 1000 / 1500: loss 9.000028
iteration 1100 / 1500: loss 9.000062
iteration 1200 / 1500: loss 8.999983
iteration 1300 / 1500: loss 9.000039
iteration 1400 / 1500: loss 9.000029
iteration 0 / 1500: loss 8.999435
iteration 100 / 1500: loss 8.999278
iteration 200 / 1500: loss 8.999123
iteration 300 / 1500: loss 8.999167
iteration 400 / 1500: loss 8.998991
iteration 500 / 1500: loss 8.999111
iteration 600 / 1500: loss 8.998825
iteration 700 / 1500: loss 8.998949
iteration 800 / 1500: loss 8.998990
iteration 900 / 1500: loss 8.998956
iteration 1000 / 1500: loss 8.998969
iteration 1100 / 1500: loss 8.998937
iteration 1200 / 1500: loss 8.998881
iteration 1300 / 1500: loss 8.998746
iteration 1400 / 1500: loss 8.999045
iteration 0 / 1500: loss 9.000016
iteration 100 / 1500: loss 9.000582
iteration 200 / 1500: loss 9.000595
iteration 300 / 1500: loss 9.000166
iteration 400 / 1500: loss 9.000232
iteration 500 / 1500: loss 9.000315
iteration 600 / 1500: loss 9.000352
iteration 700 / 1500: loss 8.999968
iteration 800 / 1500: loss 8.999993
iteration 900 / 1500: loss 8.999935
iteration 1000 / 1500: loss 9.000269
iteration 1100 / 1500: loss 8.999955
iteration 1200 / 1500: loss 8.999699
iteration 1300 / 1500: loss 8.999889
iteration 1400 / 1500: loss 8.999833
iteration 0 / 1500: loss 9.001126
iteration 100 / 1500: loss 9.000904
iteration 200 / 1500: loss 9.001006
iteration 300 / 1500: loss 9.000827
iteration 400 / 1500: loss 9.000411
iteration 500 / 1500: loss 9.000594
iteration 600 / 1500: loss 9.000450
```

```
iteration 700 / 1500: loss 9.000485
iteration 800 / 1500: loss 9.000265
iteration 900 / 1500: loss 9.000070
iteration 1000 / 1500: loss 9.000169
iteration 1100 / 1500: loss 9.000338
iteration 1200 / 1500: loss 9.000235
iteration 1300 / 1500: loss 9.000398
iteration 1400 / 1500: loss 9.000237
iteration 0 / 1500: loss 9.000879
iteration 100 / 1500: loss 9.000814
iteration 200 / 1500: loss 9.000547
iteration 300 / 1500: loss 9.000577
iteration 400 / 1500: loss 9.000215
iteration 500 / 1500: loss 9.000366
iteration 600 / 1500: loss 9.000401
iteration 700 / 1500: loss 9.000416
iteration 800 / 1500: loss 9.000073
iteration 900 / 1500: loss 9.000158
iteration 1000 / 1500: loss 9.000127
iteration 1100 / 1500: loss 9.000075
iteration 1200 / 1500: loss 9.000002
iteration 1300 / 1500: loss 9.000124
iteration 1400 / 1500: loss 8.999925
iteration 0 / 1500: loss 9.000544
iteration 100 / 1500: loss 9.000317
iteration 200 / 1500: loss 9.000186
iteration 300 / 1500: loss 9.000264
iteration 400 / 1500: loss 9.000275
iteration 500 / 1500: loss 9.000252
iteration 600 / 1500: loss 9.000086
iteration 700 / 1500: loss 9.000166
iteration 800 / 1500: loss 8.999943
iteration 900 / 1500: loss 9.000035
iteration 1000 / 1500: loss 9.000023
iteration 1100 / 1500: loss 9.000004
iteration 1200 / 1500: loss 8.999969
iteration 1300 / 1500: loss 8.999938
iteration 1400 / 1500: loss 9.000120
iteration 0 / 1500: loss 9.000204
iteration 100 / 1500: loss 9.000097
iteration 200 / 1500: loss 9.000102
iteration 300 / 1500: loss 9.000051
iteration 400 / 1500: loss 9.000082
iteration 500 / 1500: loss 8.999993
iteration 600 / 1500: loss 9.000093
iteration 700 / 1500: loss 9.000097
iteration 800 / 1500: loss 8.999968
iteration 900 / 1500: loss 9.000014
```

```
iteration 1000 / 1500: loss 9.000083
iteration 1100 / 1500: loss 9.000021
iteration 1200 / 1500: loss 8.999991
iteration 1300 / 1500: loss 9.000012
iteration 1400 / 1500: loss 9.000034
iteration 0 / 1500: loss 9.000096
iteration 100 / 1500: loss 9.000137
iteration 200 / 1500: loss 9.000122
iteration 300 / 1500: loss 8.999975
iteration 400 / 1500: loss 8.999983
iteration 500 / 1500: loss 8.999851
iteration 600 / 1500: loss 8.999952
iteration 700 / 1500: loss 9.000121
iteration 800 / 1500: loss 8.999930
iteration 900 / 1500: loss 8.999994
iteration 1000 / 1500: loss 9.000156
iteration 1100 / 1500: loss 9.000088
iteration 1200 / 1500: loss 9.000005
iteration 1300 / 1500: loss 8.999991
iteration 1400 / 1500: loss 8.999951
iteration 0 / 1500: loss 9.000137
iteration 100 / 1500: loss 8.999994
iteration 200 / 1500: loss 9.000079
iteration 300 / 1500: loss 8.999930
iteration 400 / 1500: loss 9.000030
iteration 500 / 1500: loss 8.999943
iteration 600 / 1500: loss 8.999915
iteration 700 / 1500: loss 9.000028
iteration 800 / 1500: loss 9.000077
iteration 900 / 1500: loss 8.999991
iteration 1000 / 1500: loss 8.999992
iteration 1100 / 1500: loss 9.000014
iteration 1200 / 1500: loss 9.000038
iteration 1300 / 1500: loss 9.000014
iteration 1400 / 1500: loss 8.999955
iteration 0 / 1500: loss 9.000126
iteration 100 / 1500: loss 9.000097
iteration 200 / 1500: loss 9.000054
iteration 300 / 1500: loss 9.000006
iteration 400 / 1500: loss 9.000105
iteration 500 / 1500: loss 9.000033
iteration 600 / 1500: loss 9.000008
iteration 700 / 1500: loss 9.000074
iteration 800 / 1500: loss 8.999963
iteration 900 / 1500: loss 8.999906
iteration 1000 / 1500: loss 8.999987
iteration 1100 / 1500: loss 8.999910
iteration 1200 / 1500: loss 9.000026
```

```
iteration 1300 / 1500: loss 9.000101
iteration 1400 / 1500: loss 9.000008
iteration 0 / 1500: loss 8.999328
iteration 100 / 1500: loss 8.999239
iteration 200 / 1500: loss 8.999081
iteration 300 / 1500: loss 8.998883
iteration 400 / 1500: loss 8.998921
iteration 500 / 1500: loss 8.998841
iteration 600 / 1500: loss 8.999103
iteration 700 / 1500: loss 8.998776
iteration 800 / 1500: loss 8.998964
iteration 900 / 1500: loss 8.998720
iteration 1000 / 1500: loss 8.998892
iteration 1100 / 1500: loss 8.998877
iteration 1200 / 1500: loss 8.998985
iteration 1300 / 1500: loss 8.999338
iteration 1400 / 1500: loss 8.998890
iteration 0 / 1500: loss 9.000585
iteration 100 / 1500: loss 9.000469
iteration 200 / 1500: loss 9.000819
iteration 300 / 1500: loss 9.000113
iteration 400 / 1500: loss 9.000383
iteration 500 / 1500: loss 9.000573
iteration 600 / 1500: loss 9.000285
iteration 700 / 1500: loss 9.000412
iteration 800 / 1500: loss 9.000233
iteration 900 / 1500: loss 9.000335
iteration 1000 / 1500: loss 8.999979
iteration 1100 / 1500: loss 9.000205
iteration 1200 / 1500: loss 9.000472
iteration 1300 / 1500: loss 9.000113
iteration 1400 / 1500: loss 9.000238
iteration 0 / 1500: loss 9.001058
iteration 100 / 1500: loss 9.000947
iteration 200 / 1500: loss 9.000713
iteration 300 / 1500: loss 9.001039
iteration 400 / 1500: loss 9.000500
iteration 500 / 1500: loss 9.000798
iteration 600 / 1500: loss 9.000556
iteration 700 / 1500: loss 9.000353
iteration 800 / 1500: loss 9.000114
iteration 900 / 1500: loss 9.000345
iteration 1000 / 1500: loss 9.000372
iteration 1100 / 1500: loss 9.000221
iteration 1200 / 1500: loss 8.999915
iteration 1300 / 1500: loss 9.000171
iteration 1400 / 1500: loss 9.000225
iteration 0 / 1500: loss 9.000632
```

