knn

October 6, 2023

[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 = 'cs231n/assignments/assignment1/'
assert FOLDERNAME is not None, "[!] Enter the foldername."

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

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

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

1 k-Nearest Neighbor (kNN) exercise

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

The kNN classifier consists of two stages:

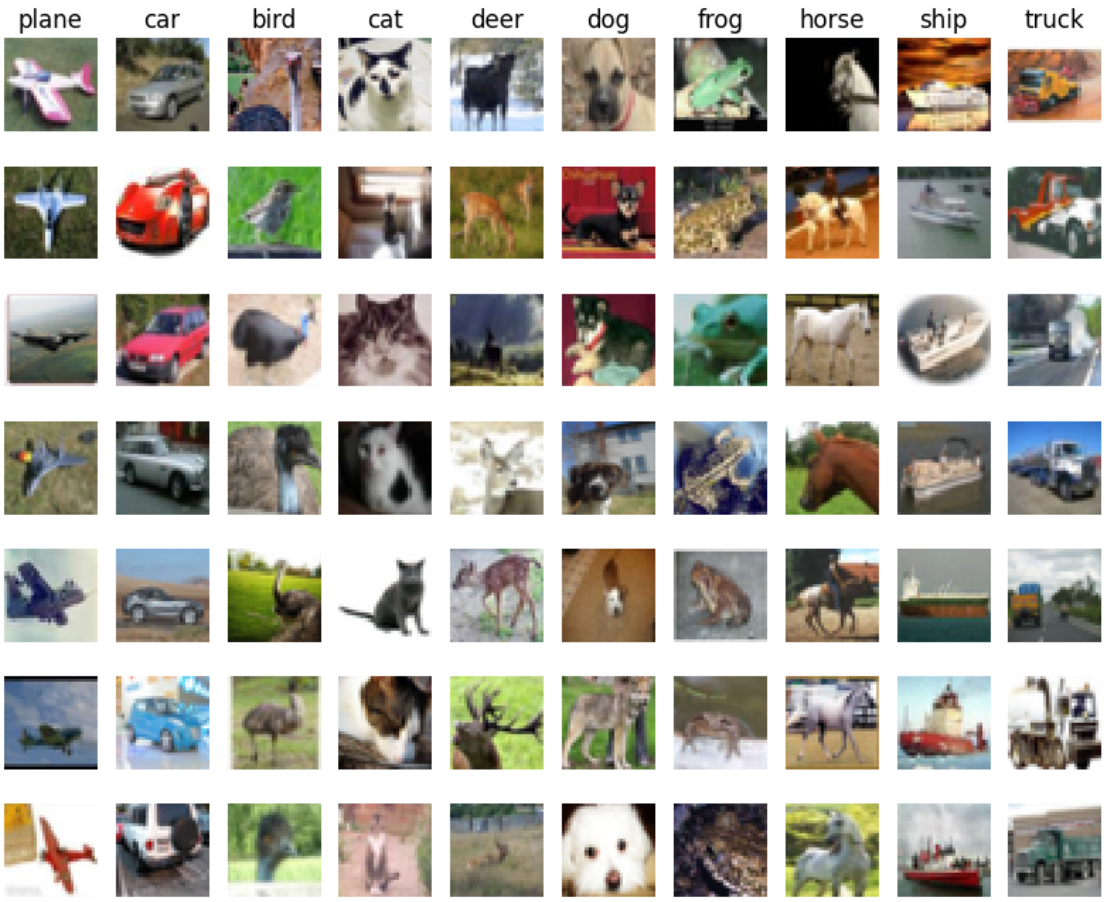
- During training, the classifier takes the training data and simply remembers it
- During testing, kNN classifies every test image by comparing to all training images and transfering the labels of the k most similar training examples
- The value of k is cross-validated

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

[2]:
```
# 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
```

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

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

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

# As a sanity check, we print out the size of the training and test data.
# 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('````````````````````````````````````````````````')
```

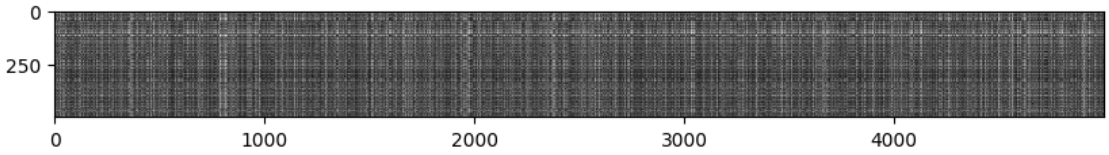
```
||||||||||||||||||||||||||||||||||||||||||||||||
Training data shape:  (50000, 32, 32, 3)
Training labels shape:  (50000,)
Test data shape:  (10000, 32, 32, 3)
Test labels shape:  (10000,)
``````````````````````````````````````````````````
```

[4]:
```
# 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('``')
```

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




```
```

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

[6]:
```
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.

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

```
```

[8]:
```
# 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?

*Your Answer :* - The bright rows results from picture similarity mismatch between test samples and training samples in terms of their visual content. In other words, if a test image differs significantly from all the training images, it will lead to a bright row.

- Bright columns arise when the training images do not closely match the test images. In simpler terms, if all the test images greatly differ from a particular training image, it will result in a bright column.

[9]:
```
# 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`:

[10]:
```
# 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* : Preprocessing steps that will not change the performance are: Option 1, Option 2, and Option 3

*YourExplanation* : 1. Mean subtraction - Subtracting the mean ($\mu$) from pixel values has minimal impact on the Nearest Neighbour classifier because it merely shifts pixel values, canceling out during distance calculations.

2. Per-pixel mean subtraction

- Subtracting per-pixel means doesn't affect the L1 distance or classifier performance, as the L1 distance remains unchanged: $L_1 = ||p_{ij} - q_{ij}^{(k)}||$.

3. Standartization (Subtracting the mean and dividing by the standard deviation)

- Standardization transforms data to have a mean of zero and a variance of one, resembling a standard normal distribution. This is useful when dealing with varying feature ranges, ensuring fair consideration of each feature's influence.
- It scales the L1 distance uniformly by a fixed factor, preserving nearest neighbors and not affecting classifier performance. While helping in gradient descent and training speed, it doesn't impact classifier accuracy here.

4. Per-pixel standartization

- Pixel-wise mean subtraction and division by standard deviation, a form of normalization, enhance classifier performance by increasing noise robustness.
- L1 distance inversely correlates with pixel-wise standard deviation

5. Rotating the coordinate axes of the data.

- Let's take two points $A = (2, 3)$ and $B = (5, 3)$. The L1 distance between $A$ and $B$ is: $||A - B|| = |2 - 5| + |3 - 3| = 3$ .If we rotate the coordinate axes by 60 degrees, the new coordinates become $A' = (2, 3)$ and $B' = (3.5, 1.73)$. The L1 distance between $A'$ and $B'$ is: $||A' - B'|| = |2 - 3.5| + |3 - 1.73| \approx 3.2$
- This demonstrates that rotating data coordinate axes results in a change in the L1 distance between points.

[11]:
```
# 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
``````````````````````````````````````````````
```

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

[13]:
```
# 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 43.134551 seconds
One loop version took 45.072177 seconds
No loop version took 0.623994 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.

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

# Splitting the feature data X_train into num_folds subsets.
X_train_folds = np.array_split(X_train, num_folds)
# Splitting the corresponding label data y_train into num_folds subsets.
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)*****

# Importing the KNearestNeighbor class from the cs231n.classifiers module
from cs231n.classifiers import KNearestNeighbor

# Iterating over different values of k, which are stored in k_choices
for k_current in k_choices:
    # Initializing empty list to store accuracies for k_current
    k_to_accuracies[k_current] = []

    # Performing k-fold cross-validation
    for validation_fold in range(num_folds):

        # Creating a new classifier object
        classifier = KNearestNeighbor()

        # Training the classifier using training data from all folds in␣
 ↪X_train_folds except the current validation fold
        # Concatenating training data from other folds and use it for training
        classifier.train(np.concatenate(X_train_folds[:validation_fold] +␣
 ↪X_train_folds[validation_fold + 1:]),
                         np.concatenate(y_train_folds[:validation_fold] +␣
 ↪y_train_folds[validation_fold + 1:]))

        # Using the trained classifier to predict labels for the data in␣
 ↪current validation fold
        y_pred_fold = classifier.predict(X_train_folds[validation_fold], k =␣
 ↪k_current)

        # Calculating the accuracy of the predictions for current validation␣
 ↪fold
```

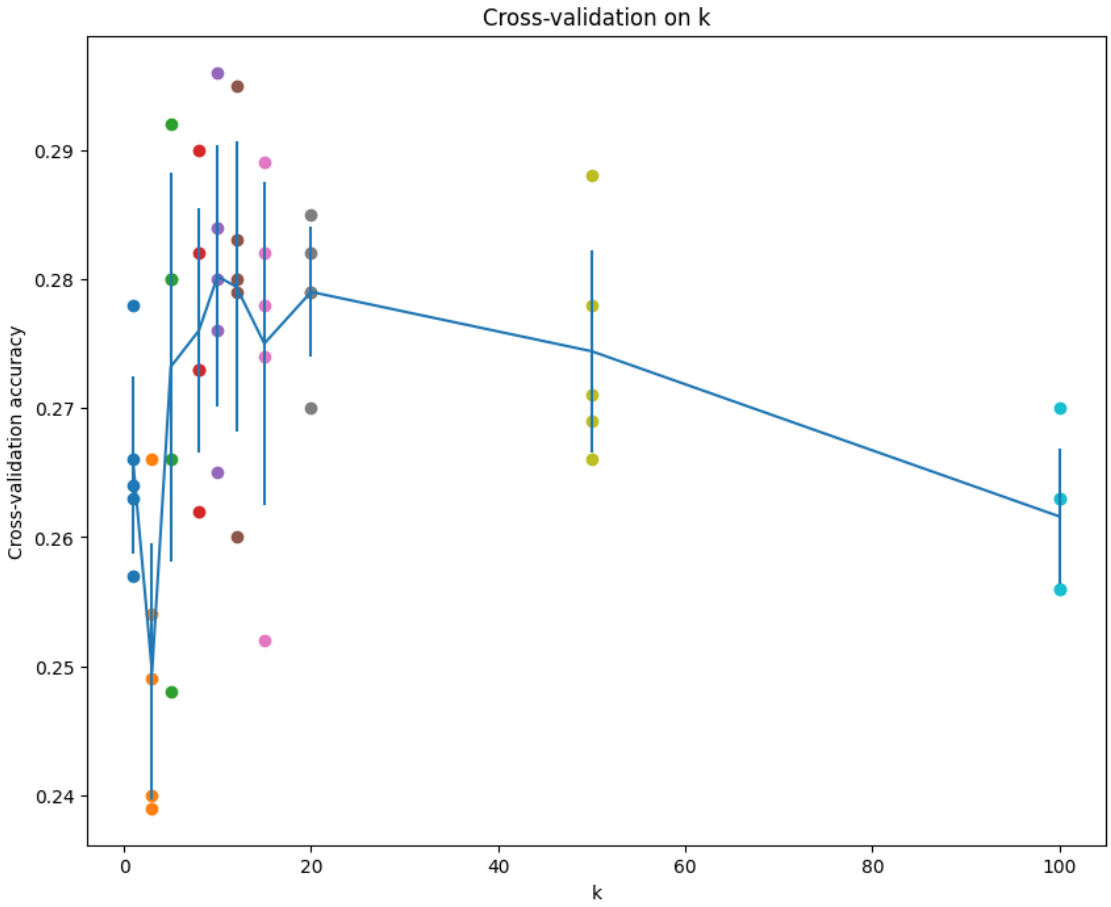
```
        accuracy = np.mean(y_pred_fold == y_train_folds[validation_fold])

        # Appending the accuracy to the list with the current k value so that␣
 ↪we get k and accuracy pairs in the dictionary.
        k_to_accuracies[k_current].append(accuracy)

pass

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

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

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

[15]:
```
# 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('````````````````````````````````````````````````')
```

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




```
```

[16]:
```
experimental_k = k_choices[accuracies_mean.argmax()]
print(experimental_k)
```

```
10
```

[17]:
```
# 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.

# Determining the optimal value for 'k' based on the results of␣
 ↪cross-validation.
experimental_k = k_choices[accuracies_mean.argmax()]

best_k = experimental_k
```

```
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 : True statements are 2,4.

YourExplanation :

1. False: The decision boundary of the k-NN classifier is not necessarily linear. It can be quite complex and non-linear, depending on the distribution of the training data.
2. True: The training error of a 1-NN (1-Nearest Neighbor) will always be lower than or equal to that of 5-NN (5-Nearest Neighbors) because 1-NN simply memorizes the training data, resulting in zero training error.
3. False: The test error of a 1-NN will not always be lower than that of a 5-NN. A 1-NN can be sensitive to noise and outliers, potentially leading to higher test error compared to 5-NN, which smooths out the decision boundary.
4. True: The time needed to classify a test example with the k-NN classifier grows with the size of the training set. As the training set becomes larger, k-NN needs to search through more data points to make a prediction, increasing the computational time.

svm

October 6, 2023

[ ]:
```
# 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 = 'cs231n/assignments/assignment1/'
assert FOLDERNAME is not None, "[!] Enter the foldername."

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

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

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

1 Multiclass Support Vector Machine exercise

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

In this exercise you will:

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

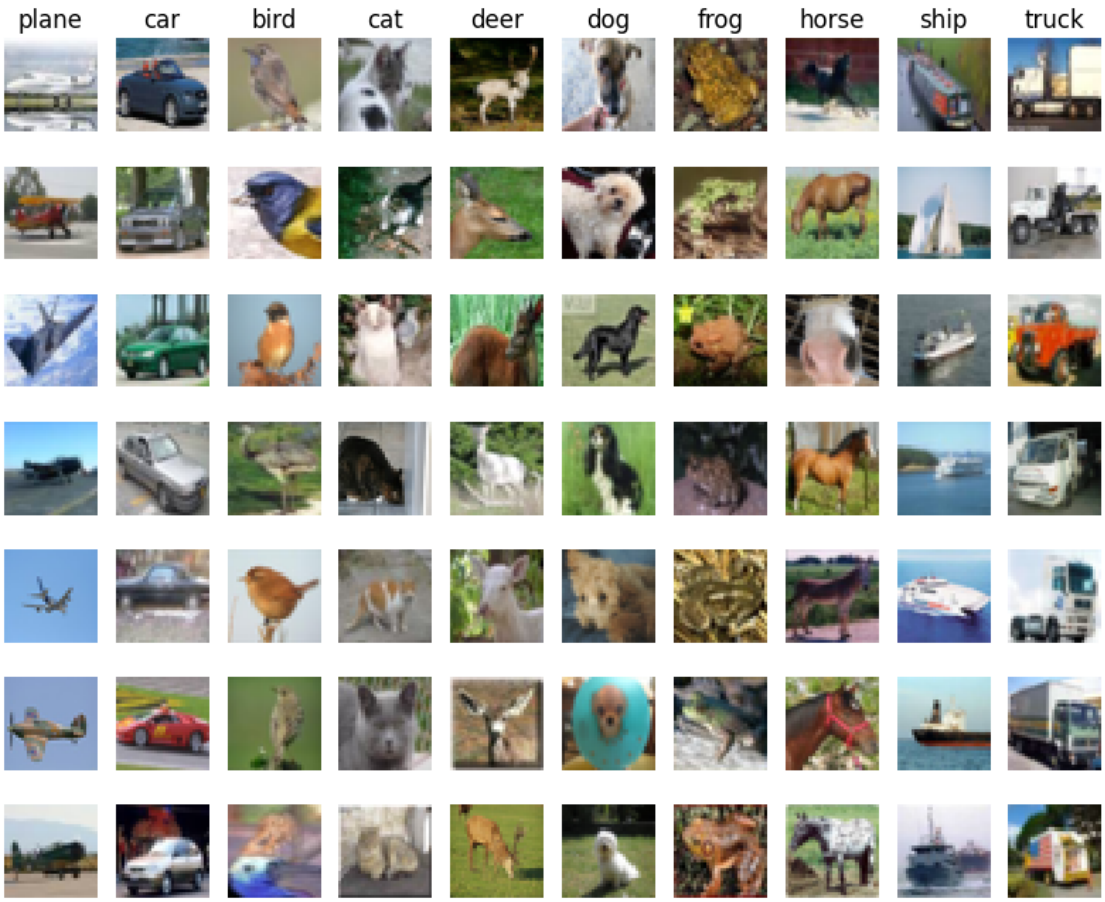
[ ]:
```
# Run some setup code for this notebook.
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

# 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,)
```

[ ]:
```
# 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)
```

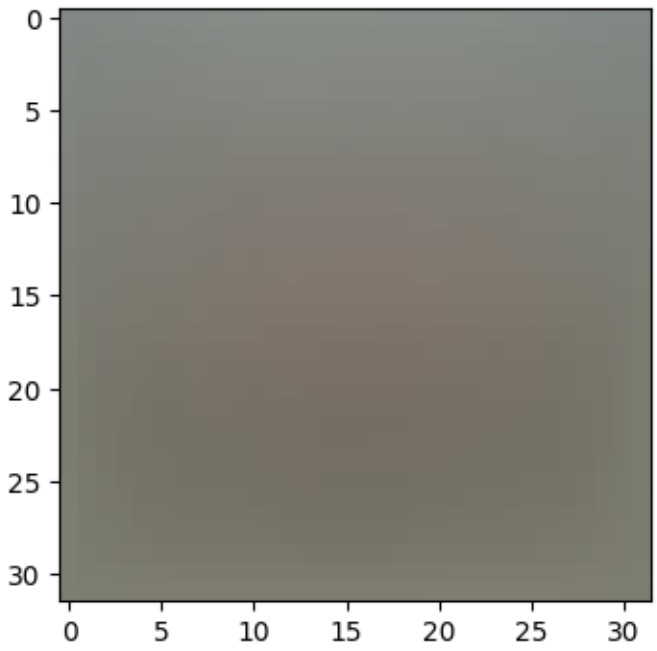
```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

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

```
loss: 8.967418
```

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: 11.042283 analytic: 0.000000, relative error: 1.000000e+00
numerical: 1.699520 analytic: 0.000000, relative error: 1.000000e+00
numerical: 6.855507 analytic: 0.000000, relative error: 1.000000e+00
numerical: -0.912618 analytic: 0.000000, relative error: 1.000000e+00
numerical: 9.911521 analytic: 0.000000, relative error: 1.000000e+00
numerical: 3.268000 analytic: 0.000000, relative error: 1.000000e+00
numerical: -10.220423 analytic: 0.000000, relative error: 1.000000e+00
numerical: -2.412000 analytic: 0.000000, relative error: 1.000000e+00
numerical: -6.367687 analytic: 0.000000, relative error: 1.000000e+00
numerical: -4.606000 analytic: 0.000000, relative error: 1.000000e+00
numerical: -14.004909 analytic: -0.001762, relative error: 9.997484e-01
numerical: -11.582655 analytic: 0.006174, relative error: 1.000000e+00
numerical: 8.354130 analytic: 0.002433, relative error: 9.994178e-01
numerical: -20.889944 analytic: 0.007170, relative error: 1.000000e+00
numerical: 16.217820 analytic: -0.000159, relative error: 1.000000e+00
numerical: -1.190195 analytic: 0.005412, relative error: 1.000000e+00
numerical: 1.330341 analytic: 0.011780, relative error: 9.824464e-01
numerical: 15.946821 analytic: -0.003691, relative error: 1.000000e+00
numerical: -8.981715 analytic: -0.004985, relative error: 9.988905e-01
numerical: -6.708374 analytic: -0.010764, relative error: 9.967959e-01
```

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 : The SVM loss function is defined as the maximum between 0 and the score value

subtracted from the margin, and it is not differentiable precisely at hinge point where margin equals 1. During gradient checks, it is expected that the analytical gradient and numerically approximated gradient may not match exactly.

To check if differentiation fails at the hinge point, both analytical and numerical methods can be used. The error between the two gradients is typically very small but not exactly zero.