```
iteration 100 / 1500: loss 9.000571
iteration 200 / 1500: loss 9.000406
iteration 300 / 1500: loss 9.000326
iteration 400 / 1500: loss 9.000164
iteration 500 / 1500: loss 9.000282
iteration 600 / 1500: loss 9.000338
iteration 700 / 1500: loss 9.000222
iteration 800 / 1500: loss 9.000078
iteration 900 / 1500: loss 9.000173
iteration 1000 / 1500: loss 9.000067
iteration 1100 / 1500: loss 9.000151
iteration 1200 / 1500: loss 9.000103
iteration 1300 / 1500: loss 9.000045
iteration 1400 / 1500: loss 8.999932
iteration 0 / 1500: loss 9.000465
iteration 100 / 1500: loss 9.000278
iteration 200 / 1500: loss 9.000186
iteration 300 / 1500: loss 9.000192
iteration 400 / 1500: loss 9.000100
iteration 500 / 1500: loss 9.000016
iteration 600 / 1500: loss 9.000195
iteration 700 / 1500: loss 9.000157
iteration 800 / 1500: loss 9.000026
iteration 900 / 1500: loss 9.000066
iteration 1000 / 1500: loss 8.999845
iteration 1100 / 1500: loss 9.000055
iteration 1200 / 1500: loss 9.000013
iteration 1300 / 1500: loss 8.999968
iteration 1400 / 1500: loss 9.000007
iteration 0 / 1500: loss 9.000162
iteration 100 / 1500: loss 9.000267
iteration 200 / 1500: loss 9.000030
iteration 300 / 1500: loss 9.000082
iteration 400 / 1500: loss 9.000063
iteration 500 / 1500: loss 9.000135
iteration 600 / 1500: loss 9.000039
iteration 700 / 1500: loss 8.999861
iteration 800 / 1500: loss 9.000059
iteration 900 / 1500: loss 9.000003
iteration 1000 / 1500: loss 9.000014
iteration 1100 / 1500: loss 9.000031
iteration 1200 / 1500: loss 9.000063
iteration 1300 / 1500: loss 9.000031
iteration 1400 / 1500: loss 9.000080
iteration 0 / 1500: loss 9.000151
iteration 100 / 1500: loss 9.000128
iteration 200 / 1500: loss 9.000052
iteration 300 / 1500: loss 9.000044
```

```
iteration 400 / 1500: loss 9.000059
iteration 500 / 1500: loss 8.999935
iteration 600 / 1500: loss 9.000012
iteration 700 / 1500: loss 9.000010
iteration 800 / 1500: loss 9.000013
iteration 900 / 1500: loss 9.000057
iteration 1000 / 1500: loss 9.000027
iteration 1100 / 1500: loss 9.000024
iteration 1200 / 1500: loss 8.999995
iteration 1300 / 1500: loss 9.000071
iteration 1400 / 1500: loss 8.999976
iteration 0 / 1500: loss 9.000074
iteration 100 / 1500: loss 9.000071
iteration 200 / 1500: loss 9.000077
iteration 300 / 1500: loss 8.999959
iteration 400 / 1500: loss 9.000050
iteration 500 / 1500: loss 8.999986
iteration 600 / 1500: loss 9.000007
iteration 700 / 1500: loss 9.000065
iteration 800 / 1500: loss 9.000071
iteration 900 / 1500: loss 9.000080
iteration 1000 / 1500: loss 8.999963
iteration 1100 / 1500: loss 9.000015
iteration 1200 / 1500: loss 8.999955
iteration 1300 / 1500: loss 9.000034
iteration 1400 / 1500: loss 9.000031
iteration 0 / 1500: loss 9.000068
iteration 100 / 1500: loss 9.000108
iteration 200 / 1500: loss 8.999979
iteration 300 / 1500: loss 8.999975
iteration 400 / 1500: loss 8.999998
iteration 500 / 1500: loss 8.999972
iteration 600 / 1500: loss 8.999976
iteration 700 / 1500: loss 8.999996
iteration 800 / 1500: loss 8.999966
iteration 900 / 1500: loss 9.000011
iteration 1000 / 1500: loss 9.000018
iteration 1100 / 1500: loss 8.999964
iteration 1200 / 1500: loss 8.999989
iteration 1300 / 1500: loss 8.999963
iteration 1400 / 1500: loss 8.999979
iteration 0 / 1500: loss 8.999405
iteration 100 / 1500: loss 8.999164
iteration 200 / 1500: loss 8.999079
iteration 300 / 1500: loss 8.998926
iteration 400 / 1500: loss 8.998933
iteration 500 / 1500: loss 8.998563
iteration 600 / 1500: loss 8.998996
```

```
iteration 700 / 1500: loss 8.998907
iteration 800 / 1500: loss 8.998864
iteration 900 / 1500: loss 8.999049
iteration 1000 / 1500: loss 8.998779
iteration 1100 / 1500: loss 8.999279
iteration 1200 / 1500: loss 8.999020
iteration 1300 / 1500: loss 8.998779
iteration 1400 / 1500: loss 8.999078
iteration 0 / 1500: loss 9.001516
iteration 100 / 1500: loss 9.000385
iteration 200 / 1500: loss 9.000582
iteration 300 / 1500: loss 9.000592
iteration 400 / 1500: loss 9.000402
iteration 500 / 1500: loss 9.000530
iteration 600 / 1500: loss 9.000535
iteration 700 / 1500: loss 9.000201
iteration 800 / 1500: loss 9.000403
iteration 900 / 1500: loss 9.000057
iteration 1000 / 1500: loss 9.000388
iteration 1100 / 1500: loss 9.000138
iteration 1200 / 1500: loss 9.000444
iteration 1300 / 1500: loss 9.000282
iteration 1400 / 1500: loss 9.000119
iteration 0 / 1500: loss 9.001390
iteration 100 / 1500: loss 9.000951
iteration 200 / 1500: loss 9.000422
iteration 300 / 1500: loss 9.000341
iteration 400 / 1500: loss 9.000321
iteration 500 / 1500: loss 9.000552
iteration 600 / 1500: loss 9.000406
iteration 700 / 1500: loss 9.000022
iteration 800 / 1500: loss 9.000261
iteration 900 / 1500: loss 9.000434
iteration 1000 / 1500: loss 9.000377
iteration 1100 / 1500: loss 9.000375
iteration 1200 / 1500: loss 9.000125
iteration 1300 / 1500: loss 8.999910
iteration 1400 / 1500: loss 9.000120
iteration 0 / 1500: loss 9.000736
iteration 100 / 1500: loss 9.000555
iteration 200 / 1500: loss 9.000363
iteration 300 / 1500: loss 9.000144
iteration 400 / 1500: loss 9.000206
iteration 500 / 1500: loss 9.000014
iteration 600 / 1500: loss 9.000028
iteration 700 / 1500: loss 9.000155
iteration 800 / 1500: loss 9.000155
iteration 900 / 1500: loss 9.000009
```

```
iteration 1000 / 1500: loss 8.999890
iteration 1100 / 1500: loss 9.000178
iteration 1200 / 1500: loss 8.999911
iteration 1300 / 1500: loss 8.999957
iteration 1400 / 1500: loss 9.000068
iteration 0 / 1500: loss 9.000302
iteration 100 / 1500: loss 9.000138
iteration 200 / 1500: loss 9.000311
iteration 300 / 1500: loss 8.999977
iteration 400 / 1500: loss 9.000090
iteration 500 / 1500: loss 9.000095
iteration 600 / 1500: loss 9.000137
iteration 700 / 1500: loss 9.000023
iteration 800 / 1500: loss 8.999877
iteration 900 / 1500: loss 8.999980
iteration 1000 / 1500: loss 8.999983
iteration 1100 / 1500: loss 9.000192
iteration 1200 / 1500: loss 9.000017
iteration 1300 / 1500: loss 9.000086
iteration 1400 / 1500: loss 9.000204
iteration 0 / 1500: loss 9.000133
iteration 100 / 1500: loss 9.000135
iteration 200 / 1500: loss 9.000061
iteration 300 / 1500: loss 9.000108
iteration 400 / 1500: loss 9.000068
iteration 500 / 1500: loss 8.999892
iteration 600 / 1500: loss 9.000064
iteration 700 / 1500: loss 8.999935
iteration 800 / 1500: loss 9.000090
iteration 900 / 1500: loss 8.999953
iteration 1000 / 1500: loss 8.999897
iteration 1100 / 1500: loss 8.999963
iteration 1200 / 1500: loss 8.999976
iteration 1300 / 1500: loss 8.999864
iteration 1400 / 1500: loss 9.000017
iteration 0 / 1500: loss 9.000113
iteration 100 / 1500: loss 9.000208
iteration 200 / 1500: loss 8.999991
iteration 300 / 1500: loss 9.000108
iteration 400 / 1500: loss 8.999906
iteration 500 / 1500: loss 9.000085
iteration 600 / 1500: loss 8.999992
iteration 700 / 1500: loss 9.000027
iteration 800 / 1500: loss 9.000021
iteration 900 / 1500: loss 8.999993
iteration 1000 / 1500: loss 9.000050
iteration 1100 / 1500: loss 9.000035
iteration 1200 / 1500: loss 9.000032
```

```
iteration 1300 / 1500: loss 8.999953
iteration 1400 / 1500: loss 9.000057
iteration 0 / 1500: loss 9.000093
iteration 100 / 1500: loss 8.999983
iteration 200 / 1500: loss 8.999988
iteration 300 / 1500: loss 9.000011
iteration 400 / 1500: loss 8.999999
iteration 500 / 1500: loss 9.000013
iteration 600 / 1500: loss 9.000034
iteration 700 / 1500: loss 9.000010
iteration 800 / 1500: loss 9.000018
iteration 900 / 1500: loss 8.999987
iteration 1000 / 1500: loss 8.999993
iteration 1100 / 1500: loss 9.000051
iteration 1200 / 1500: loss 9.000014
iteration 1300 / 1500: loss 8.999904
iteration 1400 / 1500: loss 8.999985
iteration 0 / 1500: loss 9.000210
iteration 100 / 1500: loss 8.999962
iteration 200 / 1500: loss 8.999975
iteration 300 / 1500: loss 9.000018
iteration 400 / 1500: loss 8.999959
iteration 500 / 1500: loss 9.000052
iteration 600 / 1500: loss 9.000015
iteration 700 / 1500: loss 9.000044
iteration 800 / 1500: loss 9.000103
iteration 900 / 1500: loss 9.000015
iteration 1000 / 1500: loss 9.000030
iteration 1100 / 1500: loss 8.999999
iteration 1200 / 1500: loss 8.999961
iteration 1300 / 1500: loss 8.999962
iteration 1400 / 1500: loss 8.999923
iteration 0 / 1500: loss 8.999354
iteration 100 / 1500: loss 8.999074
iteration 200 / 1500: loss 8.999169
iteration 300 / 1500: loss 8.998942
iteration 400 / 1500: loss 8.999025
iteration 500 / 1500: loss 8.998635
iteration 600 / 1500: loss 8.998738
iteration 700 / 1500: loss 8.998708
iteration 800 / 1500: loss 8.998965
iteration 900 / 1500: loss 8.999150
iteration 1000 / 1500: loss 8.999266
iteration 1100 / 1500: loss 8.999494
iteration 1200 / 1500: loss 8.999050
iteration 1300 / 1500: loss 8.999165
iteration 1400 / 1500: loss 8.999308
iteration 0 / 1500: loss 9.001804
```

```
iteration 100 / 1500: loss 9.000857
iteration 200 / 1500: loss 9.000932
iteration 300 / 1500: loss 9.000783
iteration 400 / 1500: loss 9.000507
iteration 500 / 1500: loss 8.999977
iteration 600 / 1500: loss 9.000137
iteration 700 / 1500: loss 9.000731
iteration 800 / 1500: loss 9.000551
iteration 900 / 1500: loss 8.999999
iteration 1000 / 1500: loss 8.999777
iteration 1100 / 1500: loss 9.000251
iteration 1200 / 1500: loss 9.000418
iteration 1300 / 1500: loss 9.000632
iteration 1400 / 1500: loss 9.000215
iteration 0 / 1500: loss 9.000978
iteration 100 / 1500: loss 9.001054
iteration 200 / 1500: loss 9.000723
iteration 300 / 1500: loss 9.000569
iteration 400 / 1500: loss 9.000183
iteration 500 / 1500: loss 9.000193
iteration 600 / 1500: loss 9.000427
iteration 700 / 1500: loss 9.000055
iteration 800 / 1500: loss 9.000135
iteration 900 / 1500: loss 9.000019
iteration 1000 / 1500: loss 9.000098
iteration 1100 / 1500: loss 9.000041
iteration 1200 / 1500: loss 8.999813
iteration 1300 / 1500: loss 8.999843
iteration 1400 / 1500: loss 8.999877
iteration 0 / 1500: loss 9.000633
iteration 100 / 1500: loss 9.000518
iteration 200 / 1500: loss 9.000384
iteration 300 / 1500: loss 9.000191
iteration 400 / 1500: loss 9.000134
iteration 500 / 1500: loss 9.000290
iteration 600 / 1500: loss 8.999944
iteration 700 / 1500: loss 9.000165
iteration 800 / 1500: loss 9.000108
iteration 900 / 1500: loss 9.000036
iteration 1000 / 1500: loss 8.999883
iteration 1100 / 1500: loss 9.000110
iteration 1200 / 1500: loss 8.999886
iteration 1300 / 1500: loss 9.000061
iteration 1400 / 1500: loss 9.000116
iteration 0 / 1500: loss 9.000581
iteration 100 / 1500: loss 9.000357
iteration 200 / 1500: loss 9.000144
iteration 300 / 1500: loss 9.000148
```

```
iteration 400 / 1500: loss 8.999969
iteration 500 / 1500: loss 9.000091
iteration 600 / 1500: loss 9.000096
iteration 700 / 1500: loss 9.000051
iteration 800 / 1500: loss 8.999882
iteration 900 / 1500: loss 9.000152
iteration 1000 / 1500: loss 9.000172
iteration 1100 / 1500: loss 8.999998
iteration 1200 / 1500: loss 8.999954
iteration 1300 / 1500: loss 9.000014
iteration 1400 / 1500: loss 9.000091
iteration 0 / 1500: loss 9.000147
iteration 100 / 1500: loss 9.000230
iteration 200 / 1500: loss 9.000077
iteration 300 / 1500: loss 9.000022
iteration 400 / 1500: loss 9.000069
iteration 500 / 1500: loss 8.999980
iteration 600 / 1500: loss 8.999996
iteration 700 / 1500: loss 8.999978
iteration 800 / 1500: loss 9.000041
iteration 900 / 1500: loss 9.000038
iteration 1000 / 1500: loss 8.999966
iteration 1100 / 1500: loss 9.000000
iteration 1200 / 1500: loss 8.999964
iteration 1300 / 1500: loss 9.000018
iteration 1400 / 1500: loss 8.999882
iteration 0 / 1500: loss 9.000234
iteration 100 / 1500: loss 8.999959
iteration 200 / 1500: loss 9.000132
iteration 300 / 1500: loss 8.999949
iteration 400 / 1500: loss 9.000000
iteration 500 / 1500: loss 9.000133
iteration 600 / 1500: loss 9.000031
iteration 700 / 1500: loss 8.999942
iteration 800 / 1500: loss 8.999935
iteration 900 / 1500: loss 8.999954
iteration 1000 / 1500: loss 9.000021
iteration 1100 / 1500: loss 9.000015
iteration 1200 / 1500: loss 8.999980
iteration 1300 / 1500: loss 9.000085
iteration 1400 / 1500: loss 8.999989
iteration 0 / 1500: loss 9.000087
iteration 100 / 1500: loss 9.000117
iteration 200 / 1500: loss 8.999914
iteration 300 / 1500: loss 8.999906
iteration 400 / 1500: loss 9.000039
iteration 500 / 1500: loss 9.000010
iteration 600 / 1500: loss 9.000043
```