Increasing the safety margin to avoid the exact hinge point (e.g., using margin of 1.01 instead of 1) could potentially mitigate the issue, but it comes at the cost of allowing more margin violations, which might lead to lower classification accuracy. Thus, there is a trade-off between achieving a smoother loss surface and maintaining the classification performance.

[ ]:
```
# 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: 8.967418e+00 computed in 0.018630s
Vectorized loss: 8.967418e+00 computed in 0.013200s
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 0.029247s
Vectorized loss and gradient: computed in 0.012363s
difference: 3040.613243
```

1.2.1 Stochastic Gradient Descent

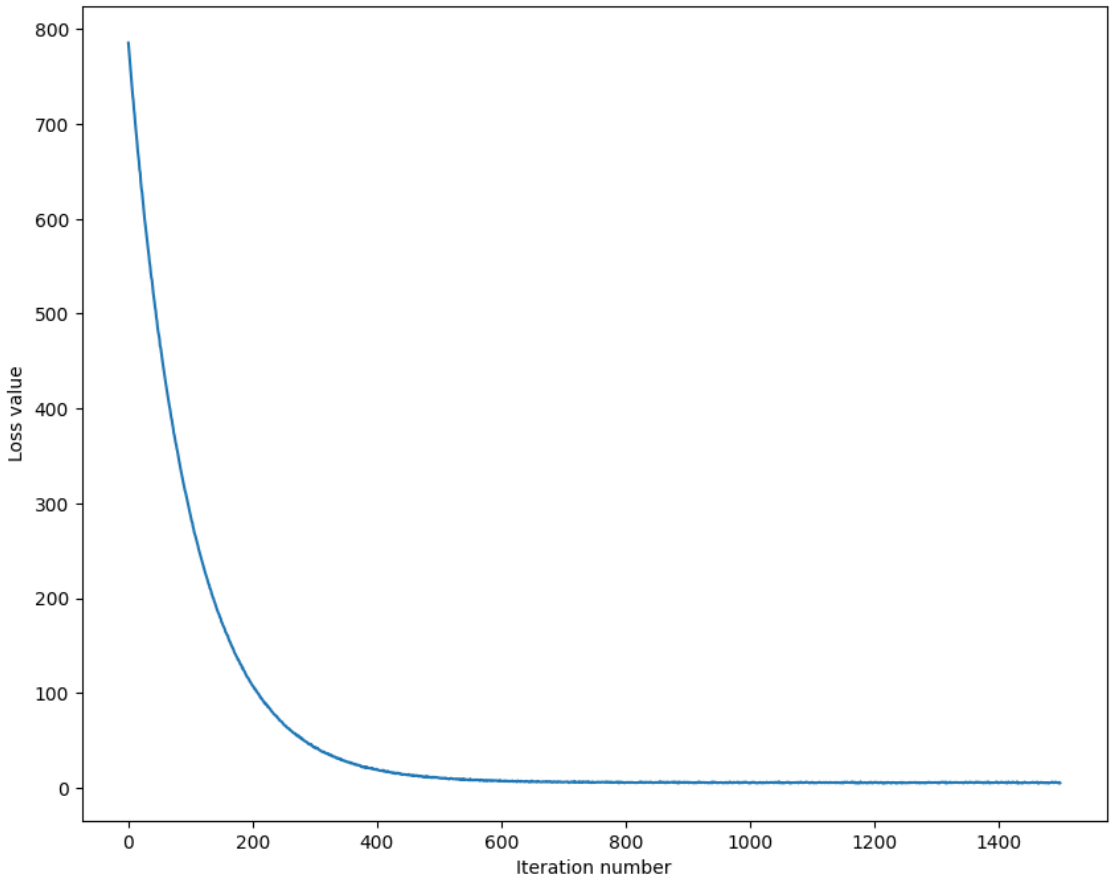
We now have vectorized and efficient expressions for the loss, the gradient and our gradient matches the numerical gradient. We are therefore ready to do SGD to minimize the loss. Your code for this part will be written inside `cs231n/classifiers/linear_classifier.py`.

[ ]:
```
# 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 785.672391
iteration 100 / 1500: loss 287.056242
iteration 200 / 1500: loss 107.367213
iteration 300 / 1500: loss 42.321357
iteration 400 / 1500: loss 19.346025
iteration 500 / 1500: loss 10.252051
iteration 600 / 1500: loss 6.860802
iteration 700 / 1500: loss 5.715234
iteration 800 / 1500: loss 5.439343
iteration 900 / 1500: loss 5.836427
iteration 1000 / 1500: loss 5.087377
iteration 1100 / 1500: loss 5.760094
iteration 1200 / 1500: loss 5.525966
iteration 1300 / 1500: loss 5.303209
iteration 1400 / 1500: loss 5.129233
That took 11.860566s
```

[ ]:
```
# 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: 0.367184
validation accuracy: 0.372000
```

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

# Provided as a reference. You may or may not want to change these␣
 ↪hyperparameters
learning_rates = [1e-7, 2e-7, 5e-8]
regularization_strengths = [2e4, 5e4]

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

# To find the optimal lr and reg strength we have to create 2 loops to figure␣
 ↪out the best hyperparameters to find the best softmax classifier.
best_learning_rate = 0
best_reg_strength = 0
# Looping over all combinations of hyperparameters
for lr in learning_rates:
    for reg in regularization_strengths:
        # Create a LinearSVM classifier object with the current hyperparameters
        svm = LinearSVM()

        # Training the classifier on the training data with 1200 iterations
        svm.train(X_train, y_train, learning_rate=lr, reg=reg,
                  num_iters=1200, verbose=True)
```

```
        # Prediction on the training and validation sets
        y_train_pred = svm.predict(X_train)
        y_val_pred = svm.predict(X_val)

        # Computing accuracy on training and validation sets
        train_accuracy = np.mean(y_train == y_train_pred)
        val_accuracy = np.mean(y_val == y_val_pred)

        # Storing the accuracy values in the results dictionary
        results[(lr, reg)] = (train_accuracy, val_accuracy)

        # Checking if this is the best validation accuracy so far and adding to
        if val_accuracy > best_val:
            best_val = val_accuracy
            best_svm = svm
            best_learning_rate = lr
            best_reg_strength = reg
print(f"Best Pair of lr and reg is (lr,reg):
 ↪{(best_learning_rate,best_reg_strength)}")

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)
```

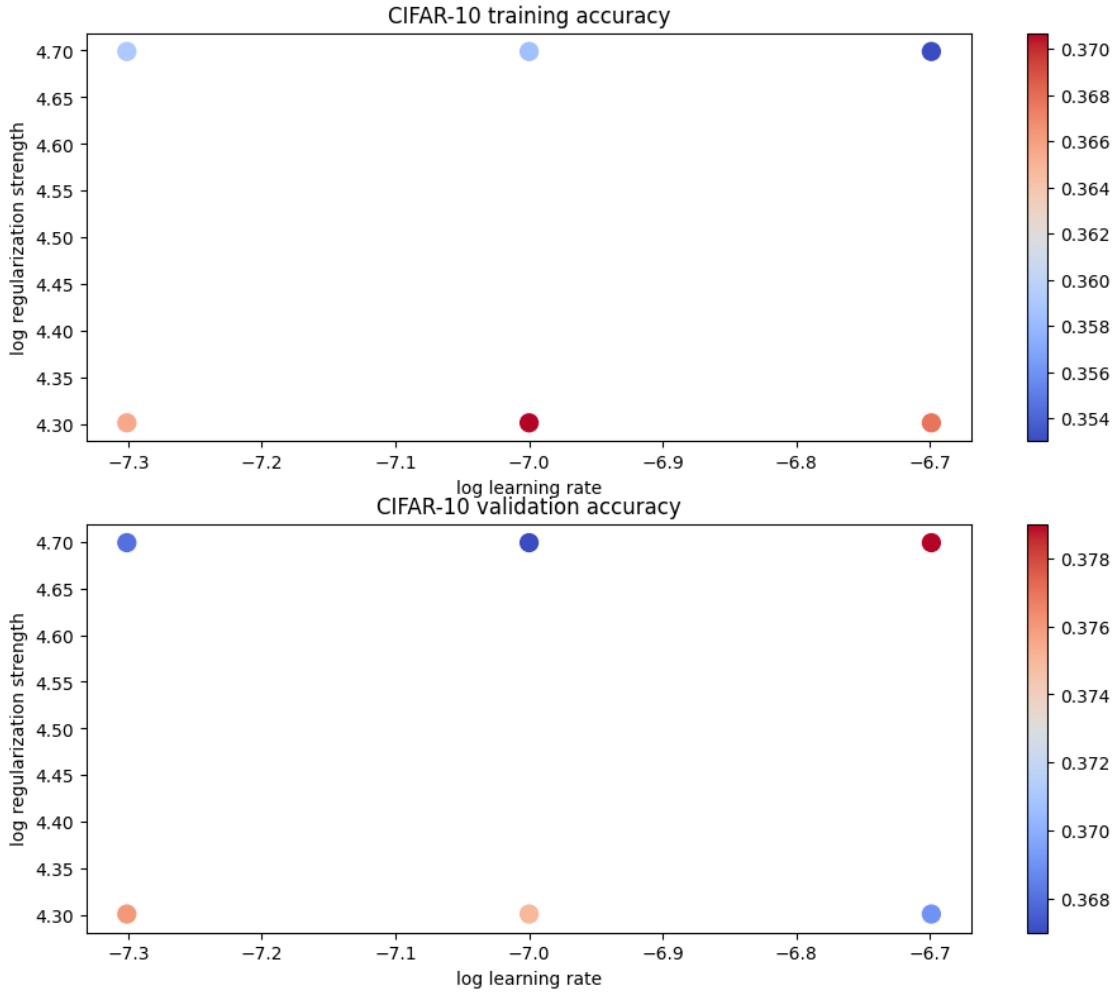
```
iteration 0 / 1200: loss 636.392334
iteration 100 / 1200: loss 282.463443
iteration 200 / 1200: loss 128.486229
iteration 300 / 1200: loss 60.529248
iteration 400 / 1200: loss 29.589954
iteration 500 / 1200: loss 15.707261
iteration 600 / 1200: loss 10.195858
iteration 700 / 1200: loss 7.366290
iteration 800 / 1200: loss 6.459745
iteration 900 / 1200: loss 5.428998
iteration 1000 / 1200: loss 5.239562
iteration 1100 / 1200: loss 5.415083
iteration 0 / 1200: loss 1552.683104
iteration 100 / 1200: loss 209.470476
```

```
iteration 200 / 1200: loss 32.854935
iteration 300 / 1200: loss 9.208821
iteration 400 / 1200: loss 6.303889
iteration 500 / 1200: loss 5.818498
iteration 600 / 1200: loss 5.472783
iteration 700 / 1200: loss 6.092746
iteration 800 / 1200: loss 5.401369
iteration 900 / 1200: loss 5.370357
iteration 1000 / 1200: loss 5.777847
iteration 1100 / 1200: loss 5.929584
iteration 0 / 1200: loss 635.229452
iteration 100 / 1200: loss 127.916151
iteration 200 / 1200: loss 30.352040
iteration 300 / 1200: loss 9.911900
iteration 400 / 1200: loss 6.132874
iteration 500 / 1200: loss 5.532101
iteration 600 / 1200: loss 5.743142
iteration 700 / 1200: loss 5.538841
iteration 800 / 1200: loss 4.813722
iteration 900 / 1200: loss 5.588044
iteration 1000 / 1200: loss 4.998336
iteration 1100 / 1200: loss 5.170718
iteration 0 / 1200: loss 1564.782100
iteration 100 / 1200: loss 32.586659
iteration 200 / 1200: loss 6.038992
iteration 300 / 1200: loss 5.853585
iteration 400 / 1200: loss 6.455551
iteration 500 / 1200: loss 5.598117
iteration 600 / 1200: loss 5.722990
iteration 700 / 1200: loss 5.473484
iteration 800 / 1200: loss 6.258077
iteration 900 / 1200: loss 6.151456
iteration 1000 / 1200: loss 5.568236
iteration 1100 / 1200: loss 5.370822
iteration 0 / 1200: loss 627.550705
iteration 100 / 1200: loss 417.789850
iteration 200 / 1200: loss 279.135987
iteration 300 / 1200: loss 188.053498
iteration 400 / 1200: loss 127.587273
iteration 500 / 1200: loss 87.013248
iteration 600 / 1200: loss 60.033360
iteration 700 / 1200: loss 41.272632
iteration 800 / 1200: loss 29.480405
iteration 900 / 1200: loss 21.738501
iteration 1000 / 1200: loss 15.624555
iteration 1100 / 1200: loss 12.226743
iteration 0 / 1200: loss 1546.822016
iteration 100 / 1200: loss 565.487708
```

```
iteration 200 / 1200: loss 209.495753
iteration 300 / 1200: loss 79.311019
iteration 400 / 1200: loss 32.432737
iteration 500 / 1200: loss 15.301361
iteration 600 / 1200: loss 8.676943
iteration 700 / 1200: loss 7.023456
iteration 800 / 1200: loss 5.886734
iteration 900 / 1200: loss 5.687791
iteration 1000 / 1200: loss 5.454899
iteration 1100 / 1200: loss 5.642962
Best Pair of lr and reg is (lr,reg):(2e-07, 50000.0)
lr 5.000000e-08 reg 2.000000e+04 train accuracy: 0.365633 val accuracy: 0.376000
lr 5.000000e-08 reg 5.000000e+04 train accuracy: 0.359061 val accuracy: 0.368000
lr 1.000000e-07 reg 2.000000e+04 train accuracy: 0.370694 val accuracy: 0.375000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.358408 val accuracy: 0.367000
lr 2.000000e-07 reg 2.000000e+04 train accuracy: 0.367755 val accuracy: 0.369000
lr 2.000000e-07 reg 5.000000e+04 train accuracy: 0.353000 val accuracy: 0.379000
best validation accuracy achieved during cross-validation: 0.379000
```

[ ]:
```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: 0.365000
```

[ ]:
```
# 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 visualization of SVM weights typically resembles a set of color maps or images, where each pixel corresponds to a feature in the input space. The color or intensity of each pixel represents the weight associated with that feature. These weights capture the importance of each feature in making classification decisions.

For exampkle, if we're using an SVM to classify images of objects, such as cars, the SVM weights for the "car" class might exhibit patterns resembling a car's front view. This is because the SVM has learned that specific features, like the shape of the car's headlights or grille, are significant for identifying cars in the dataset.

In essence, SVM weights visually show the distinctive characteristics and patterns in the training data that are crucial for distinguishing between different classes. They reflect what the SVM has learned about the important features that contribute to its classification decisions.

softmax

October 6, 2023

[ ]:
```
# 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 = 'cs231n/assignments/assignment1/'
assert FOLDERNAME is not None, "[!] Enter the foldername."

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

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

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

1 Softmax exercise

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

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

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

[ ]:
```
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
```

[ ]:
```
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`.

[ ]:
```
# 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: 2.347713
sanity check: 2.302585
```

Inline Question 1

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

Your Answer : At the beginning of training, before any learning has taken place, we anticipate that the initial loss will be approximately equal to -log(0.1). This expectation arises because, in the CIFAR-10 dataset with ten classes, each class is equally likely to be chosen, resulting in a 0.1 probability for the correct class. The softmax loss is essentially the negative logarithm of this probability, which yields -log(0.1)

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

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

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

```
numerical: 0.101299 analytic: 0.101299, relative error: 1.202176e-07
numerical: 1.001526 analytic: 1.001526, relative error: 4.890997e-09
numerical: -1.137471 analytic: -1.137471, relative error: 2.660656e-08
```

```
numerical: 1.897927 analytic: 1.897927, relative error: 2.339798e-08
numerical: -0.890519 analytic: -0.890519, relative error: 2.699743e-08
numerical: -0.578125 analytic: -0.578126, relative error: 6.979492e-09
numerical: 0.682306 analytic: 0.682306, relative error: 4.525685e-08
numerical: 0.141387 analytic: 0.141387, relative error: 1.088888e-07
numerical: -0.684046 analytic: -0.684046, relative error: 4.192710e-08
numerical: -0.149372 analytic: -0.149372, relative error: 1.419782e-07
numerical: -0.702057 analytic: -0.702057, relative error: 2.676690e-08
numerical: 0.459252 analytic: 0.459252, relative error: 6.931593e-08
numerical: 0.124031 analytic: 0.124031, relative error: 3.891745e-08
numerical: -1.111487 analytic: -1.111487, relative error: 1.153439e-08
numerical: -0.870222 analytic: -0.870222, relative error: 2.062644e-08
numerical: -0.732025 analytic: -0.732025, relative error: 3.117526e-08
numerical: 0.919055 analytic: 0.919055, relative error: 6.515832e-09
numerical: 0.173343 analytic: 0.173343, relative error: 1.385261e-07
numerical: 1.043050 analytic: 1.043050, relative error: 8.570468e-10
numerical: -1.910165 analytic: -1.910165, relative error: 9.833835e-09
```

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

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

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

```
naive loss: 2.347713e+00 computed in 0.095631s
vectorized loss: 2.347713e+00 computed in 0.013473s
Loss difference: 0.000000
Gradient difference: 0.000000
```