```
iteration 700 / 1500: loss 9.000068
iteration 800 / 1500: loss 8.999962
iteration 900 / 1500: loss 9.000022
iteration 1000 / 1500: loss 9.000110
iteration 1100 / 1500: loss 9.000024
iteration 1200 / 1500: loss 9.000015
iteration 1300 / 1500: loss 9.000029
iteration 1400 / 1500: loss 8.999953
iteration 0 / 1500: loss 9.000118
iteration 100 / 1500: loss 9.000121
iteration 200 / 1500: loss 9.000068
iteration 300 / 1500: loss 8.999976
iteration 400 / 1500: loss 9.000043
iteration 500 / 1500: loss 9.000028
iteration 600 / 1500: loss 8.999927
iteration 700 / 1500: loss 8.999967
iteration 800 / 1500: loss 8.999960
iteration 900 / 1500: loss 8.999946
iteration 1000 / 1500: loss 8.999955
iteration 1100 / 1500: loss 9.000060
iteration 1200 / 1500: loss 8.999997
iteration 1300 / 1500: loss 9.000013
iteration 1400 / 1500: loss 9.000012
iteration 0 / 1500: loss 8.999360
iteration 100 / 1500: loss 8.999177
iteration 200 / 1500: loss 8.998845
iteration 300 / 1500: loss 8.998852
iteration 400 / 1500: loss 8.998655
iteration 500 / 1500: loss 8.998862
iteration 600 / 1500: loss 8.998773
iteration 700 / 1500: loss 8.998813
iteration 800 / 1500: loss 8.998523
iteration 900 / 1500: loss 8.998902
iteration 1000 / 1500: loss 8.999260
iteration 1100 / 1500: loss 8.999291
iteration 1200 / 1500: loss 8.999272
iteration 1300 / 1500: loss 8.999463
iteration 1400 / 1500: loss 8.999391
iteration 0 / 1500: loss 9.001496
iteration 100 / 1500: loss 9.001094
iteration 200 / 1500: loss 9.001008
iteration 300 / 1500: loss 9.000918
iteration 400 / 1500: loss 9.000626
iteration 500 / 1500: loss 9.000454
iteration 600 / 1500: loss 9.000390
iteration 700 / 1500: loss 9.000621
iteration 800 / 1500: loss 9.000234
iteration 900 / 1500: loss 9.000078
```

```
iteration 1000 / 1500: loss 9.000648
iteration 1100 / 1500: loss 9.000192
iteration 1200 / 1500: loss 9.000243
iteration 1300 / 1500: loss 8.999862
iteration 1400 / 1500: loss 9.000098
iteration 0 / 1500: loss 9.001044
iteration 100 / 1500: loss 9.000700
iteration 200 / 1500: loss 9.000538
iteration 300 / 1500: loss 9.000193
iteration 400 / 1500: loss 9.000702
iteration 500 / 1500: loss 9.000259
iteration 600 / 1500: loss 9.000165
iteration 700 / 1500: loss 9.000196
iteration 800 / 1500: loss 9.000347
iteration 900 / 1500: loss 9.000155
iteration 1000 / 1500: loss 9.000103
iteration 1100 / 1500: loss 9.000201
iteration 1200 / 1500: loss 8.999965
iteration 1300 / 1500: loss 9.000048
iteration 1400 / 1500: loss 8.999899
iteration 0 / 1500: loss 9.000508
iteration 100 / 1500: loss 9.000615
iteration 200 / 1500: loss 9.000311
iteration 300 / 1500: loss 9.000263
iteration 400 / 1500: loss 9.000044
iteration 500 / 1500: loss 9.000285
iteration 600 / 1500: loss 9.000042
iteration 700 / 1500: loss 8.999988
iteration 800 / 1500: loss 9.000183
iteration 900 / 1500: loss 9.000039
iteration 1000 / 1500: loss 8.999919
iteration 1100 / 1500: loss 9.000133
iteration 1200 / 1500: loss 8.999904
iteration 1300 / 1500: loss 8.999999
iteration 1400 / 1500: loss 9.000031
iteration 0 / 1500: loss 9.000372
iteration 100 / 1500: loss 9.000160
iteration 200 / 1500: loss 9.000148
iteration 300 / 1500: loss 9.000096
iteration 400 / 1500: loss 8.999934
iteration 500 / 1500: loss 8.999905
iteration 600 / 1500: loss 9.000107
iteration 700 / 1500: loss 9.000073
iteration 800 / 1500: loss 8.999975
iteration 900 / 1500: loss 8.999910
iteration 1000 / 1500: loss 8.999987
iteration 1100 / 1500: loss 8.999958
iteration 1200 / 1500: loss 9.000043
```

```
iteration 1300 / 1500: loss 8.999911
iteration 1400 / 1500: loss 9.000129
iteration 0 / 1500: loss 9.000309
iteration 100 / 1500: loss 9.000126
iteration 200 / 1500: loss 9.000029
iteration 300 / 1500: loss 8.999984
iteration 400 / 1500: loss 9.000037
iteration 500 / 1500: loss 9.000187
iteration 600 / 1500: loss 8.999957
iteration 700 / 1500: loss 9.000031
iteration 800 / 1500: loss 9.000008
iteration 900 / 1500: loss 9.000083
iteration 1000 / 1500: loss 9.000044
iteration 1100 / 1500: loss 8.999995
iteration 1200 / 1500: loss 8.999885
iteration 1300 / 1500: loss 9.000095
iteration 1400 / 1500: loss 8.999950
iteration 0 / 1500: loss 9.000270
iteration 100 / 1500: loss 9.000064
iteration 200 / 1500: loss 9.000032
iteration 300 / 1500: loss 9.000001
iteration 400 / 1500: loss 9.000084
iteration 500 / 1500: loss 8.999920
iteration 600 / 1500: loss 9.000015
iteration 700 / 1500: loss 9.000057
iteration 800 / 1500: loss 9.000015
iteration 900 / 1500: loss 9.000129
iteration 1000 / 1500: loss 9.000037
iteration 1100 / 1500: loss 8.999893
iteration 1200 / 1500: loss 8.999987
iteration 1300 / 1500: loss 8.999952
iteration 1400 / 1500: loss 9.000115
iteration 0 / 1500: loss 9.000131
iteration 100 / 1500: loss 9.000012
iteration 200 / 1500: loss 9.000079
iteration 300 / 1500: loss 9.000021
iteration 400 / 1500: loss 9.000097
iteration 500 / 1500: loss 9.000006
iteration 600 / 1500: loss 9.000005
iteration 700 / 1500: loss 8.999959
iteration 800 / 1500: loss 9.000010
iteration 900 / 1500: loss 9.000041
iteration 1000 / 1500: loss 9.000012
iteration 1100 / 1500: loss 8.999990
iteration 1200 / 1500: loss 9.000027
iteration 1300 / 1500: loss 9.000003
iteration 1400 / 1500: loss 8.999978
iteration 0 / 1500: loss 9.000133
```