[ ]:
```
# 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-8, 2e-7, 1e-7, 2e-6, 3e-5]
regularization_strengths = [1e3, 3.5e4, 5e4]

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

# This is simply doing a grid search over hyperparameters.
# To find the optimal lr and reg strength we have to create 2 loops to figure␣
 ↪out the best hyperparameters to find the best softmax classifier.
best_learning_rate = 0
best_reg_strength = 0
# Iterate over different combinations of hyperparameters
for lr in learning_rates:
    for reg in regularization_strengths:
        # Creating a Softmax classifier instance
        softmax = Softmax()

        # Training the Softmax classifier on the training data for 1500␣
 ↪iterations
        softmax.train(X_train, y_train, learning_rate=lr, reg=reg,
                      num_iters=1500, verbose=True)

        # Predicting on the training and validation sets
        y_train_pred = softmax.predict(X_train)
        y_val_pred = softmax.predict(X_val)

        # Calculating the accuracies
        train_accuracy = np.mean(y_train == y_train_pred)
        val_accuracy = np.mean(y_val == y_val_pred)
```

```
        # Storing the results in a dictionary so that we can compare the␣
 ↪results of this version of the classifier with these set of hyper params to␣
 ↪other models.
        results[(lr, reg)] = (train_accuracy, val_accuracy)

        # Checking if this is the best validation accuracy so far and is yes␣
 ↪then assigning to get the best
        if val_accuracy > best_val:
            best_val = val_accuracy
            best_softmax = softmax
            best_learning_rate = lr
            best_reg_strength = reg
print(f"Best Pair of lr and reg is (lr,reg):
 ↪{(best_learning_rate,best_reg_strength)}")

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)
```

```
iteration 0 / 1500: loss 36.316609
iteration 100 / 1500: loss 36.073047
iteration 200 / 1500: loss 35.233692
iteration 300 / 1500: loss 34.820098
iteration 400 / 1500: loss 34.351820
iteration 500 / 1500: loss 34.071754
iteration 600 / 1500: loss 33.334860
iteration 700 / 1500: loss 33.652979
iteration 800 / 1500: loss 33.499578
iteration 900 / 1500: loss 33.097452
iteration 1000 / 1500: loss 32.952140
iteration 1100 / 1500: loss 32.533762
iteration 1200 / 1500: loss 32.347198
iteration 1300 / 1500: loss 32.311144
iteration 1400 / 1500: loss 32.274609
iteration 0 / 1500: loss 1080.482646
iteration 100 / 1500: loss 939.186610
iteration 200 / 1500: loss 816.718685
```

```
iteration 300 / 1500: loss 709.997798
iteration 400 / 1500: loss 617.140955
iteration 500 / 1500: loss 536.602866
iteration 600 / 1500: loss 466.622819
iteration 700 / 1500: loss 405.729725
iteration 800 / 1500: loss 353.035940
iteration 900 / 1500: loss 306.743714
iteration 1000 / 1500: loss 266.856176
iteration 1100 / 1500: loss 232.520823
iteration 1200 / 1500: loss 202.094993
iteration 1300 / 1500: loss 175.969967
iteration 1400 / 1500: loss 153.159684
iteration 0 / 1500: loss 1547.353554
iteration 100 / 1500: loss 1266.715933
iteration 200 / 1500: loss 1036.832993
iteration 300 / 1500: loss 848.975046
iteration 400 / 1500: loss 695.085515
iteration 500 / 1500: loss 569.083774
iteration 600 / 1500: loss 466.013050
iteration 700 / 1500: loss 381.965942
iteration 800 / 1500: loss 312.875949
iteration 900 / 1500: loss 256.552214
iteration 1000 / 1500: loss 210.242373
iteration 1100 / 1500: loss 172.490289
iteration 1200 / 1500: loss 141.576737
iteration 1300 / 1500: loss 116.192699
iteration 1400 / 1500: loss 95.550449
iteration 0 / 1500: loss 35.616087
iteration 100 / 1500: loss 31.383752
iteration 200 / 1500: loss 29.033425
iteration 300 / 1500: loss 26.423921
iteration 400 / 1500: loss 24.649608
iteration 500 / 1500: loss 22.439785
iteration 600 / 1500: loss 21.153695
iteration 700 / 1500: loss 19.240854
iteration 800 / 1500: loss 17.979921
iteration 900 / 1500: loss 16.821137
iteration 1000 / 1500: loss 15.381222
iteration 1100 / 1500: loss 14.446203
iteration 1200 / 1500: loss 13.462747
iteration 1300 / 1500: loss 12.474733
iteration 1400 / 1500: loss 11.595045
iteration 0 / 1500: loss 1080.263290
iteration 100 / 1500: loss 65.776912
iteration 200 / 1500: loss 5.887119
iteration 300 / 1500: loss 2.328118
iteration 400 / 1500: loss 2.093984
iteration 500 / 1500: loss 2.108257
```

```
iteration 600 / 1500: loss 2.106822
iteration 700 / 1500: loss 2.138113
iteration 800 / 1500: loss 2.083934
iteration 900 / 1500: loss 2.119192
iteration 1000 / 1500: loss 2.143298
iteration 1100 / 1500: loss 2.095798
iteration 1200 / 1500: loss 2.134641
iteration 1300 / 1500: loss 2.173169
iteration 1400 / 1500: loss 2.117696
iteration 0 / 1500: loss 1568.050807
iteration 100 / 1500: loss 29.427760
iteration 200 / 1500: loss 2.616158
iteration 300 / 1500: loss 2.150774
iteration 400 / 1500: loss 2.181951
iteration 500 / 1500: loss 2.113549
iteration 600 / 1500: loss 2.105868
iteration 700 / 1500: loss 2.145527
iteration 800 / 1500: loss 2.106802
iteration 900 / 1500: loss 2.168854
iteration 1000 / 1500: loss 2.158514
iteration 1100 / 1500: loss 2.190300
iteration 1200 / 1500: loss 2.117505
iteration 1300 / 1500: loss 2.171777
iteration 1400 / 1500: loss 2.178185
iteration 0 / 1500: loss 36.880181
iteration 100 / 1500: loss 33.041185
iteration 200 / 1500: loss 31.168799
iteration 300 / 1500: loss 29.706165
iteration 400 / 1500: loss 28.254585
iteration 500 / 1500: loss 27.591210
iteration 600 / 1500: loss 26.368788
iteration 700 / 1500: loss 25.080497
iteration 800 / 1500: loss 24.427949
iteration 900 / 1500: loss 23.305841
iteration 1000 / 1500: loss 22.372280
iteration 1100 / 1500: loss 21.657389
iteration 1200 / 1500: loss 20.962266
iteration 1300 / 1500: loss 19.712806
iteration 1400 / 1500: loss 19.160910
iteration 0 / 1500: loss 1079.763731
iteration 100 / 1500: loss 265.590846
iteration 200 / 1500: loss 66.475113
iteration 300 / 1500: loss 17.893089
iteration 400 / 1500: loss 5.999073
iteration 500 / 1500: loss 2.976847
iteration 600 / 1500: loss 2.309058
iteration 700 / 1500: loss 2.157284
iteration 800 / 1500: loss 2.135818
```

```
iteration 900 / 1500: loss 2.106932
iteration 1000 / 1500: loss 2.172374
iteration 1100 / 1500: loss 2.103289
iteration 1200 / 1500: loss 2.071648
iteration 1300 / 1500: loss 2.148428
iteration 1400 / 1500: loss 2.067904
iteration 0 / 1500: loss 1553.103107
iteration 100 / 1500: loss 208.938649
iteration 200 / 1500: loss 29.790851
iteration 300 / 1500: loss 5.793372
iteration 400 / 1500: loss 2.670339
iteration 500 / 1500: loss 2.198073
iteration 600 / 1500: loss 2.169863
iteration 700 / 1500: loss 2.103527
iteration 800 / 1500: loss 2.133435
iteration 900 / 1500: loss 2.138942
iteration 1000 / 1500: loss 2.143258
iteration 1100 / 1500: loss 2.072654
iteration 1200 / 1500: loss 2.116633
iteration 1300 / 1500: loss 2.146572
iteration 1400 / 1500: loss 2.135522
iteration 0 / 1500: loss 36.008717
iteration 100 / 1500: loss 15.656978
iteration 200 / 1500: loss 7.877814
iteration 300 / 1500: loss 4.625329
iteration 400 / 1500: loss 3.048986
iteration 500 / 1500: loss 2.302882
iteration 600 / 1500: loss 2.067097
iteration 700 / 1500: loss 2.001171
iteration 800 / 1500: loss 1.866237
iteration 900 / 1500: loss 1.927080
iteration 1000 / 1500: loss 1.923320
iteration 1100 / 1500: loss 1.731046
iteration 1200 / 1500: loss 1.861990
iteration 1300 / 1500: loss 1.872138
iteration 1400 / 1500: loss 1.976305
iteration 0 / 1500: loss 1087.402021
iteration 100 / 1500: loss 2.188802
iteration 200 / 1500: loss 2.200831
iteration 300 / 1500: loss 2.166470
iteration 400 / 1500: loss 2.159856
iteration 500 / 1500: loss 2.086656
iteration 600 / 1500: loss 2.123634
iteration 700 / 1500: loss 2.151914
iteration 800 / 1500: loss 2.173347
iteration 900 / 1500: loss 2.193971
iteration 1000 / 1500: loss 2.141113
iteration 1100 / 1500: loss 2.151920
```

```
iteration 1200 / 1500: loss 2.179938
iteration 1300 / 1500: loss 2.211356
iteration 1400 / 1500: loss 2.135848
iteration 0 / 1500: loss 1520.319871
iteration 100 / 1500: loss 2.168555
iteration 200 / 1500: loss 2.222904
iteration 300 / 1500: loss 2.150369
iteration 400 / 1500: loss 2.171968
iteration 500 / 1500: loss 2.138092
iteration 600 / 1500: loss 2.222273
iteration 700 / 1500: loss 2.215237
iteration 800 / 1500: loss 2.147038
iteration 900 / 1500: loss 2.145841
iteration 1000 / 1500: loss 2.151871
iteration 1100 / 1500: loss 2.126298
iteration 1200 / 1500: loss 2.142999
iteration 1300 / 1500: loss 2.176929
iteration 1400 / 1500: loss 2.195919
iteration 0 / 1500: loss 36.588127
iteration 100 / 1500: loss 9.032052
iteration 200 / 1500: loss 13.219118
iteration 300 / 1500: loss 11.224696
iteration 400 / 1500: loss 7.562027
iteration 500 / 1500: loss 12.698649
iteration 600 / 1500: loss 14.532674
iteration 700 / 1500: loss 23.620417
iteration 800 / 1500: loss 15.020990
iteration 900 / 1500: loss 12.449357
iteration 1000 / 1500: loss 14.453884
iteration 1100 / 1500: loss 8.520174
iteration 1200 / 1500: loss 11.690602
iteration 1300 / 1500: loss 6.674356
iteration 1400 / 1500: loss 8.823973
iteration 0 / 1500: loss 1089.339474

/content/drive/My
Drive/cs231n/assignments/assignment1/cs231n/classifiers/softmax.py:128:
RuntimeWarning: divide by zero encountered in log
  loss = np.sum(-np.log(softmax_probs [np.arange(no_train), y]) )

iteration 100 / 1500: loss inf
iteration 200 / 1500: loss inf
iteration 300 / 1500: loss inf
iteration 400 / 1500: loss inf
iteration 500 / 1500: loss inf
iteration 600 / 1500: loss inf
iteration 700 / 1500: loss inf
iteration 800 / 1500: loss inf
iteration 900 / 1500: loss inf
```

```
iteration 1000 / 1500: loss inf
iteration 1100 / 1500: loss inf
iteration 1200 / 1500: loss inf
iteration 1300 / 1500: loss inf
iteration 1400 / 1500: loss inf
iteration 0 / 1500: loss 1515.635159
iteration 100 / 1500: loss inf
iteration 200 / 1500: loss inf
iteration 300 / 1500: loss inf
iteration 400 / 1500: loss inf
iteration 500 / 1500: loss inf

/content/drive/My
Drive/cs231n/assignments/assignment1/cs231n/classifiers/softmax.py:139:
RuntimeWarning: overflow encountered in double_scalars
  loss += reg * np.sum(W * W)
/usr/local/lib/python3.10/dist-packages/numpy/core/fromnumeric.py:86:
RuntimeWarning: overflow encountered in reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
/content/drive/My
Drive/cs231n/assignments/assignment1/cs231n/classifiers/softmax.py:139:
RuntimeWarning: overflow encountered in multiply
  loss += reg * np.sum(W * W)

iteration 600 / 1500: loss inf
iteration 700 / 1500: loss inf
iteration 800 / 1500: loss inf
iteration 900 / 1500: loss inf
iteration 1000 / 1500: loss inf

/content/drive/My
Drive/cs231n/assignments/assignment1/cs231n/classifiers/softmax.py:141:
RuntimeWarning: overflow encountered in multiply
  dW += reg * 2 * W

iteration 1100 / 1500: loss nan
iteration 1200 / 1500: loss nan
iteration 1300 / 1500: loss nan
iteration 1400 / 1500: loss nan
Best Pair of lr and reg is (lr,reg):(2e-06, 1000.0)
lr 1.000000e-08 reg 1.000000e+03 train accuracy: 0.173980 val accuracy: 0.177000
lr 1.000000e-08 reg 3.500000e+04 train accuracy: 0.203408 val accuracy: 0.227000
lr 1.000000e-08 reg 5.000000e+04 train accuracy: 0.208265 val accuracy: 0.200000
lr 1.000000e-07 reg 1.000000e+03 train accuracy: 0.269939 val accuracy: 0.295000
lr 1.000000e-07 reg 3.500000e+04 train accuracy: 0.325163 val accuracy: 0.340000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.300980 val accuracy: 0.323000
lr 2.000000e-07 reg 1.000000e+03 train accuracy: 0.313776 val accuracy: 0.305000
lr 2.000000e-07 reg 3.500000e+04 train accuracy: 0.315714 val accuracy: 0.335000
lr 2.000000e-07 reg 5.000000e+04 train accuracy: 0.301612 val accuracy: 0.317000
lr 2.000000e-06 reg 1.000000e+03 train accuracy: 0.396796 val accuracy: 0.395000
```

```
lr 2.000000e-06 reg 3.500000e+04 train accuracy: 0.271306 val accuracy: 0.269000
lr 2.000000e-06 reg 5.000000e+04 train accuracy: 0.258612 val accuracy: 0.260000
lr 3.000000e-05 reg 1.000000e+03 train accuracy: 0.219020 val accuracy: 0.230000
lr 3.000000e-05 reg 3.500000e+04 train accuracy: 0.079286 val accuracy: 0.094000
lr 3.000000e-05 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
best validation accuracy achieved during cross-validation: 0.395000
```

[ ]:
```
# 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: 0.395000
```

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.

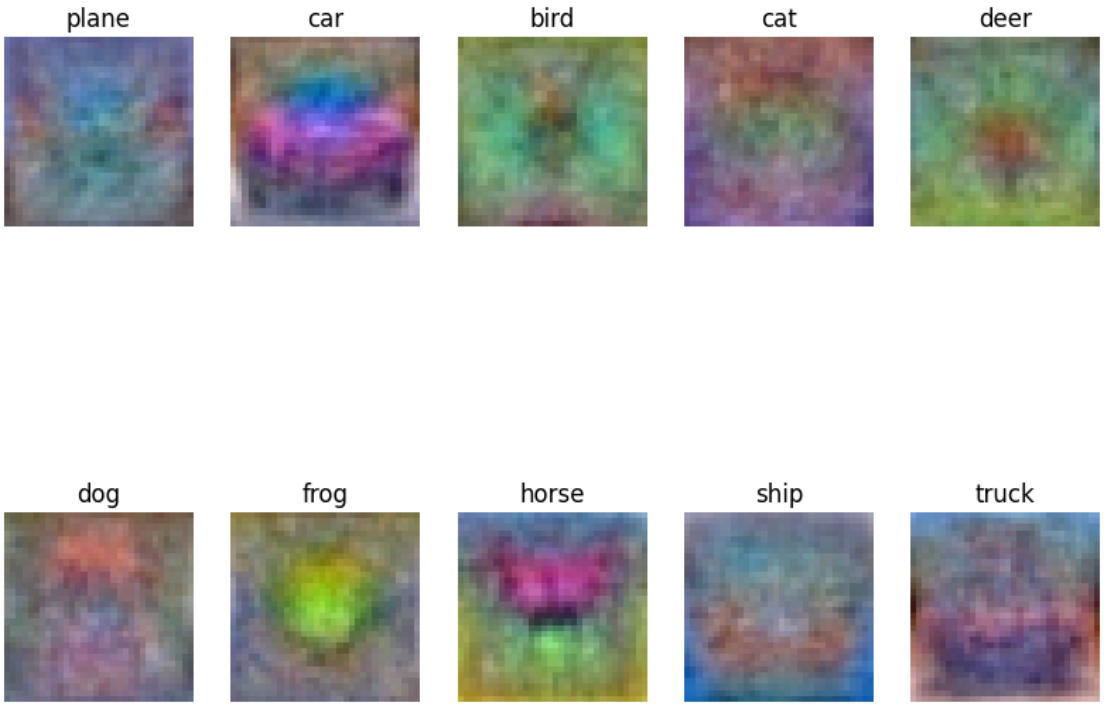
YourAnswer : True.

YourExplanation : In the case of the SVM loss, loss for each data point is determined by how close it is to the correct classification margin. If a new data point is added that is correctly classified with a margin greater than 1 (in linear SVM case), it would not contribute to the loss at all because the hinge loss for that data point would be zero. Therefore, adding such a data point would leave the overall training loss unchanged.

In contrast, the Softmax classifier computes the loss for each data point based on the probabilities assigned to all classes. The loss depends not only on the correctness of the classification but also on the confidence of the classifier's prediction. If a new data point is added and it has a different ground truth label, it would contribute to the Softmax loss, and this contribution would likely change the overall training loss. So, adding a new data point could potentially change the overall Softmax classifier loss.

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



[ ]:

two_layer_net

October 6, 2023

[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 = 'cs231n/assignments/assignment1/'
assert FOLDERNAME is not None, "[!] Enter the foldername."

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

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

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

1 Fully-Connected Neural Nets

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

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

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

  return out, cache
```

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

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

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

  return dx, dw
```

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

[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:

The issue of zero/close to zero gradient flow during backpropagation can lead to vanishing gradients, which can slow down or even hinder the training of neural networks.

Sigmoid:

Potential Issue: The sigmoid activation function can suffer from vanishing gradient problems, especially when its output is close to 0 or 1. This is because the gradient of the sigmoid function becomes very small in these regions.

Types of Input: For sigmoid, inputs that are significantly far from 0 can lead to this issue. If the input to the sigmoid function is very large (positive or negative), the sigmoid output will be close

to 0 or 1, and the gradient will be close to 0.

ReLU (Rectified Linear Unit):

Potential Issue: The ReLU activation function can suffer from a different problem called "dying ReLU" when the input is negative. In this case, the gradient is exactly 0, and no weight updates occur during backpropagation for those neurons.

Types of Input: For ReLU, inputs that result in negative values will lead to this issue. Any input less than or equal to 0 will produce a gradient of 0, causing the neuron to "die."

Leaky ReLU:

Potential Issue: Leaky ReLU was introduced to address the "dying ReLU" problem of the standard ReLU. It has a small, non-zero gradient for negative inputs, which helps prevent neurons from becoming completely inactive during training.

Types of Input: Leaky ReLU is less prone to vanishing gradients compared to the standard ReLU. It is less sensitive to the types of inputs that lead to the "dying ReLU" problem. In summary, the activation functions most prone to the problem of zero or close to zero gradient flow during backpropagation are:

Sigmoid, when the input is far from 0.

ReLU, when the input is negative.

Leaky ReLU is introduced to mitigate the issues associated with the standard ReLU and has a less severe problem with vanishing gradients, especially for negative inputs.

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:  2.3025458445007376
dx error:  8.234144091578429e-09
```

8 Two-layer network

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

[11]:
```
np.random.seed(231)
N, D, H, C = 3, 5, 50, 7
X = np.random.randn(N, D)
y = np.random.randint(C, size=N)

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

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

print('Testing test-time forward pass ... ')
model.params['W1'] = np.linspace(-0.7, 0.3, num=D*H).reshape(D, H)
model.params['b1'] = np.linspace(-0.1, 0.9, num=H)
model.params['W2'] = np.linspace(-0.3, 0.4, num=H*C).reshape(H, C)
model.params['b2'] = np.linspace(-0.9, 0.1, num=C)
X = np.linspace(-5.5, 4.5, num=N*D).reshape(D, N).T
scores = model.loss(X)
correct_scores = np.asarray(
  [[11.53165108,  12.2917344,   13.05181771,  13.81190102,  14.57198434, 15.
 ↪33206765,  16.09215096],
   [12.05769098,  12.74614105,  13.43459113,  14.1230412,   14.81149128, 15.
 ↪49994135,  16.18839143],
   [12.58373087,  13.20054771,  13.81736455,  14.43418138,  15.05099822, 15.
 ↪66781506,  16.2846319 ]])
scores_diff = np.abs(scores - correct_scores).sum()
assert scores_diff < 1e-6, 'Problem with test-time forward pass'
```

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

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

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

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