```
iteration 100 / 1500: loss 9.000083
iteration 200 / 1500: loss 9.000108
iteration 300 / 1500: loss 9.000013
iteration 400 / 1500: loss 8.999943
iteration 500 / 1500: loss 8.999953
iteration 600 / 1500: loss 8.999945
iteration 700 / 1500: loss 9.000009
iteration 800 / 1500: loss 9.000032
iteration 900 / 1500: loss 8.999940
iteration 1000 / 1500: loss 8.999979
iteration 1100 / 1500: loss 8.999988
iteration 1200 / 1500: loss 8.999997
iteration 1300 / 1500: loss 9.000118
iteration 1400 / 1500: loss 8.999957
iteration 0 / 1500: loss 8.999309
iteration 100 / 1500: loss 8.999101
iteration 200 / 1500: loss 8.998852
iteration 300 / 1500: loss 8.998802
iteration 400 / 1500: loss 8.999037
iteration 500 / 1500: loss 8.998756
iteration 600 / 1500: loss 8.998998
iteration 700 / 1500: loss 8.999015
iteration 800 / 1500: loss 8.999095
iteration 900 / 1500: loss 8.999437
iteration 1000 / 1500: loss 8.999079
iteration 1100 / 1500: loss 8.999576
iteration 1200 / 1500: loss 8.999667
iteration 1300 / 1500: loss 8.999538
iteration 1400 / 1500: loss 8.999514
iteration 0 / 1500: loss 9.002116
iteration 100 / 1500: loss 9.001560
iteration 200 / 1500: loss 9.001224
iteration 300 / 1500: loss 9.000816
iteration 400 / 1500: loss 9.000443
iteration 500 / 1500: loss 9.000457
iteration 600 / 1500: loss 9.000468
iteration 700 / 1500: loss 9.000181
iteration 800 / 1500: loss 9.000221
iteration 900 / 1500: loss 9.000429
iteration 1000 / 1500: loss 9.000121
iteration 1100 / 1500: loss 9.000172
iteration 1200 / 1500: loss 9.000479
iteration 1300 / 1500: loss 9.000048
iteration 1400 / 1500: loss 9.000008
iteration 0 / 1500: loss 9.001270
iteration 100 / 1500: loss 9.000649
iteration 200 / 1500: loss 9.000666
iteration 300 / 1500: loss 9.000226
```

```
iteration 400 / 1500: loss 9.000192
iteration 500 / 1500: loss 9.000092
iteration 600 / 1500: loss 9.000010
iteration 700 / 1500: loss 9.000070
iteration 800 / 1500: loss 9.000246
iteration 900 / 1500: loss 8.999874
iteration 1000 / 1500: loss 9.000215
iteration 1100 / 1500: loss 8.999931
iteration 1200 / 1500: loss 9.000134
iteration 1300 / 1500: loss 9.000090
iteration 1400 / 1500: loss 9.000252
iteration 0 / 1500: loss 9.000420
iteration 100 / 1500: loss 9.000539
iteration 200 / 1500: loss 9.000182
iteration 300 / 1500: loss 9.000073
iteration 400 / 1500: loss 9.000363
iteration 500 / 1500: loss 9.000033
iteration 600 / 1500: loss 9.000039
iteration 700 / 1500: loss 9.000080
iteration 800 / 1500: loss 9.000086
iteration 900 / 1500: loss 8.999974
iteration 1000 / 1500: loss 8.999873
iteration 1100 / 1500: loss 8.999997
iteration 1200 / 1500: loss 9.000147
iteration 1300 / 1500: loss 9.000086
iteration 1400 / 1500: loss 9.000113
iteration 0 / 1500: loss 9.000355
iteration 100 / 1500: loss 9.000163
iteration 200 / 1500: loss 9.000096
iteration 300 / 1500: loss 9.000079
iteration 400 / 1500: loss 8.999894
iteration 500 / 1500: loss 9.000113
iteration 600 / 1500: loss 9.000032
iteration 700 / 1500: loss 9.000027
iteration 800 / 1500: loss 9.000174
iteration 900 / 1500: loss 9.000044
iteration 1000 / 1500: loss 8.999969
iteration 1100 / 1500: loss 8.999988
iteration 1200 / 1500: loss 8.999869
iteration 1300 / 1500: loss 8.999957
iteration 1400 / 1500: loss 9.000227
iteration 0 / 1500: loss 9.000141
iteration 100 / 1500: loss 9.000152
iteration 200 / 1500: loss 9.000108
iteration 300 / 1500: loss 8.999998
iteration 400 / 1500: loss 8.999998
iteration 500 / 1500: loss 8.999925
iteration 600 / 1500: loss 8.999929
```

```
iteration 700 / 1500: loss 8.999966
iteration 800 / 1500: loss 8.999951
iteration 900 / 1500: loss 9.000070
iteration 1000 / 1500: loss 9.000070
iteration 1100 / 1500: loss 8.999970
iteration 1200 / 1500: loss 9.000035
iteration 1300 / 1500: loss 9.000110
iteration 1400 / 1500: loss 9.000058
iteration 0 / 1500: loss 9.000065
iteration 100 / 1500: loss 9.000104
iteration 200 / 1500: loss 9.000001
iteration 300 / 1500: loss 9.000072
iteration 400 / 1500: loss 8.999960
iteration 500 / 1500: loss 9.000082
iteration 600 / 1500: loss 9.000006
iteration 700 / 1500: loss 8.999982
iteration 800 / 1500: loss 8.999953
iteration 900 / 1500: loss 9.000102
iteration 1000 / 1500: loss 8.999921
iteration 1100 / 1500: loss 9.000107
iteration 1200 / 1500: loss 8.999955
iteration 1300 / 1500: loss 9.000129
iteration 1400 / 1500: loss 9.000049
iteration 0 / 1500: loss 9.000207
iteration 100 / 1500: loss 9.000016
iteration 200 / 1500: loss 8.999985
iteration 300 / 1500: loss 9.000015
iteration 400 / 1500: loss 8.999949
iteration 500 / 1500: loss 9.000068
iteration 600 / 1500: loss 8.999943
iteration 700 / 1500: loss 9.000040
iteration 800 / 1500: loss 9.000099
iteration 900 / 1500: loss 9.000128
iteration 1000 / 1500: loss 9.000007
iteration 1100 / 1500: loss 9.000074
iteration 1200 / 1500: loss 9.000037
iteration 1300 / 1500: loss 8.999995
iteration 1400 / 1500: loss 9.000093
iteration 0 / 1500: loss 9.000150
iteration 100 / 1500: loss 8.999984
iteration 200 / 1500: loss 8.999928
iteration 300 / 1500: loss 8.999966
iteration 400 / 1500: loss 8.999883
iteration 500 / 1500: loss 8.999941
iteration 600 / 1500: loss 8.999988
iteration 700 / 1500: loss 8.999995
iteration 800 / 1500: loss 9.000021
iteration 900 / 1500: loss 8.999957
```

```
iteration 1000 / 1500: loss 8.999984
iteration 1100 / 1500: loss 8.999945
iteration 1200 / 1500: loss 8.999993
iteration 1300 / 1500: loss 9.000064
iteration 1400 / 1500: loss 9.000008
lr 1.000000e-08 reg 1.500000e+04 train accuracy: 0.132714 val accuracy: 0.140000
lr 1.000000e-08 reg 2.500000e+04 train accuracy: 0.135286 val accuracy: 0.144000
lr 1.000000e-08 reg 3.500000e+04 train accuracy: 0.139837 val accuracy: 0.151000
lr 1.000000e-08 reg 4.500000e+04 train accuracy: 0.147306 val accuracy: 0.157000
lr 1.000000e-08 reg 5.500000e+04 train accuracy: 0.163816 val accuracy: 0.181000
lr 1.000000e-08 reg 6.500000e+04 train accuracy: 0.199122 val accuracy: 0.212000
lr 1.000000e-08 reg 7.500000e+04 train accuracy: 0.275878 val accuracy: 0.282000
lr 1.000000e-08 reg 8.500000e+04 train accuracy: 0.373918 val accuracy: 0.376000
lr 1.000000e-08 reg 9.500000e+04 train accuracy: 0.410367 val accuracy: 0.423000
lr 2.000000e-08 reg 1.500000e+04 train accuracy: 0.415633 val accuracy: 0.424000
lr 2.000000e-08 reg 2.500000e+04 train accuracy: 0.416490 val accuracy: 0.421000
lr 2.000000e-08 reg 3.500000e+04 train accuracy: 0.417306 val accuracy: 0.424000
lr 2.000000e-08 reg 4.500000e+04 train accuracy: 0.416918 val accuracy: 0.421000
lr 2.000000e-08 reg 5.500000e+04 train accuracy: 0.415510 val accuracy: 0.417000
lr 2.000000e-08 reg 6.500000e+04 train accuracy: 0.416592 val accuracy: 0.422000
lr 2.000000e-08 reg 7.500000e+04 train accuracy: 0.416245 val accuracy: 0.420000
lr 2.000000e-08 reg 8.500000e+04 train accuracy: 0.415694 val accuracy: 0.416000
lr 2.000000e-08 reg 9.500000e+04 train accuracy: 0.414408 val accuracy: 0.410000
lr 3.000000e-08 reg 1.500000e+04 train accuracy: 0.415816 val accuracy: 0.421000
lr 3.000000e-08 reg 2.500000e+04 train accuracy: 0.415306 val accuracy: 0.422000
lr 3.000000e-08 reg 3.500000e+04 train accuracy: 0.415347 val accuracy: 0.421000
lr 3.000000e-08 reg 4.500000e+04 train accuracy: 0.415327 val accuracy: 0.424000
lr 3.000000e-08 reg 5.500000e+04 train accuracy: 0.416898 val accuracy: 0.421000
lr 3.000000e-08 reg 6.500000e+04 train accuracy: 0.413020 val accuracy: 0.407000
lr 3.000000e-08 reg 7.500000e+04 train accuracy: 0.415102 val accuracy: 0.415000
lr 3.000000e-08 reg 8.500000e+04 train accuracy: 0.416245 val accuracy: 0.419000
lr 3.000000e-08 reg 9.500000e+04 train accuracy: 0.412959 val accuracy: 0.411000
lr 4.000000e-08 reg 1.500000e+04 train accuracy: 0.414714 val accuracy: 0.417000
lr 4.000000e-08 reg 2.500000e+04 train accuracy: 0.416408 val accuracy: 0.419000
lr 4.000000e-08 reg 3.500000e+04 train accuracy: 0.416184 val accuracy: 0.415000
lr 4.000000e-08 reg 4.500000e+04 train accuracy: 0.417469 val accuracy: 0.420000
lr 4.000000e-08 reg 5.500000e+04 train accuracy: 0.415408 val accuracy: 0.420000
lr 4.000000e-08 reg 6.500000e+04 train accuracy: 0.415449 val accuracy: 0.415000
lr 4.000000e-08 reg 7.500000e+04 train accuracy: 0.414163 val accuracy: 0.415000
lr 4.000000e-08 reg 8.500000e+04 train accuracy: 0.416490 val accuracy: 0.414000
lr 4.000000e-08 reg 9.500000e+04 train accuracy: 0.413755 val accuracy: 0.418000
lr 5.000000e-08 reg 1.500000e+04 train accuracy: 0.415306 val accuracy: 0.418000
lr 5.000000e-08 reg 2.500000e+04 train accuracy: 0.415490 val accuracy: 0.416000
lr 5.000000e-08 reg 3.500000e+04 train accuracy: 0.415592 val accuracy: 0.415000
lr 5.000000e-08 reg 4.500000e+04 train accuracy: 0.415102 val accuracy: 0.422000
lr 5.000000e-08 reg 5.500000e+04 train accuracy: 0.414143 val accuracy: 0.411000
lr 5.000000e-08 reg 6.500000e+04 train accuracy: 0.414653 val accuracy: 0.417000
lr 5.000000e-08 reg 7.500000e+04 train accuracy: 0.415796 val accuracy: 0.419000
```

```
lr 5.000000e-08 reg 8.500000e+04 train accuracy: 0.416980 val accuracy: 0.417000
lr 5.000000e-08 reg 9.500000e+04 train accuracy: 0.413980 val accuracy: 0.409000
lr 6.000000e-08 reg 1.500000e+04 train accuracy: 0.416796 val accuracy: 0.423000
lr 6.000000e-08 reg 2.500000e+04 train accuracy: 0.416061 val accuracy: 0.420000
lr 6.000000e-08 reg 3.500000e+04 train accuracy: 0.413816 val accuracy: 0.415000
lr 6.000000e-08 reg 4.500000e+04 train accuracy: 0.415469 val accuracy: 0.417000
lr 6.000000e-08 reg 5.500000e+04 train accuracy: 0.416388 val accuracy: 0.416000
lr 6.000000e-08 reg 6.500000e+04 train accuracy: 0.414571 val accuracy: 0.422000
lr 6.000000e-08 reg 7.500000e+04 train accuracy: 0.414061 val accuracy: 0.413000
lr 6.000000e-08 reg 8.500000e+04 train accuracy: 0.414082 val accuracy: 0.418000
lr 6.000000e-08 reg 9.500000e+04 train accuracy: 0.414510 val accuracy: 0.415000
lr 7.000000e-08 reg 1.500000e+04 train accuracy: 0.414306 val accuracy: 0.420000
lr 7.000000e-08 reg 2.500000e+04 train accuracy: 0.415735 val accuracy: 0.419000
lr 7.000000e-08 reg 3.500000e+04 train accuracy: 0.415204 val accuracy: 0.413000
lr 7.000000e-08 reg 4.500000e+04 train accuracy: 0.415388 val accuracy: 0.421000
lr 7.000000e-08 reg 5.500000e+04 train accuracy: 0.415531 val accuracy: 0.424000
lr 7.000000e-08 reg 6.500000e+04 train accuracy: 0.414633 val accuracy: 0.410000
lr 7.000000e-08 reg 7.500000e+04 train accuracy: 0.415143 val accuracy: 0.426000
lr 7.000000e-08 reg 8.500000e+04 train accuracy: 0.415327 val accuracy: 0.415000
lr 7.000000e-08 reg 9.500000e+04 train accuracy: 0.415694 val accuracy: 0.416000
lr 8.000000e-08 reg 1.500000e+04 train accuracy: 0.415347 val accuracy: 0.416000
lr 8.000000e-08 reg 2.500000e+04 train accuracy: 0.415571 val accuracy: 0.417000
lr 8.000000e-08 reg 3.500000e+04 train accuracy: 0.413755 val accuracy: 0.411000
lr 8.000000e-08 reg 4.500000e+04 train accuracy: 0.415469 val accuracy: 0.413000
lr 8.000000e-08 reg 5.500000e+04 train accuracy: 0.414265 val accuracy: 0.413000
lr 8.000000e-08 reg 6.500000e+04 train accuracy: 0.414980 val accuracy: 0.421000
lr 8.000000e-08 reg 7.500000e+04 train accuracy: 0.415531 val accuracy: 0.418000
lr 8.000000e-08 reg 8.500000e+04 train accuracy: 0.416714 val accuracy: 0.423000
lr 8.000000e-08 reg 9.500000e+04 train accuracy: 0.413286 val accuracy: 0.421000
lr 9.000000e-08 reg 1.500000e+04 train accuracy: 0.413918 val accuracy: 0.417000
lr 9.000000e-08 reg 2.500000e+04 train accuracy: 0.415755 val accuracy: 0.415000
lr 9.000000e-08 reg 3.500000e+04 train accuracy: 0.415551 val accuracy: 0.417000
lr 9.000000e-08 reg 4.500000e+04 train accuracy: 0.411571 val accuracy: 0.417000
lr 9.000000e-08 reg 5.500000e+04 train accuracy: 0.413673 val accuracy: 0.417000
lr 9.000000e-08 reg 6.500000e+04 train accuracy: 0.413449 val accuracy: 0.406000
lr 9.000000e-08 reg 7.500000e+04 train accuracy: 0.412388 val accuracy: 0.407000
lr 9.000000e-08 reg 8.500000e+04 train accuracy: 0.415980 val accuracy: 0.413000
lr 9.000000e-08 reg 9.500000e+04 train accuracy: 0.416939 val accuracy: 0.419000
best validation accuracy achieved: 0.426000
```

[6]:
```
# Evaluate your trained SVM on the test set: you should be able to get at least␣
↪0.40
y_test_pred = best_svm.predict(X_test_feats)
test_accuracy = np.mean(y_test == y_test_pred)
print(test_accuracy)
```

```
0.419
```

[7]:
```
# 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 : In the visualization, I see some of the images are not in the correct classes. I think this is pretty make sense because this algorithm are learning from data, and this example give the some of the data incorrect answer, learning based algorithm can not get the correct result from incorrect data, based on these I think it is make sense.

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.