```
Testing initialization …
Testing test-time forward pass …
Testing training loss (no regularization)
Running numeric gradient check with reg =  0.0
W1 relative error: 1.83e-08
W2 relative error: 3.20e-10
b1 relative error: 9.83e-09
b2 relative error: 4.33e-10
Running numeric gradient check with reg =  0.7
W1 relative error: 2.53e-07
W2 relative error: 2.85e-08
b1 relative error: 1.56e-08
b2 relative error: 9.09e-10
```

9 Solver

Open the file `cs231n/solver.py` and read through it to familiarize yourself with the API. 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.

[13]:
```
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
model = TwoLayerNet(input_size, hidden_size, num_classes)
solver = None
```

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

solver = Solver(model, data, optim_config={'learning_rate': 3e-4,},␣
 ↪print_every=100)
solver.train()

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: 0.097000; val_acc: 0.081000
(Iteration 101 / 4900) loss: 2.025064
(Iteration 201 / 4900) loss: 1.936724
(Iteration 301 / 4900) loss: 1.766423
(Iteration 401 / 4900) loss: 1.769296
(Epoch 1 / 10) train acc: 0.405000; val_acc: 0.388000
(Iteration 501 / 4900) loss: 1.649664
(Iteration 601 / 4900) loss: 1.715659
(Iteration 701 / 4900) loss: 1.628100
(Iteration 801 / 4900) loss: 1.553015
(Iteration 901 / 4900) loss: 1.688175
(Epoch 2 / 10) train acc: 0.441000; val_acc: 0.464000
(Iteration 1001 / 4900) loss: 1.642180
(Iteration 1101 / 4900) loss: 1.589000
(Iteration 1201 / 4900) loss: 1.525500
(Iteration 1301 / 4900) loss: 1.617081
(Iteration 1401 / 4900) loss: 1.437845
(Epoch 3 / 10) train acc: 0.481000; val_acc: 0.456000
(Iteration 1501 / 4900) loss: 1.456039
(Iteration 1601 / 4900) loss: 1.577966
(Iteration 1701 / 4900) loss: 1.417849
(Iteration 1801 / 4900) loss: 1.468534
(Iteration 1901 / 4900) loss: 1.360427
(Epoch 4 / 10) train acc: 0.515000; val_acc: 0.472000
(Iteration 2001 / 4900) loss: 1.351543
(Iteration 2101 / 4900) loss: 1.444493
(Iteration 2201 / 4900) loss: 1.577119
(Iteration 2301 / 4900) loss: 1.621160
```

```
(Iteration 2401 / 4900) loss: 1.457269
(Epoch 5 / 10) train acc: 0.489000; val_acc: 0.486000
(Iteration 2501 / 4900) loss: 1.434752
(Iteration 2601 / 4900) loss: 1.314916
(Iteration 2701 / 4900) loss: 1.445745
(Iteration 2801 / 4900) loss: 1.377772
(Iteration 2901 / 4900) loss: 1.383597
(Epoch 6 / 10) train acc: 0.513000; val_acc: 0.481000
(Iteration 3001 / 4900) loss: 1.362293
(Iteration 3101 / 4900) loss: 1.274884
(Iteration 3201 / 4900) loss: 1.539416
(Iteration 3301 / 4900) loss: 1.391934
(Iteration 3401 / 4900) loss: 1.513996
(Epoch 7 / 10) train acc: 0.527000; val_acc: 0.472000
(Iteration 3501 / 4900) loss: 1.241678
(Iteration 3601 / 4900) loss: 1.262072
(Iteration 3701 / 4900) loss: 1.387504
(Iteration 3801 / 4900) loss: 1.269591
(Iteration 3901 / 4900) loss: 1.410196
(Epoch 8 / 10) train acc: 0.502000; val_acc: 0.512000
(Iteration 4001 / 4900) loss: 1.343121
(Iteration 4101 / 4900) loss: 1.338758
(Iteration 4201 / 4900) loss: 1.197816
(Iteration 4301 / 4900) loss: 1.357614
(Iteration 4401 / 4900) loss: 1.209649
(Epoch 9 / 10) train acc: 0.541000; val_acc: 0.505000
(Iteration 4501 / 4900) loss: 1.247041
(Iteration 4601 / 4900) loss: 1.402323
(Iteration 4701 / 4900) loss: 1.152403
(Iteration 4801 / 4900) loss: 1.368161
(Epoch 10 / 10) train acc: 0.540000; val_acc: 0.518000
```

10 Debug the training

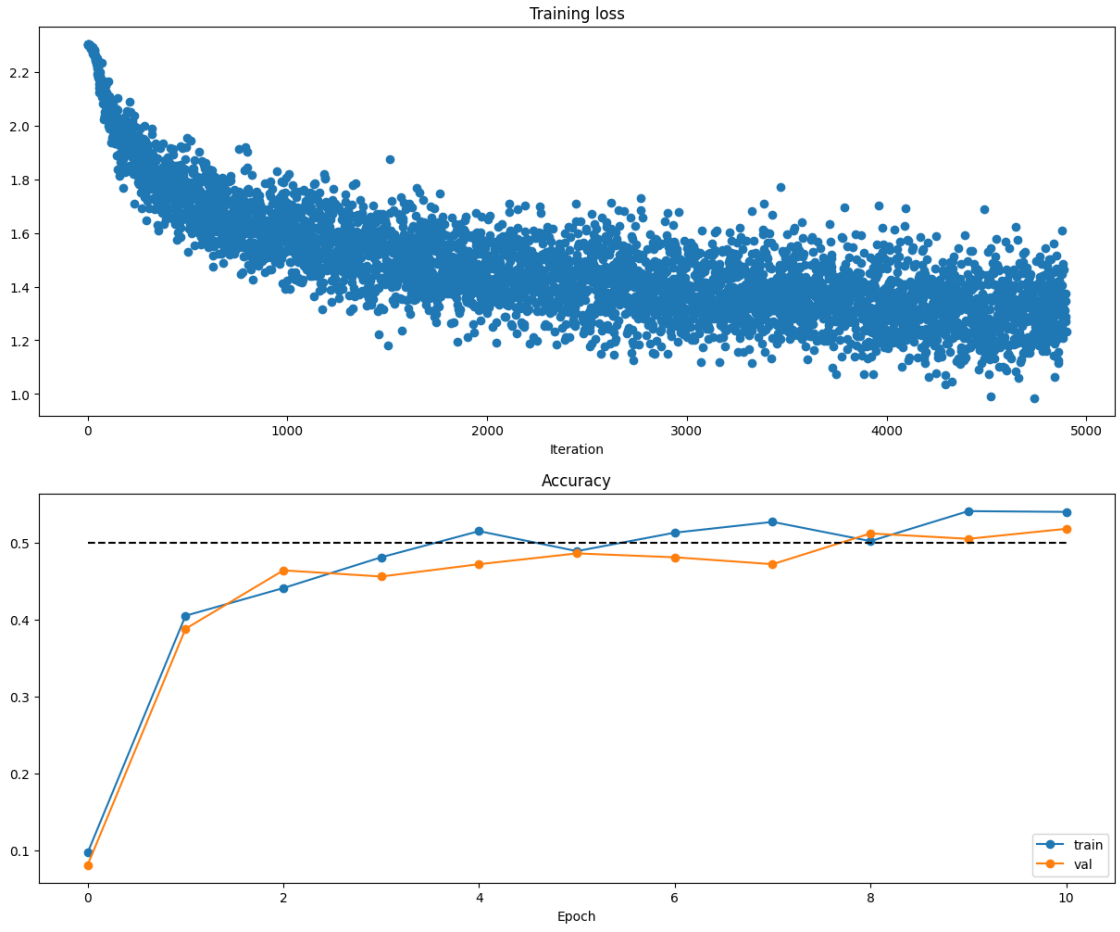
With the default parameters we provided above, you should get a validation accuracy of about 0.36 on the validation set. This isn't very good.

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.

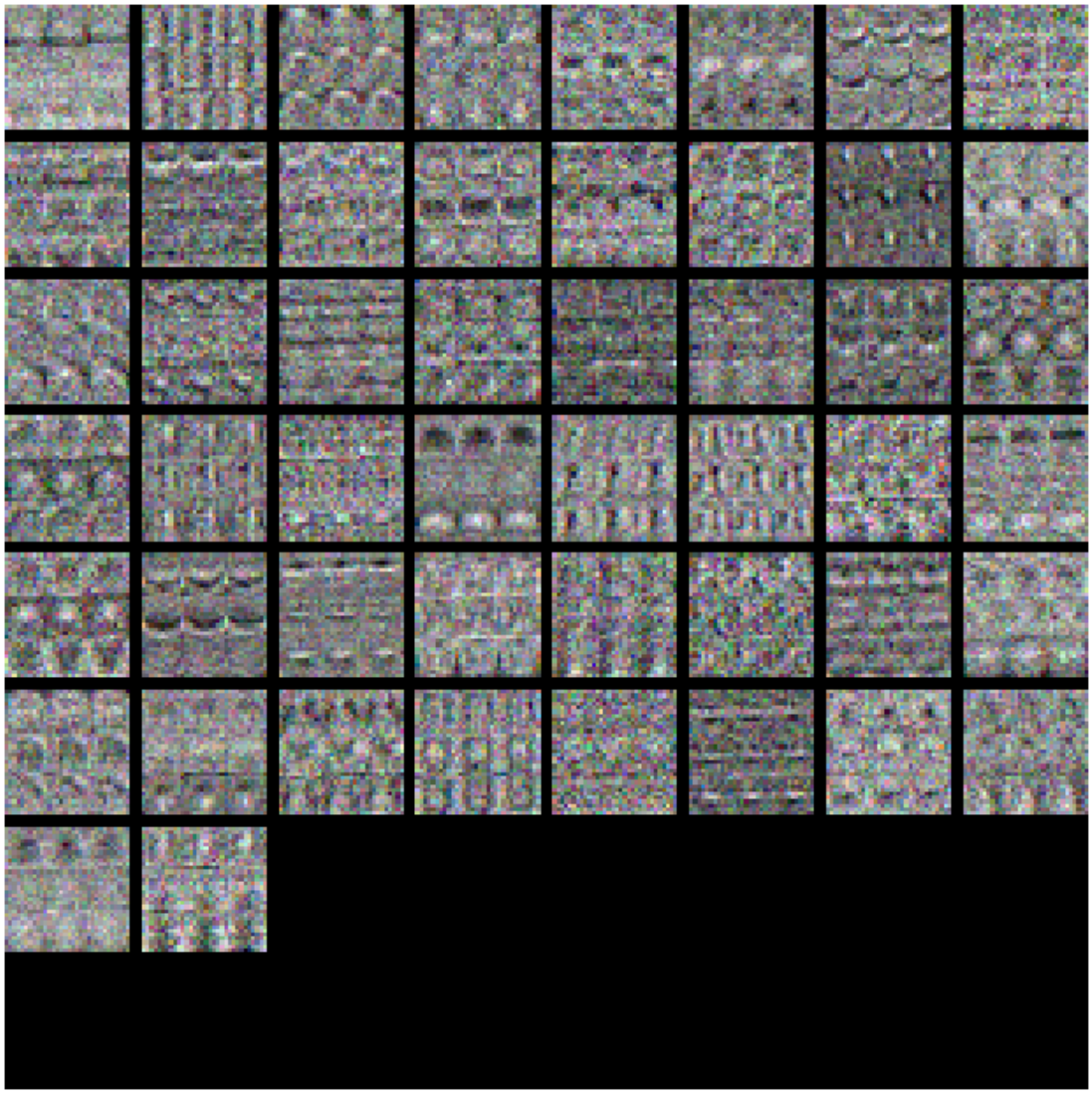
[14]:
```
# Run this cell to visualize training loss and train / val accuracy

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()
```



[15]:
```
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.).

[19]:
```
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)*****

from itertools import product
```

```
# Initialize the best validation accuracy to track the best model found
best_validation_accuracy = -1

# Define the input size, number of classes, hidden layer size, number of␣
 ↪epochs, and batch size
input_size = 32 * 32 * 3
num_classes = 10
hidden_size = 50
num_epochs = 10
batch_size = 256

# Define a dictionary of hyperparameters to search
hyperparameter_options = {
    "learning_rates": [3e-3, 2e-3, 1.5e-3],
    "lr_decay_rates": [0.25, 0.45]
}

# Generate all possible combinations of hyperparameters
param_combinations = list(product(hyperparameter_options["learning_rates"],␣
 ↪hyperparameter_options["lr_decay_rates"]))

# Iterate through each combination of hyperparameters
for learning_rate, lr_decay in param_combinations:
    print("Hyperparameters - Learning Rate: {}, Learning Rate Decay: {}".
 ↪format(learning_rate, lr_decay))

    # Create a new neural network model
    neural_network_model = TwoLayerNet(input_size, hidden_size, num_classes)

    # Create a solver with the specified hyperparameters
    solver = Solver(neural_network_model, data, update_rule='sgd',
                    optim_config={
                        'learning_rate': learning_rate
                    },
                    lr_decay=lr_decay,
                    num_epochs=num_epochs, batch_size=batch_size,
                    print_every=100)

    # Train the model
    solver.train()

    # Check if the model's validation accuracy is better than the current best␣
 ↪accuracy
    if solver.val_acc_history[-1] > best_validation_accuracy:

        best_hyperparameters = (learning_rate, lr_decay)
```

```
        best_trained_model = neural_network_model
        best_validation_accuracy = solver.val_acc_history[-1]

# The model with the highest validation accuracy whose optimal hyperparameters␣
 ↪have been found, is the best_trained_model.
print('Best Validation Accuracy:', best_validation_accuracy)
print('Best Hyperparameters:', best_hyperparameters)

pass

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

```
Hyperparameters - Learning Rate: 0.003, Learning Rate Decay: 0.25
(Iteration 1 / 1910) loss: 2.302237
(Epoch 0 / 10) train acc: 0.160000; val_acc: 0.156000
(Iteration 101 / 1910) loss: 1.859524
(Epoch 1 / 10) train acc: 0.399000; val_acc: 0.420000
(Iteration 201 / 1910) loss: 1.506938
(Iteration 301 / 1910) loss: 1.459287
(Epoch 2 / 10) train acc: 0.464000; val_acc: 0.466000
(Iteration 401 / 1910) loss: 1.481963
(Iteration 501 / 1910) loss: 1.368747
(Epoch 3 / 10) train acc: 0.495000; val_acc: 0.476000
(Iteration 601 / 1910) loss: 1.595498
(Iteration 701 / 1910) loss: 1.514998
(Epoch 4 / 10) train acc: 0.504000; val_acc: 0.476000
(Iteration 801 / 1910) loss: 1.491838
(Iteration 901 / 1910) loss: 1.487769
(Epoch 5 / 10) train acc: 0.495000; val_acc: 0.474000
(Iteration 1001 / 1910) loss: 1.560720
(Iteration 1101 / 1910) loss: 1.382429
(Epoch 6 / 10) train acc: 0.490000; val_acc: 0.477000
(Iteration 1201 / 1910) loss: 1.428664
(Iteration 1301 / 1910) loss: 1.485529
(Epoch 7 / 10) train acc: 0.484000; val_acc: 0.478000
(Iteration 1401 / 1910) loss: 1.498987
(Iteration 1501 / 1910) loss: 1.521307
(Epoch 8 / 10) train acc: 0.513000; val_acc: 0.478000
(Iteration 1601 / 1910) loss: 1.372475
(Iteration 1701 / 1910) loss: 1.522900
(Epoch 9 / 10) train acc: 0.485000; val_acc: 0.478000
(Iteration 1801 / 1910) loss: 1.464076
(Iteration 1901 / 1910) loss: 1.473963
(Epoch 10 / 10) train acc: 0.479000; val_acc: 0.478000
```

```
Hyperparameters - Learning Rate: 0.003, Learning Rate Decay: 0.45
(Iteration 1 / 1910) loss: 2.300891
(Epoch 0 / 10) train acc: 0.179000; val_acc: 0.184000
(Iteration 101 / 1910) loss: 1.713781
(Epoch 1 / 10) train acc: 0.399000; val_acc: 0.412000
(Iteration 201 / 1910) loss: 1.530697
(Iteration 301 / 1910) loss: 1.507250
(Epoch 2 / 10) train acc: 0.512000; val_acc: 0.483000
(Iteration 401 / 1910) loss: 1.370563
(Iteration 501 / 1910) loss: 1.442534
(Epoch 3 / 10) train acc: 0.490000; val_acc: 0.478000
(Iteration 601 / 1910) loss: 1.335106
(Iteration 701 / 1910) loss: 1.385362
(Epoch 4 / 10) train acc: 0.503000; val_acc: 0.486000
(Iteration 801 / 1910) loss: 1.416958
(Iteration 901 / 1910) loss: 1.409148
(Epoch 5 / 10) train acc: 0.550000; val_acc: 0.500000
(Iteration 1001 / 1910) loss: 1.372645
(Iteration 1101 / 1910) loss: 1.457300
(Epoch 6 / 10) train acc: 0.504000; val_acc: 0.490000
(Iteration 1201 / 1910) loss: 1.517041
(Iteration 1301 / 1910) loss: 1.489971
(Epoch 7 / 10) train acc: 0.557000; val_acc: 0.486000
(Iteration 1401 / 1910) loss: 1.428651
(Iteration 1501 / 1910) loss: 1.510028
(Epoch 8 / 10) train acc: 0.523000; val_acc: 0.493000
(Iteration 1601 / 1910) loss: 1.406976
(Iteration 1701 / 1910) loss: 1.397870
(Epoch 9 / 10) train acc: 0.515000; val_acc: 0.488000
(Iteration 1801 / 1910) loss: 1.401177
(Iteration 1901 / 1910) loss: 1.351422
(Epoch 10 / 10) train acc: 0.520000; val_acc: 0.488000
Hyperparameters - Learning Rate: 0.002, Learning Rate Decay: 0.25
(Iteration 1 / 1910) loss: 2.300929
(Epoch 0 / 10) train acc: 0.150000; val_acc: 0.158000
(Iteration 101 / 1910) loss: 1.713604
(Epoch 1 / 10) train acc: 0.430000; val_acc: 0.420000
(Iteration 201 / 1910) loss: 1.644549
(Iteration 301 / 1910) loss: 1.533275
(Epoch 2 / 10) train acc: 0.457000; val_acc: 0.463000
(Iteration 401 / 1910) loss: 1.546245
(Iteration 501 / 1910) loss: 1.575114
(Epoch 3 / 10) train acc: 0.466000; val_acc: 0.466000
(Iteration 601 / 1910) loss: 1.528638
(Iteration 701 / 1910) loss: 1.565317
(Epoch 4 / 10) train acc: 0.482000; val_acc: 0.464000
(Iteration 801 / 1910) loss: 1.415264
(Iteration 901 / 1910) loss: 1.446481
```

```
(Epoch 5 / 10) train acc: 0.483000; val_acc: 0.464000
(Iteration 1001 / 1910) loss: 1.508947
(Iteration 1101 / 1910) loss: 1.529204
(Epoch 6 / 10) train acc: 0.474000; val_acc: 0.465000
(Iteration 1201 / 1910) loss: 1.356346
(Iteration 1301 / 1910) loss: 1.473579
(Epoch 7 / 10) train acc: 0.499000; val_acc: 0.464000
(Iteration 1401 / 1910) loss: 1.488212
(Iteration 1501 / 1910) loss: 1.507403
(Epoch 8 / 10) train acc: 0.505000; val_acc: 0.464000
(Iteration 1601 / 1910) loss: 1.428836
(Iteration 1701 / 1910) loss: 1.499964
(Epoch 9 / 10) train acc: 0.470000; val_acc: 0.464000
(Iteration 1801 / 1910) loss: 1.398749
(Iteration 1901 / 1910) loss: 1.501928
(Epoch 10 / 10) train acc: 0.495000; val_acc: 0.464000
Hyperparameters - Learning Rate: 0.002, Learning Rate Decay: 0.45
(Iteration 1 / 1910) loss: 2.303462
(Epoch 0 / 10) train acc: 0.128000; val_acc: 0.121000
(Iteration 101 / 1910) loss: 1.727003
(Epoch 1 / 10) train acc: 0.446000; val_acc: 0.429000
(Iteration 201 / 1910) loss: 1.651835
(Iteration 301 / 1910) loss: 1.490864
(Epoch 2 / 10) train acc: 0.464000; val_acc: 0.446000
(Iteration 401 / 1910) loss: 1.478829
(Iteration 501 / 1910) loss: 1.460230
(Epoch 3 / 10) train acc: 0.483000; val_acc: 0.453000
(Iteration 601 / 1910) loss: 1.536655
(Iteration 701 / 1910) loss: 1.345636
(Epoch 4 / 10) train acc: 0.495000; val_acc: 0.465000
(Iteration 801 / 1910) loss: 1.456542
(Iteration 901 / 1910) loss: 1.431487
(Epoch 5 / 10) train acc: 0.511000; val_acc: 0.474000
(Iteration 1001 / 1910) loss: 1.414174
(Iteration 1101 / 1910) loss: 1.415306
(Epoch 6 / 10) train acc: 0.499000; val_acc: 0.468000
(Iteration 1201 / 1910) loss: 1.378842
(Iteration 1301 / 1910) loss: 1.390959
(Epoch 7 / 10) train acc: 0.494000; val_acc: 0.469000
(Iteration 1401 / 1910) loss: 1.559348
(Iteration 1501 / 1910) loss: 1.455188
(Epoch 8 / 10) train acc: 0.515000; val_acc: 0.469000
(Iteration 1601 / 1910) loss: 1.453469
(Iteration 1701 / 1910) loss: 1.519589
(Epoch 9 / 10) train acc: 0.507000; val_acc: 0.469000
(Iteration 1801 / 1910) loss: 1.526029
(Iteration 1901 / 1910) loss: 1.456357
(Epoch 10 / 10) train acc: 0.503000; val_acc: 0.469000
```

```
Hyperparameters - Learning Rate: 0.0015, Learning Rate Decay: 0.25
(Iteration 1 / 1910) loss: 2.303873
(Epoch 0 / 10) train acc: 0.134000; val_acc: 0.155000
(Iteration 101 / 1910) loss: 1.792287
(Epoch 1 / 10) train acc: 0.416000; val_acc: 0.419000
(Iteration 201 / 1910) loss: 1.772274
(Iteration 301 / 1910) loss: 1.571606
(Epoch 2 / 10) train acc: 0.444000; val_acc: 0.449000
(Iteration 401 / 1910) loss: 1.540125
(Iteration 501 / 1910) loss: 1.625230
(Epoch 3 / 10) train acc: 0.481000; val_acc: 0.452000
(Iteration 601 / 1910) loss: 1.665924
(Iteration 701 / 1910) loss: 1.541718
(Epoch 4 / 10) train acc: 0.457000; val_acc: 0.450000
(Iteration 801 / 1910) loss: 1.441868
(Iteration 901 / 1910) loss: 1.544306
(Epoch 5 / 10) train acc: 0.453000; val_acc: 0.452000
(Iteration 1001 / 1910) loss: 1.488196
(Iteration 1101 / 1910) loss: 1.447303
(Epoch 6 / 10) train acc: 0.464000; val_acc: 0.453000
(Iteration 1201 / 1910) loss: 1.596483
(Iteration 1301 / 1910) loss: 1.591564
(Epoch 7 / 10) train acc: 0.461000; val_acc: 0.453000
(Iteration 1401 / 1910) loss: 1.480700
(Iteration 1501 / 1910) loss: 1.561924
(Epoch 8 / 10) train acc: 0.470000; val_acc: 0.453000
(Iteration 1601 / 1910) loss: 1.495029
(Iteration 1701 / 1910) loss: 1.617942
(Epoch 9 / 10) train acc: 0.441000; val_acc: 0.453000
(Iteration 1801 / 1910) loss: 1.512016
(Iteration 1901 / 1910) loss: 1.595935
(Epoch 10 / 10) train acc: 0.450000; val_acc: 0.453000
Hyperparameters - Learning Rate: 0.0015, Learning Rate Decay: 0.45
(Iteration 1 / 1910) loss: 2.303613
(Epoch 0 / 10) train acc: 0.128000; val_acc: 0.131000
(Iteration 101 / 1910) loss: 1.698786
(Epoch 1 / 10) train acc: 0.427000; val_acc: 0.436000
(Iteration 201 / 1910) loss: 1.622511
(Iteration 301 / 1910) loss: 1.564708
(Epoch 2 / 10) train acc: 0.440000; val_acc: 0.460000
(Iteration 401 / 1910) loss: 1.391405
(Iteration 501 / 1910) loss: 1.487227
(Epoch 3 / 10) train acc: 0.467000; val_acc: 0.467000
(Iteration 601 / 1910) loss: 1.446302
(Iteration 701 / 1910) loss: 1.441114
(Epoch 4 / 10) train acc: 0.479000; val_acc: 0.472000
(Iteration 801 / 1910) loss: 1.578016
(Iteration 901 / 1910) loss: 1.375056
```

```
(Epoch 5 / 10) train acc: 0.490000; val_acc: 0.485000
(Iteration 1001 / 1910) loss: 1.511965
(Iteration 1101 / 1910) loss: 1.490823
(Epoch 6 / 10) train acc: 0.482000; val_acc: 0.479000
(Iteration 1201 / 1910) loss: 1.459326
(Iteration 1301 / 1910) loss: 1.379629
(Epoch 7 / 10) train acc: 0.500000; val_acc: 0.481000
(Iteration 1401 / 1910) loss: 1.411075
(Iteration 1501 / 1910) loss: 1.395615
(Epoch 8 / 10) train acc: 0.471000; val_acc: 0.478000
(Iteration 1601 / 1910) loss: 1.384897
(Iteration 1701 / 1910) loss: 1.442849
(Epoch 9 / 10) train acc: 0.475000; val_acc: 0.478000
(Iteration 1801 / 1910) loss: 1.482059
(Iteration 1901 / 1910) loss: 1.480559
(Epoch 10 / 10) train acc: 0.510000; val_acc: 0.478000
Best Validation Accuracy: 0.488
Best Hyperparameters: (0.003, 0.45)
```

[21]:
```
print(best_hyperparameters)
```

```
(0.003, 0.45)
```

[22]:
```
model = best_trained_model
# Get the best hyperparameters
lr, dec = best_hyperparameters

# Train the best model with the best hyperparameters that were chosen :
solver = Solver(model, data,
                update_rule='sgd',
                optim_config={
                    'learning_rate': lr,
                },
                lr_decay = dec,
                num_epochs=20, batch_size=256,
                print_every=100)
solver.train()
```

```
(Iteration 1 / 3820) loss: 1.298325
(Epoch 0 / 20) train acc: 0.446000; val_acc: 0.436000
(Iteration 101 / 3820) loss: 1.708519
(Epoch 1 / 20) train acc: 0.439000; val_acc: 0.442000
(Iteration 201 / 3820) loss: 1.450442
(Iteration 301 / 3820) loss: 1.368832
(Epoch 2 / 20) train acc: 0.533000; val_acc: 0.487000
(Iteration 401 / 3820) loss: 1.365163
(Iteration 501 / 3820) loss: 1.252313
(Epoch 3 / 20) train acc: 0.523000; val_acc: 0.501000
```

```
(Iteration 601 / 3820) loss: 1.287483
(Iteration 701 / 3820) loss: 1.313354
(Epoch 4 / 20) train acc: 0.577000; val_acc: 0.508000
(Iteration 801 / 3820) loss: 1.382147
(Iteration 901 / 3820) loss: 1.359889
(Epoch 5 / 20) train acc: 0.545000; val_acc: 0.508000
(Iteration 1001 / 3820) loss: 1.430168
(Iteration 1101 / 3820) loss: 1.100564
(Epoch 6 / 20) train acc: 0.566000; val_acc: 0.512000
(Iteration 1201 / 3820) loss: 1.289469
(Iteration 1301 / 3820) loss: 1.250483
(Epoch 7 / 20) train acc: 0.577000; val_acc: 0.505000
(Iteration 1401 / 3820) loss: 1.212393
(Iteration 1501 / 3820) loss: 1.151840
(Epoch 8 / 20) train acc: 0.536000; val_acc: 0.511000
(Iteration 1601 / 3820) loss: 1.336639
(Iteration 1701 / 3820) loss: 1.310572
(Epoch 9 / 20) train acc: 0.524000; val_acc: 0.508000
(Iteration 1801 / 3820) loss: 1.247954
(Iteration 1901 / 3820) loss: 1.301608
(Epoch 10 / 20) train acc: 0.546000; val_acc: 0.507000
(Iteration 2001 / 3820) loss: 1.286673
(Iteration 2101 / 3820) loss: 1.380962
(Epoch 11 / 20) train acc: 0.547000; val_acc: 0.507000
(Iteration 2201 / 3820) loss: 1.253025
(Epoch 12 / 20) train acc: 0.573000; val_acc: 0.507000
(Iteration 2301 / 3820) loss: 1.246743
(Iteration 2401 / 3820) loss: 1.270031
(Epoch 13 / 20) train acc: 0.546000; val_acc: 0.507000
(Iteration 2501 / 3820) loss: 1.358104
(Iteration 2601 / 3820) loss: 1.215398
(Epoch 14 / 20) train acc: 0.549000; val_acc: 0.507000
(Iteration 2701 / 3820) loss: 1.297378
(Iteration 2801 / 3820) loss: 1.245008
(Epoch 15 / 20) train acc: 0.530000; val_acc: 0.507000
(Iteration 2901 / 3820) loss: 1.291224
(Iteration 3001 / 3820) loss: 1.167918
(Epoch 16 / 20) train acc: 0.560000; val_acc: 0.507000
(Iteration 3101 / 3820) loss: 1.255945
(Iteration 3201 / 3820) loss: 1.345913
(Epoch 17 / 20) train acc: 0.560000; val_acc: 0.507000
(Iteration 3301 / 3820) loss: 1.225775
(Iteration 3401 / 3820) loss: 1.211493
(Epoch 18 / 20) train acc: 0.557000; val_acc: 0.507000
(Iteration 3501 / 3820) loss: 1.296285
(Iteration 3601 / 3820) loss: 1.326609
(Epoch 19 / 20) train acc: 0.540000; val_acc: 0.507000
(Iteration 3701 / 3820) loss: 1.306407
```

```
(Iteration 3801 / 3820) loss: 1.277859
(Epoch 20 / 20) train acc: 0.545000; val_acc: 0.507000
```

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.

[23]:
```
best_model = model
y_val_pred = np.argmax(best_model.loss(data['X_val']), axis=1)
print('Validation set accuracy: ', (y_val_pred == data['y_val']).mean())
```

```
Validation set accuracy:  0.512
```

[24]:
```
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:  0.506
```

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. Train on a larger dataset, 3. Increase the regularization strength.

YourExplanation :

1. **Train on a larger dataset:**

Explanation: Increasing the size of the training dataset can help improve generalization. A larger dataset provides neural network with more diverse examples, helping it learn better representations and reduce overfitting. This can potentially reduce the gap between training and testing accuracy.

2. **Add more hidden units:**

Explanation: Increasing the number of hidden units in the model can enable it to learn more complex functions and potentially memorize the training data. However, we can mitigate this problem by implementing the right regularization technique.hidden units.

3. **Increase the regularization strength:**

Explanation: Increasing the regularization strength, such as using L1 or L2 regularization, can help reduce overfitting. Regularization penalizes large weights or complex models, making the model generalize better to unseen data. Increasing regularization strength can help decrease the gap between training and testing accuracy by discouraging the model from fitting noise in the training data.

[ ]:

features

October 6, 2023

[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 = 'cs231n/assignments/assignment1/'
assert FOLDERNAME is not None, "[!] Enter the foldername."

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

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

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

1 Image features exercise

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

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

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

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


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

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

1.1 Load data

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

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

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

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

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

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

```
    y_test = y_test[mask]

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

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

1.2 Extract Features

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

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

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

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

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

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

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

# Preprocessing: Add a bias dimension
X_train_feats = np.hstack([X_train_feats, np.ones((X_train_feats.shape[0], 1))])
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)*****
# permute over learning_rates and regularization_strengths
grid_search = [(lr, reg) for lr in learning_rates \
               for reg in regularization_strengths]

for config_num, config in enumerate(grid_search):
    print("Hyperparam config #{} of #{}".format(config_num+1, len(grid_search)))
    print("Hyperparam config: {}".format(config))
    lr, reg = config

    svm = LinearSVM()

    # train a linear SVM on the training set
    loss_hist = svm.train(X_train_feats, y_train, learning_rate=lr,
                          reg=reg, num_iters=2000, verbose=False)

    # make predictions on the training and validation sets
    y_train_pred = svm.predict(X_train_feats)
    y_val_pred = svm.predict(X_val_feats)

    # compute the accuracy on the training and validation sets
```

```
    current_y_train_accuracy = np.mean(y_train_pred == y_train)
    current_y_val_accuracy = np.mean(y_val_pred == y_val)

    # store results
    results[(lr, reg)] = (current_y_train_accuracy, current_y_val_accuracy)

    # store the best validation accuracy and the LinearSVM object
    if current_y_val_accuracy > best_val:
        best_val = current_y_val_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)
```

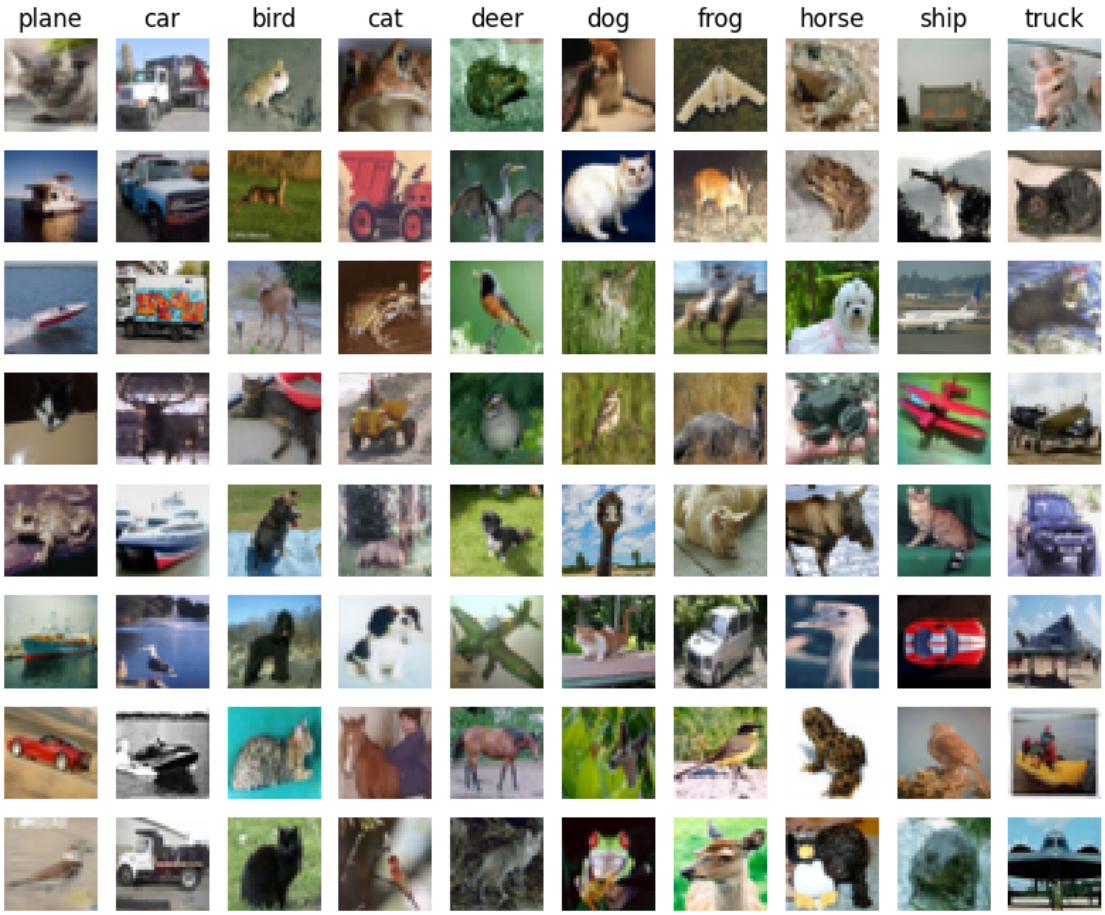
```
Hyperparam config #1 of #9
Hyperparam config: (1e-09, 50000.0)
Hyperparam config #2 of #9
Hyperparam config: (1e-09, 500000.0)
Hyperparam config #3 of #9
Hyperparam config: (1e-09, 5000000.0)
Hyperparam config #4 of #9
Hyperparam config: (1e-08, 50000.0)
Hyperparam config #5 of #9
Hyperparam config: (1e-08, 500000.0)
Hyperparam config #6 of #9
Hyperparam config: (1e-08, 5000000.0)
Hyperparam config #7 of #9
Hyperparam config: (1e-07, 50000.0)
Hyperparam config #8 of #9
Hyperparam config: (1e-07, 500000.0)
Hyperparam config #9 of #9
Hyperparam config: (1e-07, 5000000.0)
lr 1.000000e-09 reg 5.000000e+04 train accuracy: 0.091837 val accuracy: 0.091000
lr 1.000000e-09 reg 5.000000e+05 train accuracy: 0.109122 val accuracy: 0.117000
lr 1.000000e-09 reg 5.000000e+06 train accuracy: 0.413306 val accuracy: 0.418000
lr 1.000000e-08 reg 5.000000e+04 train accuracy: 0.103143 val accuracy: 0.094000
lr 1.000000e-08 reg 5.000000e+05 train accuracy: 0.415755 val accuracy: 0.422000
lr 1.000000e-08 reg 5.000000e+06 train accuracy: 0.416082 val accuracy: 0.409000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.412816 val accuracy: 0.416000
```

```
lr 1.000000e-07 reg 5.000000e+05 train accuracy: 0.398122 val accuracy: 0.390000
lr 1.000000e-07 reg 5.000000e+06 train accuracy: 0.347327 val accuracy: 0.353000
best validation accuracy achieved: 0.422000
```

[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 :

Certainly, if there are occasional cases where images of cats, dogs, and even cars are wrongly categorized as birds, and vice versa. This occurs because these animals share common visual characteristics such as facial features (eyes, ears, nose, mouth) and sometimes quadruped legs, which are captured by the HOG feature. Additionally, they often have similar color patterns, represented by the color histogram. Similarly, confusion arises between horses and deer due to their analogous textures and color features. Cars and trucks also get mixed up for similar reasons. There are also instances of misclassification due to background similarities, like in the case of planes, where the sky and sea colors resemble each other, leading to confusion with ship images.

In summary, relying solely on HOG and color histogram feature vectors doesn't provide impeccable accuracy in distinguishing between these classes. While HOG helps with texture analysis within images, it doesn't account for various forms of intra-class variation, such as rotation, scaling, translation, illumination, and posture deformation.

To enhance our model's accuracy in such scenarios, we can incorporate the Scale-invariant feature transform (SIFT) by Lowe. Furthermore, when using color histogram features alongside more advanced feature descriptors like HOG or SIFT, we can assign them lower weight to mitigate the issue of simplistic color-based class differentiation.