[8]:
```
# Preprocessing: Remove the bias dimension
# Make sure to run this cell only ONCE
print(X_train_feats.shape)
X_train_feats = X_train_feats[:, :-1]
X_val_feats = X_val_feats[:, :-1]
X_test_feats = X_test_feats[:, :-1]

print(X_train_feats.shape)
```

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

[9]:
```
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

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

#hss = [50, 75, 100]

# define the parameters which I think it will have good results in two layers NN
# To save time, I delete some parameters have which performance bad
lrs = [1e-2, 5e-2]
lr_decs = [1, 0.75]

# make the dictionary for the solver function's input
data = {
      'X_train': X_train_feats, # training data
      'y_train': y_train, # training labels
      'X_val': X_val_feats, # validation data
      'y_val': y_val, # validation labels
      'X_test': X_test_feats, # validation data
      'y_test': y_test# validation labels
}

best = 0

# go through every parameters and find out which is best.
for lr in lrs:
  for lr_dec in lr_decs:
    #model = TwoLayerNet(input_size, hidden_dim, num_classes)
    # update the solver to new parameters
    solver = Solver(net, data,
                    update_rule='sgd',
                    optim_config={
                    'learning_rate': lr,
                    },
                    lr_decay=lr_dec,
                    num_epochs=10, batch_size=100,
                    print_every=100)
    # train the model
    solver.train()
    # compare to the current best result to find out which is better
    if solver.best_val_acc > best:
      # save the best result to best_net
      best = solver.best_val_acc
      best_net = net


#pass
```