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)
```

[11]:
```
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)*****
## generate random combinations of hyperparameters given arrays of potential␣
 ↪values

input_dim = X_train_feats.shape[1]
hidden_dim = 500
```

```
num_classes = 10

# Define the hyperparameters
learning_rates = [2.5e-1, 3e-1, 3.75e-1]
regularization_strengths = [2e-3, 2.5e-3, 3e-3]
learning_rate_decay = [0.9, 0.95, 0.99]

best_val = -1
best_net = None

for lr in learning_rates:
    for reg in regularization_strengths:
        for decay in learning_rate_decay:
            net = TwoLayerNet(input_dim, hidden_dim, num_classes)
            solver = Solver(net, data={'X_train': X_train_feats,
                                       'y_train': y_train,
                                       'X_val': X_val_feats,
                                       'y_val': y_val},
                            update_rule='sgd',
                            optim_config={
                                'learning_rate': lr,
                            },
                            lr_decay=decay,
                            num_epochs=10,
                            batch_size=200,
                            print_every=100)

            solver.train()
            val_acc = solver.best_val_acc

            if val_acc > best_val:
                best_val = val_acc
                best_net = net

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


pass

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

```
(Iteration 1 / 2450) loss: 2.302590
(Epoch 0 / 10) train acc: 0.105000; val_acc: 0.105000
(Iteration 101 / 2450) loss: 1.680219
(Iteration 201 / 2450) loss: 1.408228
```

```
(Epoch 1 / 10) train acc: 0.512000; val_acc: 0.499000
(Iteration 301 / 2450) loss: 1.456595
(Iteration 401 / 2450) loss: 1.375280
(Epoch 2 / 10) train acc: 0.519000; val_acc: 0.511000
(Iteration 501 / 2450) loss: 1.354285
(Iteration 601 / 2450) loss: 1.233601
(Iteration 701 / 2450) loss: 1.342681
(Epoch 3 / 10) train acc: 0.557000; val_acc: 0.559000
(Iteration 801 / 2450) loss: 1.132941
(Iteration 901 / 2450) loss: 1.056441
(Epoch 4 / 10) train acc: 0.573000; val_acc: 0.558000
(Iteration 1001 / 2450) loss: 1.149575
(Iteration 1101 / 2450) loss: 1.129105
(Iteration 1201 / 2450) loss: 1.170073
(Epoch 5 / 10) train acc: 0.630000; val_acc: 0.576000
(Iteration 1301 / 2450) loss: 1.050413
(Iteration 1401 / 2450) loss: 1.016841
(Epoch 6 / 10) train acc: 0.629000; val_acc: 0.590000
(Iteration 1501 / 2450) loss: 1.037895
(Iteration 1601 / 2450) loss: 1.036302
(Iteration 1701 / 2450) loss: 1.080999
(Epoch 7 / 10) train acc: 0.641000; val_acc: 0.598000
(Iteration 1801 / 2450) loss: 0.965121
(Iteration 1901 / 2450) loss: 0.928868
(Epoch 8 / 10) train acc: 0.643000; val_acc: 0.593000
(Iteration 2001 / 2450) loss: 0.905653
(Iteration 2101 / 2450) loss: 0.875648
(Iteration 2201 / 2450) loss: 0.888758
(Epoch 9 / 10) train acc: 0.694000; val_acc: 0.598000
(Iteration 2301 / 2450) loss: 1.085947
(Iteration 2401 / 2450) loss: 0.976259
(Epoch 10 / 10) train acc: 0.706000; val_acc: 0.623000
(Iteration 1 / 2450) loss: 2.302571
(Epoch 0 / 10) train acc: 0.106000; val_acc: 0.078000
(Iteration 101 / 2450) loss: 1.546675
(Iteration 201 / 2450) loss: 1.476675
(Epoch 1 / 10) train acc: 0.503000; val_acc: 0.511000
(Iteration 301 / 2450) loss: 1.439773
(Iteration 401 / 2450) loss: 1.326376
(Epoch 2 / 10) train acc: 0.485000; val_acc: 0.519000
(Iteration 501 / 2450) loss: 1.301876
(Iteration 601 / 2450) loss: 1.269997
(Iteration 701 / 2450) loss: 1.093698
(Epoch 3 / 10) train acc: 0.563000; val_acc: 0.542000
(Iteration 801 / 2450) loss: 1.169613
(Iteration 901 / 2450) loss: 1.253783
(Epoch 4 / 10) train acc: 0.622000; val_acc: 0.553000
(Iteration 1001 / 2450) loss: 1.104734
```

```
(Iteration 1101 / 2450) loss: 1.147465
(Iteration 1201 / 2450) loss: 1.091180
(Epoch 5 / 10) train acc: 0.617000; val_acc: 0.570000
(Iteration 1301 / 2450) loss: 1.112119
(Iteration 1401 / 2450) loss: 0.961545
(Epoch 6 / 10) train acc: 0.654000; val_acc: 0.581000
(Iteration 1501 / 2450) loss: 1.145792
(Iteration 1601 / 2450) loss: 1.013358
(Iteration 1701 / 2450) loss: 0.935212
(Epoch 7 / 10) train acc: 0.665000; val_acc: 0.592000
(Iteration 1801 / 2450) loss: 0.993257
(Iteration 1901 / 2450) loss: 0.971591
(Epoch 8 / 10) train acc: 0.672000; val_acc: 0.588000
(Iteration 2001 / 2450) loss: 0.990226
(Iteration 2101 / 2450) loss: 0.928067
(Iteration 2201 / 2450) loss: 1.011097
(Epoch 9 / 10) train acc: 0.710000; val_acc: 0.596000
(Iteration 2301 / 2450) loss: 0.844891
(Iteration 2401 / 2450) loss: 0.853887
(Epoch 10 / 10) train acc: 0.687000; val_acc: 0.603000
(Iteration 1 / 2450) loss: 2.302609
(Epoch 0 / 10) train acc: 0.107000; val_acc: 0.079000
(Iteration 101 / 2450) loss: 1.616794
(Iteration 201 / 2450) loss: 1.410369
(Epoch 1 / 10) train acc: 0.511000; val_acc: 0.500000
(Iteration 301 / 2450) loss: 1.176935
(Iteration 401 / 2450) loss: 1.289928
(Epoch 2 / 10) train acc: 0.532000; val_acc: 0.529000
(Iteration 501 / 2450) loss: 1.290387
(Iteration 601 / 2450) loss: 1.192427
(Iteration 701 / 2450) loss: 1.046975
(Epoch 3 / 10) train acc: 0.581000; val_acc: 0.553000
(Iteration 801 / 2450) loss: 1.169429
(Iteration 901 / 2450) loss: 1.232929
(Epoch 4 / 10) train acc: 0.608000; val_acc: 0.568000
(Iteration 1001 / 2450) loss: 0.969951
(Iteration 1101 / 2450) loss: 1.041741
(Iteration 1201 / 2450) loss: 1.184966
(Epoch 5 / 10) train acc: 0.591000; val_acc: 0.576000
(Iteration 1301 / 2450) loss: 0.977164
(Iteration 1401 / 2450) loss: 1.059937
(Epoch 6 / 10) train acc: 0.645000; val_acc: 0.579000
(Iteration 1501 / 2450) loss: 1.033895
(Iteration 1601 / 2450) loss: 0.937122
(Iteration 1701 / 2450) loss: 1.079727
(Epoch 7 / 10) train acc: 0.641000; val_acc: 0.580000
(Iteration 1801 / 2450) loss: 0.991994
(Iteration 1901 / 2450) loss: 0.887567
```

```
(Epoch 8 / 10) train acc: 0.663000; val_acc: 0.587000
(Iteration 2001 / 2450) loss: 1.003819
(Iteration 2101 / 2450) loss: 0.845583
(Iteration 2201 / 2450) loss: 0.853962
(Epoch 9 / 10) train acc: 0.682000; val_acc: 0.599000
(Iteration 2301 / 2450) loss: 0.832006
(Iteration 2401 / 2450) loss: 0.813413
(Epoch 10 / 10) train acc: 0.693000; val_acc: 0.576000
(Iteration 1 / 2450) loss: 2.302548
(Epoch 0 / 10) train acc: 0.108000; val_acc: 0.087000
(Iteration 101 / 2450) loss: 1.651600
(Iteration 201 / 2450) loss: 1.388117
(Epoch 1 / 10) train acc: 0.518000; val_acc: 0.507000
(Iteration 301 / 2450) loss: 1.188503
(Iteration 401 / 2450) loss: 1.332189
(Epoch 2 / 10) train acc: 0.534000; val_acc: 0.526000
(Iteration 501 / 2450) loss: 1.311316
(Iteration 601 / 2450) loss: 1.211483
(Iteration 701 / 2450) loss: 1.207219
(Epoch 3 / 10) train acc: 0.546000; val_acc: 0.538000
(Iteration 801 / 2450) loss: 1.347609
(Iteration 901 / 2450) loss: 1.192547
(Epoch 4 / 10) train acc: 0.585000; val_acc: 0.555000
(Iteration 1001 / 2450) loss: 1.107711
(Iteration 1101 / 2450) loss: 1.207395
(Iteration 1201 / 2450) loss: 1.035080
(Epoch 5 / 10) train acc: 0.616000; val_acc: 0.571000
(Iteration 1301 / 2450) loss: 1.232361
(Iteration 1401 / 2450) loss: 0.933775
(Epoch 6 / 10) train acc: 0.625000; val_acc: 0.578000
(Iteration 1501 / 2450) loss: 1.081037
(Iteration 1601 / 2450) loss: 0.941821
(Iteration 1701 / 2450) loss: 0.962163
(Epoch 7 / 10) train acc: 0.653000; val_acc: 0.586000
(Iteration 1801 / 2450) loss: 0.998724
(Iteration 1901 / 2450) loss: 1.110704
(Epoch 8 / 10) train acc: 0.666000; val_acc: 0.575000
(Iteration 2001 / 2450) loss: 0.984894
(Iteration 2101 / 2450) loss: 1.007879
(Iteration 2201 / 2450) loss: 0.929544
(Epoch 9 / 10) train acc: 0.652000; val_acc: 0.585000
(Iteration 2301 / 2450) loss: 1.014011
(Iteration 2401 / 2450) loss: 0.928911
(Epoch 10 / 10) train acc: 0.710000; val_acc: 0.605000
(Iteration 1 / 2450) loss: 2.302603
(Epoch 0 / 10) train acc: 0.124000; val_acc: 0.109000
(Iteration 101 / 2450) loss: 1.703738
(Iteration 201 / 2450) loss: 1.358496
```

```
(Epoch 1 / 10) train acc: 0.545000; val_acc: 0.513000
(Iteration 301 / 2450) loss: 1.398140
(Iteration 401 / 2450) loss: 1.223014
(Epoch 2 / 10) train acc: 0.548000; val_acc: 0.532000
(Iteration 501 / 2450) loss: 1.371287
(Iteration 601 / 2450) loss: 1.443722
(Iteration 701 / 2450) loss: 1.222800
(Epoch 3 / 10) train acc: 0.550000; val_acc: 0.548000
(Iteration 801 / 2450) loss: 1.206559
(Iteration 901 / 2450) loss: 1.106822
(Epoch 4 / 10) train acc: 0.581000; val_acc: 0.575000
(Iteration 1001 / 2450) loss: 1.118569
(Iteration 1101 / 2450) loss: 1.144398
(Iteration 1201 / 2450) loss: 1.114489
(Epoch 5 / 10) train acc: 0.625000; val_acc: 0.575000
(Iteration 1301 / 2450) loss: 1.114156
(Iteration 1401 / 2450) loss: 0.954999
(Epoch 6 / 10) train acc: 0.617000; val_acc: 0.588000
(Iteration 1501 / 2450) loss: 1.046193
(Iteration 1601 / 2450) loss: 0.980940
(Iteration 1701 / 2450) loss: 1.038838
(Epoch 7 / 10) train acc: 0.643000; val_acc: 0.600000
(Iteration 1801 / 2450) loss: 1.058565
(Iteration 1901 / 2450) loss: 0.931590
(Epoch 8 / 10) train acc: 0.666000; val_acc: 0.585000
(Iteration 2001 / 2450) loss: 0.830425
(Iteration 2101 / 2450) loss: 0.815777
(Iteration 2201 / 2450) loss: 0.865889
(Epoch 9 / 10) train acc: 0.689000; val_acc: 0.598000
(Iteration 2301 / 2450) loss: 0.828477
(Iteration 2401 / 2450) loss: 0.958092
(Epoch 10 / 10) train acc: 0.655000; val_acc: 0.590000
(Iteration 1 / 2450) loss: 2.302594
(Epoch 0 / 10) train acc: 0.103000; val_acc: 0.112000
(Iteration 101 / 2450) loss: 1.652258
(Iteration 201 / 2450) loss: 1.425352
(Epoch 1 / 10) train acc: 0.503000; val_acc: 0.505000
(Iteration 301 / 2450) loss: 1.251127
(Iteration 401 / 2450) loss: 1.442017
(Epoch 2 / 10) train acc: 0.533000; val_acc: 0.517000
(Iteration 501 / 2450) loss: 1.404848
(Iteration 601 / 2450) loss: 1.175890
(Iteration 701 / 2450) loss: 1.193645
(Epoch 3 / 10) train acc: 0.572000; val_acc: 0.533000
(Iteration 801 / 2450) loss: 1.151646
(Iteration 901 / 2450) loss: 1.230333
(Epoch 4 / 10) train acc: 0.613000; val_acc: 0.556000
(Iteration 1001 / 2450) loss: 1.151620
```

```
(Iteration 1101 / 2450) loss: 1.149746
(Iteration 1201 / 2450) loss: 0.955736
(Epoch 5 / 10) train acc: 0.623000; val_acc: 0.571000
(Iteration 1301 / 2450) loss: 1.073147
(Iteration 1401 / 2450) loss: 0.931836
(Epoch 6 / 10) train acc: 0.647000; val_acc: 0.592000
(Iteration 1501 / 2450) loss: 1.070463
(Iteration 1601 / 2450) loss: 1.008870
(Iteration 1701 / 2450) loss: 1.191293
(Epoch 7 / 10) train acc: 0.666000; val_acc: 0.597000
(Iteration 1801 / 2450) loss: 0.866616
(Iteration 1901 / 2450) loss: 0.957306
(Epoch 8 / 10) train acc: 0.670000; val_acc: 0.590000
(Iteration 2001 / 2450) loss: 1.047245
(Iteration 2101 / 2450) loss: 0.890739
(Iteration 2201 / 2450) loss: 0.982567
(Epoch 9 / 10) train acc: 0.664000; val_acc: 0.595000
(Iteration 2301 / 2450) loss: 0.817544
(Iteration 2401 / 2450) loss: 0.850784
(Epoch 10 / 10) train acc: 0.731000; val_acc: 0.592000
(Iteration 1 / 2450) loss: 2.302592
(Epoch 0 / 10) train acc: 0.098000; val_acc: 0.098000
(Iteration 101 / 2450) loss: 1.686142
(Iteration 201 / 2450) loss: 1.453012
(Epoch 1 / 10) train acc: 0.516000; val_acc: 0.509000
(Iteration 301 / 2450) loss: 1.316147
(Iteration 401 / 2450) loss: 1.139219
(Epoch 2 / 10) train acc: 0.530000; val_acc: 0.520000
(Iteration 501 / 2450) loss: 1.269003
(Iteration 601 / 2450) loss: 1.211899
(Iteration 701 / 2450) loss: 1.242482
(Epoch 3 / 10) train acc: 0.559000; val_acc: 0.556000
(Iteration 801 / 2450) loss: 1.179953
(Iteration 901 / 2450) loss: 1.196639
(Epoch 4 / 10) train acc: 0.586000; val_acc: 0.559000
(Iteration 1001 / 2450) loss: 1.089720
(Iteration 1101 / 2450) loss: 1.149023
(Iteration 1201 / 2450) loss: 1.135365
(Epoch 5 / 10) train acc: 0.606000; val_acc: 0.572000
(Iteration 1301 / 2450) loss: 1.122102
(Iteration 1401 / 2450) loss: 0.989735
(Epoch 6 / 10) train acc: 0.640000; val_acc: 0.588000
(Iteration 1501 / 2450) loss: 0.999802
(Iteration 1601 / 2450) loss: 0.957139
(Iteration 1701 / 2450) loss: 0.993294
(Epoch 7 / 10) train acc: 0.622000; val_acc: 0.593000
(Iteration 1801 / 2450) loss: 1.109935
(Iteration 1901 / 2450) loss: 0.989502
```

```
(Epoch 8 / 10) train acc: 0.672000; val_acc: 0.597000
(Iteration 2001 / 2450) loss: 0.906633
(Iteration 2101 / 2450) loss: 0.950642
(Iteration 2201 / 2450) loss: 0.947246
(Epoch 9 / 10) train acc: 0.668000; val_acc: 0.594000
(Iteration 2301 / 2450) loss: 0.918982
(Iteration 2401 / 2450) loss: 0.970767
(Epoch 10 / 10) train acc: 0.706000; val_acc: 0.598000
(Iteration 1 / 2450) loss: 2.302592
(Epoch 0 / 10) train acc: 0.088000; val_acc: 0.098000
(Iteration 101 / 2450) loss: 1.773431
(Iteration 201 / 2450) loss: 1.373547
(Epoch 1 / 10) train acc: 0.533000; val_acc: 0.508000
(Iteration 301 / 2450) loss: 1.439413
(Iteration 401 / 2450) loss: 1.218146
(Epoch 2 / 10) train acc: 0.575000; val_acc: 0.520000
(Iteration 501 / 2450) loss: 1.403005
(Iteration 601 / 2450) loss: 1.351768
(Iteration 701 / 2450) loss: 1.315820
(Epoch 3 / 10) train acc: 0.577000; val_acc: 0.565000
(Iteration 801 / 2450) loss: 1.135354
(Iteration 901 / 2450) loss: 1.251397
(Epoch 4 / 10) train acc: 0.613000; val_acc: 0.566000
(Iteration 1001 / 2450) loss: 1.127346
(Iteration 1101 / 2450) loss: 1.128167
(Iteration 1201 / 2450) loss: 1.203917
(Epoch 5 / 10) train acc: 0.603000; val_acc: 0.570000
(Iteration 1301 / 2450) loss: 1.025409
(Iteration 1401 / 2450) loss: 0.984978
(Epoch 6 / 10) train acc: 0.642000; val_acc: 0.576000
(Iteration 1501 / 2450) loss: 0.878353
(Iteration 1601 / 2450) loss: 0.879271
(Iteration 1701 / 2450) loss: 0.959680
(Epoch 7 / 10) train acc: 0.690000; val_acc: 0.576000
(Iteration 1801 / 2450) loss: 0.857382
(Iteration 1901 / 2450) loss: 0.871493
(Epoch 8 / 10) train acc: 0.680000; val_acc: 0.594000
(Iteration 2001 / 2450) loss: 0.986654
(Iteration 2101 / 2450) loss: 1.109093
(Iteration 2201 / 2450) loss: 0.820803
(Epoch 9 / 10) train acc: 0.715000; val_acc: 0.589000
(Iteration 2301 / 2450) loss: 0.834886
(Iteration 2401 / 2450) loss: 0.899141
(Epoch 10 / 10) train acc: 0.703000; val_acc: 0.591000
(Iteration 1 / 2450) loss: 2.302577
(Epoch 0 / 10) train acc: 0.135000; val_acc: 0.159000
(Iteration 101 / 2450) loss: 1.635081
(Iteration 201 / 2450) loss: 1.489958
```

```
(Epoch 1 / 10) train acc: 0.500000; val_acc: 0.522000
(Iteration 301 / 2450) loss: 1.259024
(Iteration 401 / 2450) loss: 1.337504
(Epoch 2 / 10) train acc: 0.547000; val_acc: 0.533000
(Iteration 501 / 2450) loss: 1.324039
(Iteration 601 / 2450) loss: 1.107041
(Iteration 701 / 2450) loss: 1.215928
(Epoch 3 / 10) train acc: 0.590000; val_acc: 0.554000
(Iteration 801 / 2450) loss: 1.198566
(Iteration 901 / 2450) loss: 1.170783
(Epoch 4 / 10) train acc: 0.612000; val_acc: 0.557000
(Iteration 1001 / 2450) loss: 1.216639
(Iteration 1101 / 2450) loss: 1.146254
(Iteration 1201 / 2450) loss: 1.139356
(Epoch 5 / 10) train acc: 0.609000; val_acc: 0.566000
(Iteration 1301 / 2450) loss: 0.916929
(Iteration 1401 / 2450) loss: 0.955122
(Epoch 6 / 10) train acc: 0.611000; val_acc: 0.556000
(Iteration 1501 / 2450) loss: 1.160241
(Iteration 1601 / 2450) loss: 0.889112
(Iteration 1701 / 2450) loss: 1.074271
(Epoch 7 / 10) train acc: 0.649000; val_acc: 0.599000
(Iteration 1801 / 2450) loss: 1.052299
(Iteration 1901 / 2450) loss: 0.842121
(Epoch 8 / 10) train acc: 0.682000; val_acc: 0.586000
(Iteration 2001 / 2450) loss: 0.974252
(Iteration 2101 / 2450) loss: 0.923376
(Iteration 2201 / 2450) loss: 0.973748
(Epoch 9 / 10) train acc: 0.709000; val_acc: 0.588000
(Iteration 2301 / 2450) loss: 0.839673
(Iteration 2401 / 2450) loss: 0.924742
(Epoch 10 / 10) train acc: 0.724000; val_acc: 0.581000
(Iteration 1 / 2450) loss: 2.302556
(Epoch 0 / 10) train acc: 0.097000; val_acc: 0.078000
(Iteration 101 / 2450) loss: 1.617528
(Iteration 201 / 2450) loss: 1.434424
(Epoch 1 / 10) train acc: 0.520000; val_acc: 0.509000
(Iteration 301 / 2450) loss: 1.411699
(Iteration 401 / 2450) loss: 1.475430
(Epoch 2 / 10) train acc: 0.551000; val_acc: 0.529000
(Iteration 501 / 2450) loss: 1.256413
(Iteration 601 / 2450) loss: 1.132552
(Iteration 701 / 2450) loss: 1.165293
(Epoch 3 / 10) train acc: 0.594000; val_acc: 0.556000
(Iteration 801 / 2450) loss: 1.314951
(Iteration 901 / 2450) loss: 1.065590
(Epoch 4 / 10) train acc: 0.590000; val_acc: 0.561000
(Iteration 1001 / 2450) loss: 1.078578
```

```
(Iteration 1101 / 2450) loss: 1.104594
(Iteration 1201 / 2450) loss: 1.080943
(Epoch 5 / 10) train acc: 0.650000; val_acc: 0.561000
(Iteration 1301 / 2450) loss: 1.092316
(Iteration 1401 / 2450) loss: 1.013566
(Epoch 6 / 10) train acc: 0.664000; val_acc: 0.576000
(Iteration 1501 / 2450) loss: 0.981438
(Iteration 1601 / 2450) loss: 1.023073
(Iteration 1701 / 2450) loss: 1.000690
(Epoch 7 / 10) train acc: 0.648000; val_acc: 0.588000
(Iteration 1801 / 2450) loss: 1.040447
(Iteration 1901 / 2450) loss: 0.955241
(Epoch 8 / 10) train acc: 0.692000; val_acc: 0.592000
(Iteration 2001 / 2450) loss: 0.894488
(Iteration 2101 / 2450) loss: 0.852531
(Iteration 2201 / 2450) loss: 0.938368
(Epoch 9 / 10) train acc: 0.704000; val_acc: 0.595000
(Iteration 2301 / 2450) loss: 0.872069
(Iteration 2401 / 2450) loss: 0.809887
(Epoch 10 / 10) train acc: 0.707000; val_acc: 0.595000
(Iteration 1 / 2450) loss: 2.302579
(Epoch 0 / 10) train acc: 0.097000; val_acc: 0.102000
(Iteration 101 / 2450) loss: 1.625121
(Iteration 201 / 2450) loss: 1.399712
(Epoch 1 / 10) train acc: 0.510000; val_acc: 0.509000
(Iteration 301 / 2450) loss: 1.390677
(Iteration 401 / 2450) loss: 1.407521
(Epoch 2 / 10) train acc: 0.548000; val_acc: 0.526000
(Iteration 501 / 2450) loss: 1.322300
(Iteration 601 / 2450) loss: 1.234637
(Iteration 701 / 2450) loss: 1.227318
(Epoch 3 / 10) train acc: 0.603000; val_acc: 0.549000
(Iteration 801 / 2450) loss: 1.118280
(Iteration 901 / 2450) loss: 1.002523
(Epoch 4 / 10) train acc: 0.604000; val_acc: 0.576000
(Iteration 1001 / 2450) loss: 1.062846
(Iteration 1101 / 2450) loss: 0.965082
(Iteration 1201 / 2450) loss: 1.026164
(Epoch 5 / 10) train acc: 0.641000; val_acc: 0.568000
(Iteration 1301 / 2450) loss: 1.112514
(Iteration 1401 / 2450) loss: 0.944727
(Epoch 6 / 10) train acc: 0.634000; val_acc: 0.588000
(Iteration 1501 / 2450) loss: 1.010038
(Iteration 1601 / 2450) loss: 1.080781
(Iteration 1701 / 2450) loss: 0.973450
(Epoch 7 / 10) train acc: 0.665000; val_acc: 0.601000
(Iteration 1801 / 2450) loss: 0.974266
(Iteration 1901 / 2450) loss: 1.099848
```

```
(Epoch 8 / 10) train acc: 0.673000; val_acc: 0.588000
(Iteration 2001 / 2450) loss: 0.887619
(Iteration 2101 / 2450) loss: 0.805629
(Iteration 2201 / 2450) loss: 0.773059
(Epoch 9 / 10) train acc: 0.708000; val_acc: 0.613000
(Iteration 2301 / 2450) loss: 0.866560
(Iteration 2401 / 2450) loss: 0.751936
(Epoch 10 / 10) train acc: 0.715000; val_acc: 0.596000
(Iteration 1 / 2450) loss: 2.302558
(Epoch 0 / 10) train acc: 0.105000; val_acc: 0.107000
(Iteration 101 / 2450) loss: 1.606736
(Iteration 201 / 2450) loss: 1.314402
(Epoch 1 / 10) train acc: 0.503000; val_acc: 0.501000
(Iteration 301 / 2450) loss: 1.354438
(Iteration 401 / 2450) loss: 1.411075
(Epoch 2 / 10) train acc: 0.539000; val_acc: 0.546000
(Iteration 501 / 2450) loss: 1.247008
(Iteration 601 / 2450) loss: 1.214788
(Iteration 701 / 2450) loss: 1.243733
(Epoch 3 / 10) train acc: 0.571000; val_acc: 0.561000
(Iteration 801 / 2450) loss: 1.081534
(Iteration 901 / 2450) loss: 1.145989
(Epoch 4 / 10) train acc: 0.629000; val_acc: 0.568000
(Iteration 1001 / 2450) loss: 1.408750
(Iteration 1101 / 2450) loss: 1.025978
(Iteration 1201 / 2450) loss: 1.053640
(Epoch 5 / 10) train acc: 0.642000; val_acc: 0.572000
(Iteration 1301 / 2450) loss: 0.984155
(Iteration 1401 / 2450) loss: 1.034527
(Epoch 6 / 10) train acc: 0.641000; val_acc: 0.595000
(Iteration 1501 / 2450) loss: 1.022279
(Iteration 1601 / 2450) loss: 0.984025
(Iteration 1701 / 2450) loss: 0.814723
(Epoch 7 / 10) train acc: 0.691000; val_acc: 0.576000
(Iteration 1801 / 2450) loss: 0.943754
(Iteration 1901 / 2450) loss: 0.912201
(Epoch 8 / 10) train acc: 0.687000; val_acc: 0.587000
(Iteration 2001 / 2450) loss: 0.893751
(Iteration 2101 / 2450) loss: 0.812400
(Iteration 2201 / 2450) loss: 0.828438
(Epoch 9 / 10) train acc: 0.735000; val_acc: 0.595000
(Iteration 2301 / 2450) loss: 0.875368
(Iteration 2401 / 2450) loss: 0.706299
(Epoch 10 / 10) train acc: 0.755000; val_acc: 0.571000
(Iteration 1 / 2450) loss: 2.302616
(Epoch 0 / 10) train acc: 0.085000; val_acc: 0.107000
(Iteration 101 / 2450) loss: 1.541885
(Iteration 201 / 2450) loss: 1.386938
```

```
(Epoch 1 / 10) train acc: 0.524000; val_acc: 0.518000
(Iteration 301 / 2450) loss: 1.417305
(Iteration 401 / 2450) loss: 1.225116
(Epoch 2 / 10) train acc: 0.541000; val_acc: 0.531000
(Iteration 501 / 2450) loss: 1.278346
(Iteration 601 / 2450) loss: 1.207113
(Iteration 701 / 2450) loss: 1.202680
(Epoch 3 / 10) train acc: 0.583000; val_acc: 0.542000
(Iteration 801 / 2450) loss: 1.111258
(Iteration 901 / 2450) loss: 1.255189
(Epoch 4 / 10) train acc: 0.598000; val_acc: 0.561000
(Iteration 1001 / 2450) loss: 1.039034
(Iteration 1101 / 2450) loss: 1.186382
(Iteration 1201 / 2450) loss: 1.043002
(Epoch 5 / 10) train acc: 0.615000; val_acc: 0.582000
(Iteration 1301 / 2450) loss: 1.233770
(Iteration 1401 / 2450) loss: 1.032268
(Epoch 6 / 10) train acc: 0.637000; val_acc: 0.582000
(Iteration 1501 / 2450) loss: 1.023624
(Iteration 1601 / 2450) loss: 0.996458
(Iteration 1701 / 2450) loss: 0.932031
(Epoch 7 / 10) train acc: 0.646000; val_acc: 0.593000
(Iteration 1801 / 2450) loss: 0.904906
(Iteration 1901 / 2450) loss: 0.787306
(Epoch 8 / 10) train acc: 0.687000; val_acc: 0.602000
(Iteration 2001 / 2450) loss: 0.905181
(Iteration 2101 / 2450) loss: 1.012744
(Iteration 2201 / 2450) loss: 0.776307
(Epoch 9 / 10) train acc: 0.680000; val_acc: 0.600000
(Iteration 2301 / 2450) loss: 0.919226
(Iteration 2401 / 2450) loss: 0.968803
(Epoch 10 / 10) train acc: 0.712000; val_acc: 0.606000
(Iteration 1 / 2450) loss: 2.302598
(Epoch 0 / 10) train acc: 0.092000; val_acc: 0.098000
(Iteration 101 / 2450) loss: 1.501322
(Iteration 201 / 2450) loss: 1.428054
(Epoch 1 / 10) train acc: 0.474000; val_acc: 0.487000
(Iteration 301 / 2450) loss: 1.348629
(Iteration 401 / 2450) loss: 1.348553
(Epoch 2 / 10) train acc: 0.543000; val_acc: 0.534000
(Iteration 501 / 2450) loss: 1.111496
(Iteration 601 / 2450) loss: 1.219859
(Iteration 701 / 2450) loss: 1.174990
(Epoch 3 / 10) train acc: 0.586000; val_acc: 0.551000
(Iteration 801 / 2450) loss: 1.113552
(Iteration 901 / 2450) loss: 1.175646
(Epoch 4 / 10) train acc: 0.624000; val_acc: 0.556000
(Iteration 1001 / 2450) loss: 1.085740
```

```
(Iteration 1101 / 2450) loss: 1.062987
(Iteration 1201 / 2450) loss: 0.931131
(Epoch 5 / 10) train acc: 0.649000; val_acc: 0.572000
(Iteration 1301 / 2450) loss: 0.895707
(Iteration 1401 / 2450) loss: 0.972806
(Epoch 6 / 10) train acc: 0.662000; val_acc: 0.592000
(Iteration 1501 / 2450) loss: 1.115396
(Iteration 1601 / 2450) loss: 0.989848
(Iteration 1701 / 2450) loss: 0.930994
(Epoch 7 / 10) train acc: 0.687000; val_acc: 0.598000
(Iteration 1801 / 2450) loss: 1.030984
(Iteration 1901 / 2450) loss: 0.993944
(Epoch 8 / 10) train acc: 0.687000; val_acc: 0.598000
(Iteration 2001 / 2450) loss: 0.914280
(Iteration 2101 / 2450) loss: 0.784992
(Iteration 2201 / 2450) loss: 0.895433
(Epoch 9 / 10) train acc: 0.695000; val_acc: 0.595000
(Iteration 2301 / 2450) loss: 0.819557
(Iteration 2401 / 2450) loss: 0.799528
(Epoch 10 / 10) train acc: 0.718000; val_acc: 0.602000
(Iteration 1 / 2450) loss: 2.302605
(Epoch 0 / 10) train acc: 0.091000; val_acc: 0.119000
(Iteration 101 / 2450) loss: 1.533029
(Iteration 201 / 2450) loss: 1.404632
(Epoch 1 / 10) train acc: 0.523000; val_acc: 0.501000
(Iteration 301 / 2450) loss: 1.334484
(Iteration 401 / 2450) loss: 1.380794
(Epoch 2 / 10) train acc: 0.564000; val_acc: 0.536000
(Iteration 501 / 2450) loss: 1.251806
(Iteration 601 / 2450) loss: 1.142711
(Iteration 701 / 2450) loss: 1.090651
(Epoch 3 / 10) train acc: 0.555000; val_acc: 0.555000
(Iteration 801 / 2450) loss: 0.999355
(Iteration 901 / 2450) loss: 1.168978
(Epoch 4 / 10) train acc: 0.598000; val_acc: 0.568000
(Iteration 1001 / 2450) loss: 1.169895
(Iteration 1101 / 2450) loss: 1.123607
(Iteration 1201 / 2450) loss: 1.089484
(Epoch 5 / 10) train acc: 0.638000; val_acc: 0.572000
(Iteration 1301 / 2450) loss: 1.096008
(Iteration 1401 / 2450) loss: 0.943777
(Epoch 6 / 10) train acc: 0.646000; val_acc: 0.584000
(Iteration 1501 / 2450) loss: 0.855380
(Iteration 1601 / 2450) loss: 1.077101
(Iteration 1701 / 2450) loss: 0.999921
(Epoch 7 / 10) train acc: 0.687000; val_acc: 0.589000
(Iteration 1801 / 2450) loss: 0.882398
(Iteration 1901 / 2450) loss: 0.897782
```

```
(Epoch 8 / 10) train acc: 0.700000; val_acc: 0.596000
(Iteration 2001 / 2450) loss: 0.850920
(Iteration 2101 / 2450) loss: 0.835670
(Iteration 2201 / 2450) loss: 0.835669
(Epoch 9 / 10) train acc: 0.725000; val_acc: 0.603000
(Iteration 2301 / 2450) loss: 0.998595
(Iteration 2401 / 2450) loss: 0.909618
(Epoch 10 / 10) train acc: 0.736000; val_acc: 0.595000
(Iteration 1 / 2450) loss: 2.302585
(Epoch 0 / 10) train acc: 0.091000; val_acc: 0.078000
(Iteration 101 / 2450) loss: 1.587609
(Iteration 201 / 2450) loss: 1.269826
(Epoch 1 / 10) train acc: 0.516000; val_acc: 0.502000
(Iteration 301 / 2450) loss: 1.333190
(Iteration 401 / 2450) loss: 1.299607
(Epoch 2 / 10) train acc: 0.530000; val_acc: 0.525000
(Iteration 501 / 2450) loss: 1.192492
(Iteration 601 / 2450) loss: 1.195583
(Iteration 701 / 2450) loss: 1.400596
(Epoch 3 / 10) train acc: 0.564000; val_acc: 0.561000
(Iteration 801 / 2450) loss: 1.246339
(Iteration 901 / 2450) loss: 1.153800
(Epoch 4 / 10) train acc: 0.609000; val_acc: 0.570000
(Iteration 1001 / 2450) loss: 1.144518
(Iteration 1101 / 2450) loss: 0.995692
(Iteration 1201 / 2450) loss: 1.039465
(Epoch 5 / 10) train acc: 0.608000; val_acc: 0.579000
(Iteration 1301 / 2450) loss: 0.965568
(Iteration 1401 / 2450) loss: 1.135085
(Epoch 6 / 10) train acc: 0.634000; val_acc: 0.594000
(Iteration 1501 / 2450) loss: 0.993467
(Iteration 1601 / 2450) loss: 0.990534
(Iteration 1701 / 2450) loss: 0.848240
(Epoch 7 / 10) train acc: 0.686000; val_acc: 0.599000
(Iteration 1801 / 2450) loss: 0.852374
(Iteration 1901 / 2450) loss: 0.893288
(Epoch 8 / 10) train acc: 0.690000; val_acc: 0.606000
(Iteration 2001 / 2450) loss: 1.101195
(Iteration 2101 / 2450) loss: 0.962969
(Iteration 2201 / 2450) loss: 0.839590
(Epoch 9 / 10) train acc: 0.657000; val_acc: 0.599000
(Iteration 2301 / 2450) loss: 0.936787
(Iteration 2401 / 2450) loss: 1.004423
(Epoch 10 / 10) train acc: 0.678000; val_acc: 0.602000
(Iteration 1 / 2450) loss: 2.302551
(Epoch 0 / 10) train acc: 0.106000; val_acc: 0.112000
(Iteration 101 / 2450) loss: 1.524914
(Iteration 201 / 2450) loss: 1.420729
```

```
(Epoch 1 / 10) train acc: 0.527000; val_acc: 0.504000
(Iteration 301 / 2450) loss: 1.352284
(Iteration 401 / 2450) loss: 1.317297
(Epoch 2 / 10) train acc: 0.545000; val_acc: 0.530000
(Iteration 501 / 2450) loss: 1.253532
(Iteration 601 / 2450) loss: 1.261289
(Iteration 701 / 2450) loss: 1.095323
(Epoch 3 / 10) train acc: 0.577000; val_acc: 0.570000
(Iteration 801 / 2450) loss: 1.076630
(Iteration 901 / 2450) loss: 1.128275
(Epoch 4 / 10) train acc: 0.590000; val_acc: 0.548000
(Iteration 1001 / 2450) loss: 0.977185
(Iteration 1101 / 2450) loss: 1.158304
(Iteration 1201 / 2450) loss: 1.140002
(Epoch 5 / 10) train acc: 0.647000; val_acc: 0.583000
(Iteration 1301 / 2450) loss: 1.006819
(Iteration 1401 / 2450) loss: 1.096472
(Epoch 6 / 10) train acc: 0.661000; val_acc: 0.586000
(Iteration 1501 / 2450) loss: 0.953993
(Iteration 1601 / 2450) loss: 0.920940
(Iteration 1701 / 2450) loss: 0.895471
(Epoch 7 / 10) train acc: 0.661000; val_acc: 0.589000
(Iteration 1801 / 2450) loss: 0.865667
(Iteration 1901 / 2450) loss: 0.934535
(Epoch 8 / 10) train acc: 0.685000; val_acc: 0.594000
(Iteration 2001 / 2450) loss: 0.910644
(Iteration 2101 / 2450) loss: 0.968933
(Iteration 2201 / 2450) loss: 0.840918
(Epoch 9 / 10) train acc: 0.719000; val_acc: 0.604000
(Iteration 2301 / 2450) loss: 0.908334
(Iteration 2401 / 2450) loss: 0.885867
(Epoch 10 / 10) train acc: 0.729000; val_acc: 0.594000
(Iteration 1 / 2450) loss: 2.302550
(Epoch 0 / 10) train acc: 0.110000; val_acc: 0.087000
(Iteration 101 / 2450) loss: 1.625950
(Iteration 201 / 2450) loss: 1.467520
(Epoch 1 / 10) train acc: 0.496000; val_acc: 0.512000
(Iteration 301 / 2450) loss: 1.290050
(Iteration 401 / 2450) loss: 1.263553
(Epoch 2 / 10) train acc: 0.523000; val_acc: 0.530000
(Iteration 501 / 2450) loss: 1.180466
(Iteration 601 / 2450) loss: 1.109897
(Iteration 701 / 2450) loss: 1.145063
(Epoch 3 / 10) train acc: 0.598000; val_acc: 0.551000
(Iteration 801 / 2450) loss: 1.115838
(Iteration 901 / 2450) loss: 1.091195
(Epoch 4 / 10) train acc: 0.613000; val_acc: 0.578000
(Iteration 1001 / 2450) loss: 1.178404
```

```
(Iteration 1101 / 2450) loss: 1.279921
(Iteration 1201 / 2450) loss: 1.153007
(Epoch 5 / 10) train acc: 0.636000; val_acc: 0.572000
(Iteration 1301 / 2450) loss: 1.099594
(Iteration 1401 / 2450) loss: 1.013276
(Epoch 6 / 10) train acc: 0.694000; val_acc: 0.600000
(Iteration 1501 / 2450) loss: 0.940045
(Iteration 1601 / 2450) loss: 0.919670
(Iteration 1701 / 2450) loss: 0.855731
(Epoch 7 / 10) train acc: 0.683000; val_acc: 0.588000
(Iteration 1801 / 2450) loss: 0.885415
(Iteration 1901 / 2450) loss: 0.830475
(Epoch 8 / 10) train acc: 0.679000; val_acc: 0.594000
(Iteration 2001 / 2450) loss: 0.925012
(Iteration 2101 / 2450) loss: 0.849628
(Iteration 2201 / 2450) loss: 0.968469