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

```
(Iteration 1 / 4900) loss: 2.302595
(Epoch 0 / 10) train acc: 0.099000; val_acc: 0.098000
(Iteration 101 / 4900) loss: 2.302623
(Iteration 201 / 4900) loss: 2.301677
(Iteration 301 / 4900) loss: 2.302123
(Iteration 401 / 4900) loss: 2.301247
(Epoch 1 / 10) train acc: 0.191000; val_acc: 0.185000
(Iteration 501 / 4900) loss: 2.299721
(Iteration 601 / 4900) loss: 2.297167
(Iteration 701 / 4900) loss: 2.291884
(Iteration 801 / 4900) loss: 2.279512
(Iteration 901 / 4900) loss: 2.270177
(Epoch 2 / 10) train acc: 0.256000; val_acc: 0.257000
(Iteration 1001 / 4900) loss: 2.234032
(Iteration 1101 / 4900) loss: 2.202931
(Iteration 1201 / 4900) loss: 2.116714
(Iteration 1301 / 4900) loss: 2.063071
(Iteration 1401 / 4900) loss: 2.062834
(Epoch 3 / 10) train acc: 0.279000; val_acc: 0.286000
(Iteration 1501 / 4900) loss: 1.905925
(Iteration 1601 / 4900) loss: 1.954232
(Iteration 1701 / 4900) loss: 1.919110
(Iteration 1801 / 4900) loss: 1.890601
(Iteration 1901 / 4900) loss: 1.816904
(Epoch 4 / 10) train acc: 0.361000; val_acc: 0.337000
(Iteration 2001 / 4900) loss: 1.886741
(Iteration 2101 / 4900) loss: 1.778404
(Iteration 2201 / 4900) loss: 1.733758
(Iteration 2301 / 4900) loss: 1.631332
(Iteration 2401 / 4900) loss: 1.627314
(Epoch 5 / 10) train acc: 0.399000; val_acc: 0.406000
(Iteration 2501 / 4900) loss: 1.704783
(Iteration 2601 / 4900) loss: 1.576779
(Iteration 2701 / 4900) loss: 1.477441
(Iteration 2801 / 4900) loss: 1.700936
(Iteration 2901 / 4900) loss: 1.751340
(Epoch 6 / 10) train acc: 0.443000; val_acc: 0.424000
(Iteration 3001 / 4900) loss: 1.424293
(Iteration 3101 / 4900) loss: 1.571553
(Iteration 3201 / 4900) loss: 1.482227
(Iteration 3301 / 4900) loss: 1.413670
(Iteration 3401 / 4900) loss: 1.696482
(Epoch 7 / 10) train acc: 0.471000; val_acc: 0.451000
(Iteration 3501 / 4900) loss: 1.417008
(Iteration 3601 / 4900) loss: 1.498639
```

```
(Iteration 3701 / 4900) loss: 1.412720
(Iteration 3801 / 4900) loss: 1.416810
(Iteration 3901 / 4900) loss: 1.346687
(Epoch 8 / 10) train acc: 0.482000; val_acc: 0.473000
(Iteration 4001 / 4900) loss: 1.459321
(Iteration 4101 / 4900) loss: 1.477582
(Iteration 4201 / 4900) loss: 1.371930
(Iteration 4301 / 4900) loss: 1.450581
(Iteration 4401 / 4900) loss: 1.350389
(Epoch 9 / 10) train acc: 0.489000; val_acc: 0.483000
(Iteration 4501 / 4900) loss: 1.424942
(Iteration 4601 / 4900) loss: 1.352685
(Iteration 4701 / 4900) loss: 1.267797
(Iteration 4801 / 4900) loss: 1.579761
(Epoch 10 / 10) train acc: 0.514000; val_acc: 0.501000
(Iteration 1 / 4900) loss: 1.341244
(Epoch 0 / 10) train acc: 0.508000; val_acc: 0.503000
(Iteration 101 / 4900) loss: 1.337918
(Iteration 201 / 4900) loss: 1.342404
(Iteration 301 / 4900) loss: 1.489987
(Iteration 401 / 4900) loss: 1.423070
(Epoch 1 / 10) train acc: 0.531000; val_acc: 0.499000
(Iteration 501 / 4900) loss: 1.343995
(Iteration 601 / 4900) loss: 1.548661
(Iteration 701 / 4900) loss: 1.710971
(Iteration 801 / 4900) loss: 1.348510
(Iteration 901 / 4900) loss: 1.333691
(Epoch 2 / 10) train acc: 0.516000; val_acc: 0.505000
(Iteration 1001 / 4900) loss: 1.299504
(Iteration 1101 / 4900) loss: 1.347384
(Iteration 1201 / 4900) loss: 1.394978
(Iteration 1301 / 4900) loss: 1.469263
(Iteration 1401 / 4900) loss: 1.639476
(Epoch 3 / 10) train acc: 0.522000; val_acc: 0.517000
(Iteration 1501 / 4900) loss: 1.420589
(Iteration 1601 / 4900) loss: 1.408130
(Iteration 1701 / 4900) loss: 1.422713
(Iteration 1801 / 4900) loss: 1.447974
(Iteration 1901 / 4900) loss: 1.339378
(Epoch 4 / 10) train acc: 0.539000; val_acc: 0.518000
(Iteration 2001 / 4900) loss: 1.441989
(Iteration 2101 / 4900) loss: 1.329818
(Iteration 2201 / 4900) loss: 1.300148
(Iteration 2301 / 4900) loss: 1.260843
(Iteration 2401 / 4900) loss: 1.282459
(Epoch 5 / 10) train acc: 0.516000; val_acc: 0.513000
(Iteration 2501 / 4900) loss: 1.491033
(Iteration 2601 / 4900) loss: 1.301544
```

```
(Iteration 2701 / 4900) loss: 1.299889
(Iteration 2801 / 4900) loss: 1.480548
(Iteration 2901 / 4900) loss: 1.461976
(Epoch 6 / 10) train acc: 0.538000; val_acc: 0.520000
(Iteration 3001 / 4900) loss: 1.441370
(Iteration 3101 / 4900) loss: 1.274232
(Iteration 3201 / 4900) loss: 1.429270
(Iteration 3301 / 4900) loss: 1.297921
(Iteration 3401 / 4900) loss: 1.424711
(Epoch 7 / 10) train acc: 0.555000; val_acc: 0.518000
(Iteration 3501 / 4900) loss: 1.514184
(Iteration 3601 / 4900) loss: 1.379993
(Iteration 3701 / 4900) loss: 1.332097
(Iteration 3801 / 4900) loss: 1.136557
(Iteration 3901 / 4900) loss: 1.407635
(Epoch 8 / 10) train acc: 0.517000; val_acc: 0.517000
(Iteration 4001 / 4900) loss: 1.600676
(Iteration 4101 / 4900) loss: 1.226623
(Iteration 4201 / 4900) loss: 1.451558
(Iteration 4301 / 4900) loss: 1.580824
(Iteration 4401 / 4900) loss: 1.391461
(Epoch 9 / 10) train acc: 0.537000; val_acc: 0.510000
(Iteration 4501 / 4900) loss: 1.593912
(Iteration 4601 / 4900) loss: 1.507316
(Iteration 4701 / 4900) loss: 1.311261
(Iteration 4801 / 4900) loss: 1.270675
(Epoch 10 / 10) train acc: 0.552000; val_acc: 0.510000
(Iteration 1 / 4900) loss: 1.440319
(Epoch 0 / 10) train acc: 0.517000; val_acc: 0.514000
(Iteration 101 / 4900) loss: 1.054148
(Iteration 201 / 4900) loss: 1.382422
(Iteration 301 / 4900) loss: 1.349976
(Iteration 401 / 4900) loss: 1.291422
(Epoch 1 / 10) train acc: 0.533000; val_acc: 0.521000
(Iteration 501 / 4900) loss: 1.384404
(Iteration 601 / 4900) loss: 1.231072
(Iteration 701 / 4900) loss: 1.193242
(Iteration 801 / 4900) loss: 1.343614
(Iteration 901 / 4900) loss: 1.068817
(Epoch 2 / 10) train acc: 0.534000; val_acc: 0.525000
(Iteration 1001 / 4900) loss: 1.261494
(Iteration 1101 / 4900) loss: 1.374296
(Iteration 1201 / 4900) loss: 1.439104
(Iteration 1301 / 4900) loss: 1.116205
(Iteration 1401 / 4900) loss: 1.215486
(Epoch 3 / 10) train acc: 0.587000; val_acc: 0.550000
(Iteration 1501 / 4900) loss: 1.258649
(Iteration 1601 / 4900) loss: 1.059182
```

```
(Iteration 1701 / 4900) loss: 1.316627
(Iteration 1801 / 4900) loss: 1.201926
(Iteration 1901 / 4900) loss: 1.212856
(Epoch 4 / 10) train acc: 0.571000; val_acc: 0.550000
(Iteration 2001 / 4900) loss: 1.374807
(Iteration 2101 / 4900) loss: 1.133893
(Iteration 2201 / 4900) loss: 1.178429
(Iteration 2301 / 4900) loss: 1.365748
(Iteration 2401 / 4900) loss: 0.989399
(Epoch 5 / 10) train acc: 0.594000; val_acc: 0.558000
(Iteration 2501 / 4900) loss: 1.095078
(Iteration 2601 / 4900) loss: 0.975368
(Iteration 2701 / 4900) loss: 1.217238
(Iteration 2801 / 4900) loss: 1.250504
(Iteration 2901 / 4900) loss: 1.095912
(Epoch 6 / 10) train acc: 0.584000; val_acc: 0.565000
(Iteration 3001 / 4900) loss: 1.123875
(Iteration 3101 / 4900) loss: 0.976545
(Iteration 3201 / 4900) loss: 1.087850
(Iteration 3301 / 4900) loss: 1.190932
(Iteration 3401 / 4900) loss: 1.171675
(Epoch 7 / 10) train acc: 0.584000; val_acc: 0.569000
(Iteration 3501 / 4900) loss: 1.127085
(Iteration 3601 / 4900) loss: 1.130063
(Iteration 3701 / 4900) loss: 1.010154
(Iteration 3801 / 4900) loss: 1.054917
(Iteration 3901 / 4900) loss: 1.103505
(Epoch 8 / 10) train acc: 0.655000; val_acc: 0.581000
(Iteration 4001 / 4900) loss: 0.912129
(Iteration 4101 / 4900) loss: 1.068557
(Iteration 4201 / 4900) loss: 1.013715
(Iteration 4301 / 4900) loss: 1.224477
(Iteration 4401 / 4900) loss: 1.119996
(Epoch 9 / 10) train acc: 0.631000; val_acc: 0.588000
(Iteration 4501 / 4900) loss: 1.106364
(Iteration 4601 / 4900) loss: 0.900708
(Iteration 4701 / 4900) loss: 1.087298
(Iteration 4801 / 4900) loss: 0.937114
(Epoch 10 / 10) train acc: 0.656000; val_acc: 0.576000
(Iteration 1 / 4900) loss: 0.832427
(Epoch 0 / 10) train acc: 0.635000; val_acc: 0.588000
(Iteration 101 / 4900) loss: 1.119595
(Iteration 201 / 4900) loss: 1.024039
(Iteration 301 / 4900) loss: 0.983168
(Iteration 401 / 4900) loss: 0.903544
(Epoch 1 / 10) train acc: 0.664000; val_acc: 0.577000
(Iteration 501 / 4900) loss: 0.995646
(Iteration 601 / 4900) loss: 0.945463
```

```
(Iteration 701 / 4900) loss: 0.914078
(Iteration 801 / 4900) loss: 1.108749
(Iteration 901 / 4900) loss: 1.019080
(Epoch 2 / 10) train acc: 0.647000; val_acc: 0.584000
(Iteration 1001 / 4900) loss: 0.996854
(Iteration 1101 / 4900) loss: 0.946615
(Iteration 1201 / 4900) loss: 1.061041
(Iteration 1301 / 4900) loss: 0.927422
(Iteration 1401 / 4900) loss: 0.928551
(Epoch 3 / 10) train acc: 0.674000; val_acc: 0.596000
(Iteration 1501 / 4900) loss: 1.188922
(Iteration 1601 / 4900) loss: 0.863982
(Iteration 1701 / 4900) loss: 0.906916
(Iteration 1801 / 4900) loss: 0.984361
(Iteration 1901 / 4900) loss: 1.113104
(Epoch 4 / 10) train acc: 0.653000; val_acc: 0.585000
(Iteration 2001 / 4900) loss: 0.965125
(Iteration 2101 / 4900) loss: 0.958011
(Iteration 2201 / 4900) loss: 0.910412
(Iteration 2301 / 4900) loss: 0.968846
(Iteration 2401 / 4900) loss: 1.032792
(Epoch 5 / 10) train acc: 0.678000; val_acc: 0.594000
(Iteration 2501 / 4900) loss: 1.068301
(Iteration 2601 / 4900) loss: 1.049778
(Iteration 2701 / 4900) loss: 1.063213
(Iteration 2801 / 4900) loss: 1.044234
(Iteration 2901 / 4900) loss: 1.259086
(Epoch 6 / 10) train acc: 0.692000; val_acc: 0.586000
(Iteration 3001 / 4900) loss: 1.019418
(Iteration 3101 / 4900) loss: 0.948876
(Iteration 3201 / 4900) loss: 1.006792
(Iteration 3301 / 4900) loss: 0.974333
(Iteration 3401 / 4900) loss: 0.970310
(Epoch 7 / 10) train acc: 0.673000; val_acc: 0.591000
(Iteration 3501 / 4900) loss: 0.811192
(Iteration 3601 / 4900) loss: 1.040137
(Iteration 3701 / 4900) loss: 1.024643
(Iteration 3801 / 4900) loss: 0.856033
(Iteration 3901 / 4900) loss: 0.883005
(Epoch 8 / 10) train acc: 0.693000; val_acc: 0.591000
(Iteration 4001 / 4900) loss: 0.721339
(Iteration 4101 / 4900) loss: 0.876666
(Iteration 4201 / 4900) loss: 0.977559
(Iteration 4301 / 4900) loss: 1.004469
(Iteration 4401 / 4900) loss: 1.030922
(Epoch 9 / 10) train acc: 0.686000; val_acc: 0.592000
(Iteration 4501 / 4900) loss: 0.935271
(Iteration 4601 / 4900) loss: 0.911695
```

```
(Iteration 4701 / 4900) loss: 0.858153
(Iteration 4801 / 4900) loss: 0.750351
(Epoch 10 / 10) train acc: 0.693000; val_acc: 0.593000
```

[10]:
```
# Run your best neural net classifier on the test set. You should be able
# to get more than 55% accuracy.

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

```
0.574
```