(Epoch 9 / 10) train acc: 0.720000; val_acc: 0.578000
(Iteration 2301 / 2450) loss: 0.747593
(Iteration 2401 / 2450) loss: 0.614179
(Epoch 10 / 10) train acc: 0.753000; val_acc: 0.580000
(Iteration 1 / 2450) loss: 2.302604
(Epoch 0 / 10) train acc: 0.094000; val_acc: 0.087000
(Iteration 101 / 2450) loss: 1.500925
(Iteration 201 / 2450) loss: 1.357090
(Epoch 1 / 10) train acc: 0.531000; val_acc: 0.498000
(Iteration 301 / 2450) loss: 1.312334
(Iteration 401 / 2450) loss: 1.354219
(Epoch 2 / 10) train acc: 0.538000; val_acc: 0.549000
(Iteration 501 / 2450) loss: 1.132834
(Iteration 601 / 2450) loss: 1.249365
(Iteration 701 / 2450) loss: 1.180626
(Epoch 3 / 10) train acc: 0.593000; val_acc: 0.544000
(Iteration 801 / 2450) loss: 1.212617
(Iteration 901 / 2450) loss: 1.172867
(Epoch 4 / 10) train acc: 0.633000; val_acc: 0.560000
(Iteration 1001 / 2450) loss: 0.990172
(Iteration 1101 / 2450) loss: 1.055831
(Iteration 1201 / 2450) loss: 0.977004
(Epoch 5 / 10) train acc: 0.665000; val_acc: 0.574000
(Iteration 1301 / 2450) loss: 1.095596
(Iteration 1401 / 2450) loss: 0.879300
(Epoch 6 / 10) train acc: 0.653000; val_acc: 0.596000
(Iteration 1501 / 2450) loss: 0.935463
(Iteration 1601 / 2450) loss: 0.980505
(Iteration 1701 / 2450) loss: 0.952244
(Epoch 7 / 10) train acc: 0.691000; val_acc: 0.584000
(Iteration 1801 / 2450) loss: 0.957604
(Iteration 1901 / 2450) loss: 1.039959
```

```
(Epoch 8 / 10) train acc: 0.715000; val_acc: 0.584000
(Iteration 2001 / 2450) loss: 0.853465
(Iteration 2101 / 2450) loss: 0.808301
(Iteration 2201 / 2450) loss: 0.805367
(Epoch 9 / 10) train acc: 0.723000; val_acc: 0.588000
(Iteration 2301 / 2450) loss: 0.850611
(Iteration 2401 / 2450) loss: 0.998346
(Epoch 10 / 10) train acc: 0.732000; val_acc: 0.594000
(Iteration 1 / 2450) loss: 2.302578
(Epoch 0 / 10) train acc: 0.103000; val_acc: 0.078000
(Iteration 101 / 2450) loss: 1.526807
(Iteration 201 / 2450) loss: 1.532871
(Epoch 1 / 10) train acc: 0.519000; val_acc: 0.495000
(Iteration 301 / 2450) loss: 1.435658
(Iteration 401 / 2450) loss: 1.187896
(Epoch 2 / 10) train acc: 0.576000; val_acc: 0.540000
(Iteration 501 / 2450) loss: 1.107614
(Iteration 601 / 2450) loss: 1.238281
(Iteration 701 / 2450) loss: 1.165474
(Epoch 3 / 10) train acc: 0.610000; val_acc: 0.547000
(Iteration 801 / 2450) loss: 1.141910
(Iteration 901 / 2450) loss: 1.094827
(Epoch 4 / 10) train acc: 0.615000; val_acc: 0.570000
(Iteration 1001 / 2450) loss: 1.116869
(Iteration 1101 / 2450) loss: 0.965683
(Iteration 1201 / 2450) loss: 0.993838
(Epoch 5 / 10) train acc: 0.672000; val_acc: 0.570000
(Iteration 1301 / 2450) loss: 1.084146
(Iteration 1401 / 2450) loss: 0.915160
(Epoch 6 / 10) train acc: 0.662000; val_acc: 0.571000
(Iteration 1501 / 2450) loss: 0.873889
(Iteration 1601 / 2450) loss: 0.860925
(Iteration 1701 / 2450) loss: 0.922919
(Epoch 7 / 10) train acc: 0.711000; val_acc: 0.603000
(Iteration 1801 / 2450) loss: 0.792171
(Iteration 1901 / 2450) loss: 0.866988
(Epoch 8 / 10) train acc: 0.723000; val_acc: 0.592000
(Iteration 2001 / 2450) loss: 0.834041
(Iteration 2101 / 2450) loss: 0.829692
(Iteration 2201 / 2450) loss: 0.800132
(Epoch 9 / 10) train acc: 0.736000; val_acc: 0.604000
(Iteration 2301 / 2450) loss: 0.770359
(Iteration 2401 / 2450) loss: 0.674174
(Epoch 10 / 10) train acc: 0.734000; val_acc: 0.594000
(Iteration 1 / 2450) loss: 2.302596
(Epoch 0 / 10) train acc: 0.092000; val_acc: 0.119000
(Iteration 101 / 2450) loss: 1.508109
(Iteration 201 / 2450) loss: 1.322627
```

```
(Epoch 1 / 10) train acc: 0.487000; val_acc: 0.512000
(Iteration 301 / 2450) loss: 1.420753
(Iteration 401 / 2450) loss: 1.295057
(Epoch 2 / 10) train acc: 0.555000; val_acc: 0.518000
(Iteration 501 / 2450) loss: 1.229694
(Iteration 601 / 2450) loss: 1.196163
(Iteration 701 / 2450) loss: 1.084944
(Epoch 3 / 10) train acc: 0.607000; val_acc: 0.547000
(Iteration 801 / 2450) loss: 0.977724
(Iteration 901 / 2450) loss: 1.101807
(Epoch 4 / 10) train acc: 0.621000; val_acc: 0.545000
(Iteration 1001 / 2450) loss: 1.075862
(Iteration 1101 / 2450) loss: 1.014233
(Iteration 1201 / 2450) loss: 0.936870
(Epoch 5 / 10) train acc: 0.655000; val_acc: 0.584000
(Iteration 1301 / 2450) loss: 0.997261
(Iteration 1401 / 2450) loss: 1.033844
(Epoch 6 / 10) train acc: 0.657000; val_acc: 0.589000
(Iteration 1501 / 2450) loss: 0.868964
(Iteration 1601 / 2450) loss: 0.937498
(Iteration 1701 / 2450) loss: 0.846969
(Epoch 7 / 10) train acc: 0.677000; val_acc: 0.592000
(Iteration 1801 / 2450) loss: 0.787612
(Iteration 1901 / 2450) loss: 0.723087
(Epoch 8 / 10) train acc: 0.681000; val_acc: 0.587000
(Iteration 2001 / 2450) loss: 0.788432
(Iteration 2101 / 2450) loss: 0.850356
(Iteration 2201 / 2450) loss: 0.794506
(Epoch 9 / 10) train acc: 0.734000; val_acc: 0.593000
(Iteration 2301 / 2450) loss: 0.817185
(Iteration 2401 / 2450) loss: 0.938676
(Epoch 10 / 10) train acc: 0.733000; val_acc: 0.580000
(Iteration 1 / 2450) loss: 2.302606
(Epoch 0 / 10) train acc: 0.098000; val_acc: 0.098000
(Iteration 101 / 2450) loss: 1.386668
(Iteration 201 / 2450) loss: 1.322978
(Epoch 1 / 10) train acc: 0.542000; val_acc: 0.525000
(Iteration 301 / 2450) loss: 1.277339
(Iteration 401 / 2450) loss: 1.297927
(Epoch 2 / 10) train acc: 0.593000; val_acc: 0.543000
(Iteration 501 / 2450) loss: 1.214614
(Iteration 601 / 2450) loss: 1.331624
(Iteration 701 / 2450) loss: 1.124822
(Epoch 3 / 10) train acc: 0.599000; val_acc: 0.552000
(Iteration 801 / 2450) loss: 1.115154
(Iteration 901 / 2450) loss: 1.005493
(Epoch 4 / 10) train acc: 0.628000; val_acc: 0.575000
(Iteration 1001 / 2450) loss: 1.160101
```

```
(Iteration 1101 / 2450) loss: 0.943158
(Iteration 1201 / 2450) loss: 0.976827
(Epoch 5 / 10) train acc: 0.677000; val_acc: 0.578000
(Iteration 1301 / 2450) loss: 0.974030
(Iteration 1401 / 2450) loss: 1.011371
(Epoch 6 / 10) train acc: 0.677000; val_acc: 0.582000
(Iteration 1501 / 2450) loss: 0.973511
(Iteration 1601 / 2450) loss: 0.902096
(Iteration 1701 / 2450) loss: 0.933765
(Epoch 7 / 10) train acc: 0.661000; val_acc: 0.587000
(Iteration 1801 / 2450) loss: 0.840129
(Iteration 1901 / 2450) loss: 1.021712
(Epoch 8 / 10) train acc: 0.710000; val_acc: 0.586000
(Iteration 2001 / 2450) loss: 0.797776
(Iteration 2101 / 2450) loss: 0.853179
(Iteration 2201 / 2450) loss: 0.881909
(Epoch 9 / 10) train acc: 0.718000; val_acc: 0.590000
(Iteration 2301 / 2450) loss: 0.744160
(Iteration 2401 / 2450) loss: 0.853461
(Epoch 10 / 10) train acc: 0.733000; val_acc: 0.604000
(Iteration 1 / 2450) loss: 2.302577
(Epoch 0 / 10) train acc: 0.101000; val_acc: 0.079000
(Iteration 101 / 2450) loss: 1.471542
(Iteration 201 / 2450) loss: 1.373571
(Epoch 1 / 10) train acc: 0.522000; val_acc: 0.498000
(Iteration 301 / 2450) loss: 1.451223
(Iteration 401 / 2450) loss: 1.273223
(Epoch 2 / 10) train acc: 0.556000; val_acc: 0.546000
(Iteration 501 / 2450) loss: 1.383373
(Iteration 601 / 2450) loss: 1.293620
(Iteration 701 / 2450) loss: 1.236315
(Epoch 3 / 10) train acc: 0.581000; val_acc: 0.562000
(Iteration 801 / 2450) loss: 1.089092
(Iteration 901 / 2450) loss: 1.093534
(Epoch 4 / 10) train acc: 0.630000; val_acc: 0.567000
(Iteration 1001 / 2450) loss: 1.119122
(Iteration 1101 / 2450) loss: 1.083621
(Iteration 1201 / 2450) loss: 0.986892
(Epoch 5 / 10) train acc: 0.658000; val_acc: 0.579000
(Iteration 1301 / 2450) loss: 1.068523
(Iteration 1401 / 2450) loss: 1.024273
(Epoch 6 / 10) train acc: 0.675000; val_acc: 0.577000
(Iteration 1501 / 2450) loss: 0.931280
(Iteration 1601 / 2450) loss: 0.908827
(Iteration 1701 / 2450) loss: 0.813058
(Epoch 7 / 10) train acc: 0.683000; val_acc: 0.581000
(Iteration 1801 / 2450) loss: 0.947263
(Iteration 1901 / 2450) loss: 0.908694
```

```
(Epoch 8 / 10) train acc: 0.707000; val_acc: 0.599000
(Iteration 2001 / 2450) loss: 0.829609
(Iteration 2101 / 2450) loss: 0.742917
(Iteration 2201 / 2450) loss: 0.891543
(Epoch 9 / 10) train acc: 0.701000; val_acc: 0.590000
(Iteration 2301 / 2450) loss: 0.813059
(Iteration 2401 / 2450) loss: 0.774235
(Epoch 10 / 10) train acc: 0.738000; val_acc: 0.586000
(Iteration 1 / 2450) loss: 2.302563
(Epoch 0 / 10) train acc: 0.105000; val_acc: 0.112000
(Iteration 101 / 2450) loss: 1.636822
(Iteration 201 / 2450) loss: 1.356668
(Epoch 1 / 10) train acc: 0.523000; val_acc: 0.501000
(Iteration 301 / 2450) loss: 1.322604
(Iteration 401 / 2450) loss: 1.175570
(Epoch 2 / 10) train acc: 0.527000; val_acc: 0.540000
(Iteration 501 / 2450) loss: 1.324523
(Iteration 601 / 2450) loss: 1.137212
(Iteration 701 / 2450) loss: 1.301822
(Epoch 3 / 10) train acc: 0.567000; val_acc: 0.553000
(Iteration 801 / 2450) loss: 1.110391
(Iteration 901 / 2450) loss: 0.909644
(Epoch 4 / 10) train acc: 0.628000; val_acc: 0.556000
(Iteration 1001 / 2450) loss: 1.006448
(Iteration 1101 / 2450) loss: 1.009221
(Iteration 1201 / 2450) loss: 1.053628
(Epoch 5 / 10) train acc: 0.668000; val_acc: 0.569000
(Iteration 1301 / 2450) loss: 0.980648
(Iteration 1401 / 2450) loss: 0.933525
(Epoch 6 / 10) train acc: 0.655000; val_acc: 0.591000
(Iteration 1501 / 2450) loss: 0.829797
(Iteration 1601 / 2450) loss: 1.022829
(Iteration 1701 / 2450) loss: 0.855453
(Epoch 7 / 10) train acc: 0.709000; val_acc: 0.584000
(Iteration 1801 / 2450) loss: 0.865858
(Iteration 1901 / 2450) loss: 0.844155
(Epoch 8 / 10) train acc: 0.705000; val_acc: 0.577000
(Iteration 2001 / 2450) loss: 0.911727
(Iteration 2101 / 2450) loss: 0.863176
(Iteration 2201 / 2450) loss: 0.884527
(Epoch 9 / 10) train acc: 0.728000; val_acc: 0.587000
(Iteration 2301 / 2450) loss: 0.858537
(Iteration 2401 / 2450) loss: 0.711779
(Epoch 10 / 10) train acc: 0.740000; val_acc: 0.572000
(Iteration 1 / 2450) loss: 2.302539
(Epoch 0 / 10) train acc: 0.096000; val_acc: 0.094000
(Iteration 101 / 2450) loss: 1.481474
(Iteration 201 / 2450) loss: 1.485028
```

```
(Epoch 1 / 10) train acc: 0.524000; val_acc: 0.509000
(Iteration 301 / 2450) loss: 1.356218
(Iteration 401 / 2450) loss: 1.212160
(Epoch 2 / 10) train acc: 0.579000; val_acc: 0.546000
(Iteration 501 / 2450) loss: 1.145868
(Iteration 601 / 2450) loss: 1.135062
(Iteration 701 / 2450) loss: 1.219885
(Epoch 3 / 10) train acc: 0.586000; val_acc: 0.555000
(Iteration 801 / 2450) loss: 1.091372
(Iteration 901 / 2450) loss: 0.960972
(Epoch 4 / 10) train acc: 0.625000; val_acc: 0.578000
(Iteration 1001 / 2450) loss: 1.060929
(Iteration 1101 / 2450) loss: 0.990925
(Iteration 1201 / 2450) loss: 0.999602
(Epoch 5 / 10) train acc: 0.659000; val_acc: 0.588000
(Iteration 1301 / 2450) loss: 1.039946
(Iteration 1401 / 2450) loss: 0.904014
(Epoch 6 / 10) train acc: 0.678000; val_acc: 0.588000
(Iteration 1501 / 2450) loss: 0.916503
(Iteration 1601 / 2450) loss: 0.922264
(Iteration 1701 / 2450) loss: 0.984847
(Epoch 7 / 10) train acc: 0.682000; val_acc: 0.601000
(Iteration 1801 / 2450) loss: 0.970635
(Iteration 1901 / 2450) loss: 0.943539
(Epoch 8 / 10) train acc: 0.701000; val_acc: 0.588000
(Iteration 2001 / 2450) loss: 0.721101
(Iteration 2101 / 2450) loss: 0.905975
(Iteration 2201 / 2450) loss: 0.644092
(Epoch 9 / 10) train acc: 0.708000; val_acc: 0.608000
(Iteration 2301 / 2450) loss: 0.844278
(Iteration 2401 / 2450) loss: 0.820968
(Epoch 10 / 10) train acc: 0.728000; val_acc: 0.605000
(Iteration 1 / 2450) loss: 2.302607
(Epoch 0 / 10) train acc: 0.144000; val_acc: 0.134000
(Iteration 101 / 2450) loss: 1.484712
(Iteration 201 / 2450) loss: 1.437679
(Epoch 1 / 10) train acc: 0.537000; val_acc: 0.527000
(Iteration 301 / 2450) loss: 1.284883
(Iteration 401 / 2450) loss: 1.200903
(Epoch 2 / 10) train acc: 0.586000; val_acc: 0.534000
(Iteration 501 / 2450) loss: 1.149950
(Iteration 601 / 2450) loss: 1.142064
(Iteration 701 / 2450) loss: 1.169498
(Epoch 3 / 10) train acc: 0.552000; val_acc: 0.553000
(Iteration 801 / 2450) loss: 1.188351
(Iteration 901 / 2450) loss: 1.168420
(Epoch 4 / 10) train acc: 0.602000; val_acc: 0.580000
(Iteration 1001 / 2450) loss: 1.084910
```

```
(Iteration 1101 / 2450) loss: 1.243510
(Iteration 1201 / 2450) loss: 1.036811
(Epoch 5 / 10) train acc: 0.642000; val_acc: 0.547000
(Iteration 1301 / 2450) loss: 1.020346
(Iteration 1401 / 2450) loss: 1.088137
(Epoch 6 / 10) train acc: 0.693000; val_acc: 0.592000
(Iteration 1501 / 2450) loss: 0.889637
(Iteration 1601 / 2450) loss: 1.004013
(Iteration 1701 / 2450) loss: 0.913606
(Epoch 7 / 10) train acc: 0.691000; val_acc: 0.590000
(Iteration 1801 / 2450) loss: 0.906178
(Iteration 1901 / 2450) loss: 0.866577
(Epoch 8 / 10) train acc: 0.730000; val_acc: 0.589000
(Iteration 2001 / 2450) loss: 0.885654
(Iteration 2101 / 2450) loss: 0.734707
(Iteration 2201 / 2450) loss: 0.818790
(Epoch 9 / 10) train acc: 0.729000; val_acc: 0.569000
(Iteration 2301 / 2450) loss: 0.757231
(Iteration 2401 / 2450) loss: 0.844083
(Epoch 10 / 10) train acc: 0.767000; val_acc: 0.580000
(Iteration 1 / 2450) loss: 2.302606
(Epoch 0 / 10) train acc: 0.100000; val_acc: 0.092000
(Iteration 101 / 2450) loss: 1.480533
(Iteration 201 / 2450) loss: 1.327906
(Epoch 1 / 10) train acc: 0.549000; val_acc: 0.504000
(Iteration 301 / 2450) loss: 1.364089
(Iteration 401 / 2450) loss: 1.303405
(Epoch 2 / 10) train acc: 0.554000; val_acc: 0.531000
(Iteration 501 / 2450) loss: 1.284494
(Iteration 601 / 2450) loss: 1.339019
(Iteration 701 / 2450) loss: 1.146597
(Epoch 3 / 10) train acc: 0.579000; val_acc: 0.552000
(Iteration 801 / 2450) loss: 1.106643
(Iteration 901 / 2450) loss: 1.180601
(Epoch 4 / 10) train acc: 0.628000; val_acc: 0.572000
(Iteration 1001 / 2450) loss: 1.097405
(Iteration 1101 / 2450) loss: 1.017957
(Iteration 1201 / 2450) loss: 1.028098
(Epoch 5 / 10) train acc: 0.634000; val_acc: 0.578000
(Iteration 1301 / 2450) loss: 1.083824
(Iteration 1401 / 2450) loss: 1.000636
(Epoch 6 / 10) train acc: 0.697000; val_acc: 0.573000
(Iteration 1501 / 2450) loss: 1.013754
(Iteration 1601 / 2450) loss: 0.813524
(Iteration 1701 / 2450) loss: 0.926185
(Epoch 7 / 10) train acc: 0.674000; val_acc: 0.558000
(Iteration 1801 / 2450) loss: 0.928624
(Iteration 1901 / 2450) loss: 0.926328
```

```
(Epoch 8 / 10) train acc: 0.718000; val_acc: 0.573000
(Iteration 2001 / 2450) loss: 0.808590
(Iteration 2101 / 2450) loss: 0.784206
(Iteration 2201 / 2450) loss: 0.869854
(Epoch 9 / 10) train acc: 0.766000; val_acc: 0.576000
(Iteration 2301 / 2450) loss: 0.664075
(Iteration 2401 / 2450) loss: 0.729566
(Epoch 10 / 10) train acc: 0.734000; val_acc: 0.575000
Best validation accuracy achieved during cross-validation: 0.623000
```

[13]:
```python
# Run your best neural net classifier on the test set. You should be able
# to get more than 55% accuracy.

# Run your best neural net classifier on the test set.
scores = best_net.loss(X_test_feats)
y_test_pred = np.argmax(scores, axis=1)
test_acc = np.mean(y_test_pred == y_test)

# Print the test accuracy
print('Test accuracy: %f' % test_acc)
```

```
Test accuracy: 0.598000
```

[ ]:
```
```