knn

October 9, 2022

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

# Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment1/'
FOLDERNAME = 'ENPM809K/assignment1/assignment1/'
assert FOLDERNAME is not None, "[!] Enter the foldername."

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

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

```
Mounted at /content/drive
/content/drive/My Drive/ENPM809K/assignment1/assignment1/cs231n/datasets
/content/drive/My Drive/ENPM809K/assignment1/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:

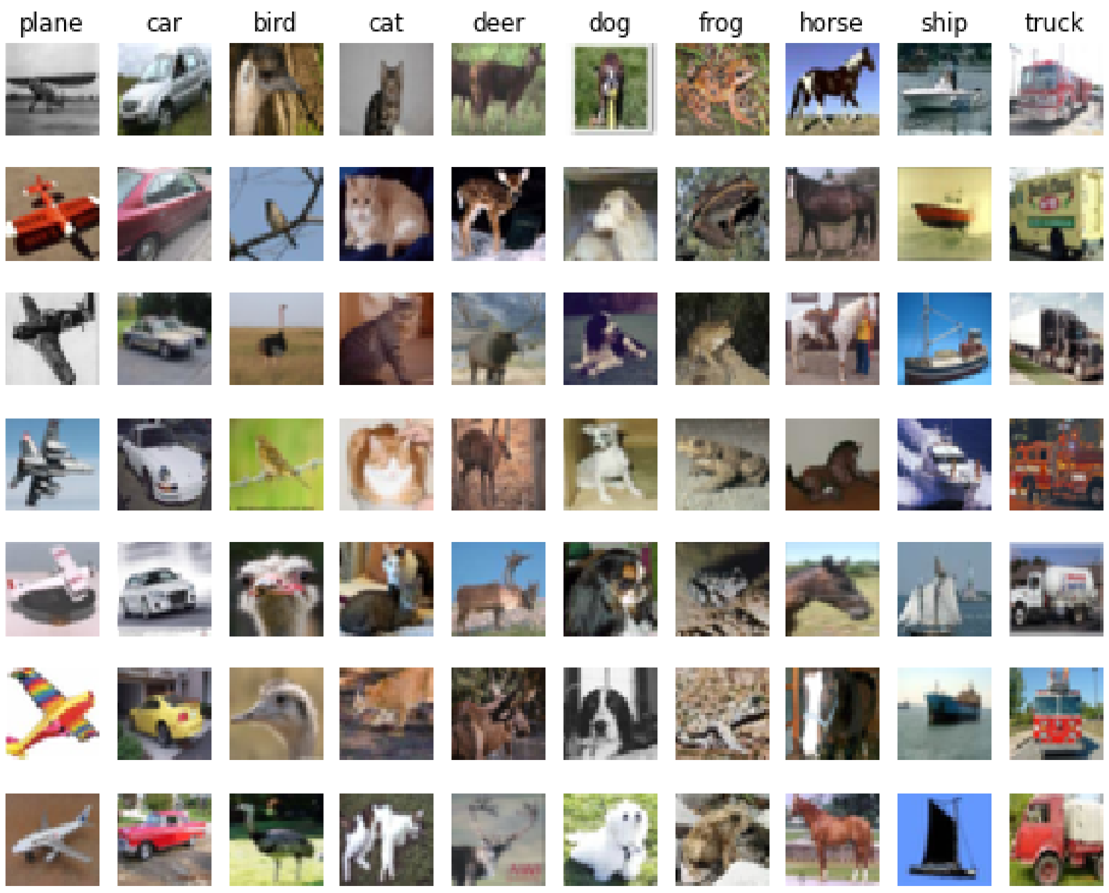
- During training, the classifier takes the training data and simply remembers it
- During testing, kNN classifies every test image by comparing to all training images and transfering the labels of the k most similar training examples
- The value of k is cross-validated

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.

[ ]:
```
# 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
```

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

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

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



~~~

[ ]:
```python
# 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)
```
~~~

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

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

[ ]:
```python
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

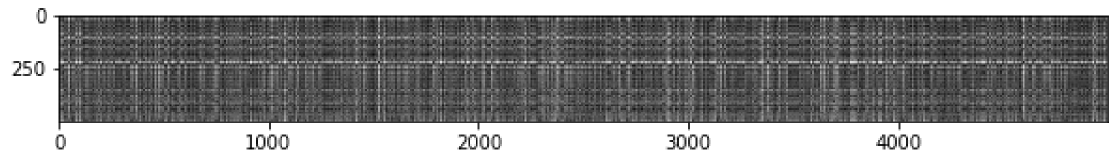
Lets begin with computing the distance matrix between all training and test examples. For example, if there are **Ntr** training examples and **Nte** test examples, this stage should result in a **Nte x Ntr** matrix where each element (i,j) is the distance between the i-th test and j-th train example.

**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.

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

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

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


0
250
0
1000
2000
3000
4000

```
~~~
```

Inline Question 1

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

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

YourAnswer : *1.Either this is a test image which is not close to any class in the training data set, or is way different from most of the training data. 2. This training data point doesn't have any similar points in the test set.*

[ ]:
```
# 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`:

```
[ ]: # 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 μ across all pixels over all images is

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

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

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

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

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

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

$\color{blue}{\textit{Your Explanation:}}$ In option 1,2 the L1 distance is preserved under subraction of means. It is same as offsetting the data around the origin. For option 3, subracting mean and dividing by stardard deviation is scaling(normalisation) of data which proves to be useful when data is vast and lies in various different ranges. However, 4th doesnt work because we are subracting different value for each dimension of the training data, we are scaling differently. Thus it will affect the performance of the classifier.Rotations transforms points differently which will cause the distance between the same
~~~

three points to transform. Due to these disparities, this transformation will hange the performance of the knn classifier.

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

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

[ ]:
```
# 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 128.578891 seconds
One loop version took 108.025254 seconds
No loop version took 0.620642 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.
~~~

[ ]:
```
# 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)*****
#using the array_split function, we have split the training sets into 5 folds
X_train_folds = np.array_split(X_train, num_folds)
y_train_folds = np.array_split(y_train, num_folds)


# *****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)*****
#we make use of two loops, outer one to loop over different 'k' values whereas␣
 ↪the
#inner loop for shuffling all the folds betweent validation and train sets
for k in k_choices:
  k_to_accuracies[k] = []
  for fold in range(num_folds):
    #setting up our validation sets
    X_validation=X_train_folds[fold]
    y_validation=y_train_folds[fold]
    #setting train data by concatinating folds before the current selected fold
    #and folds after the selected fold
    X_train_data= np.concatenate(X_train_folds[:fold]+X_train_folds[fold+1:])
    y_train_data= np.concatenate(y_train_folds[:fold]+y_train_folds[fold+1:])
    #start classifier training on above set training data
    classifier.train(X_train_data, y_train_data)
    #calculate distance of the validation fold
    dists_validation = classifier.compute_distances_no_loops(X_validation)
    #predict lablels based on the validation distance and 'k' neighbors
    y_validation_pred = classifier.predict_labels(dists_validation, k)
    #get the score and accuracy and append to the dictionary
    num_correct = np.sum(y_validation_pred == y_validation)
    accuracy = float(num_correct) / num_test
    k_to_accuracies[k].append(accuracy)


# *****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('``````````````````````````````````````````````')
```

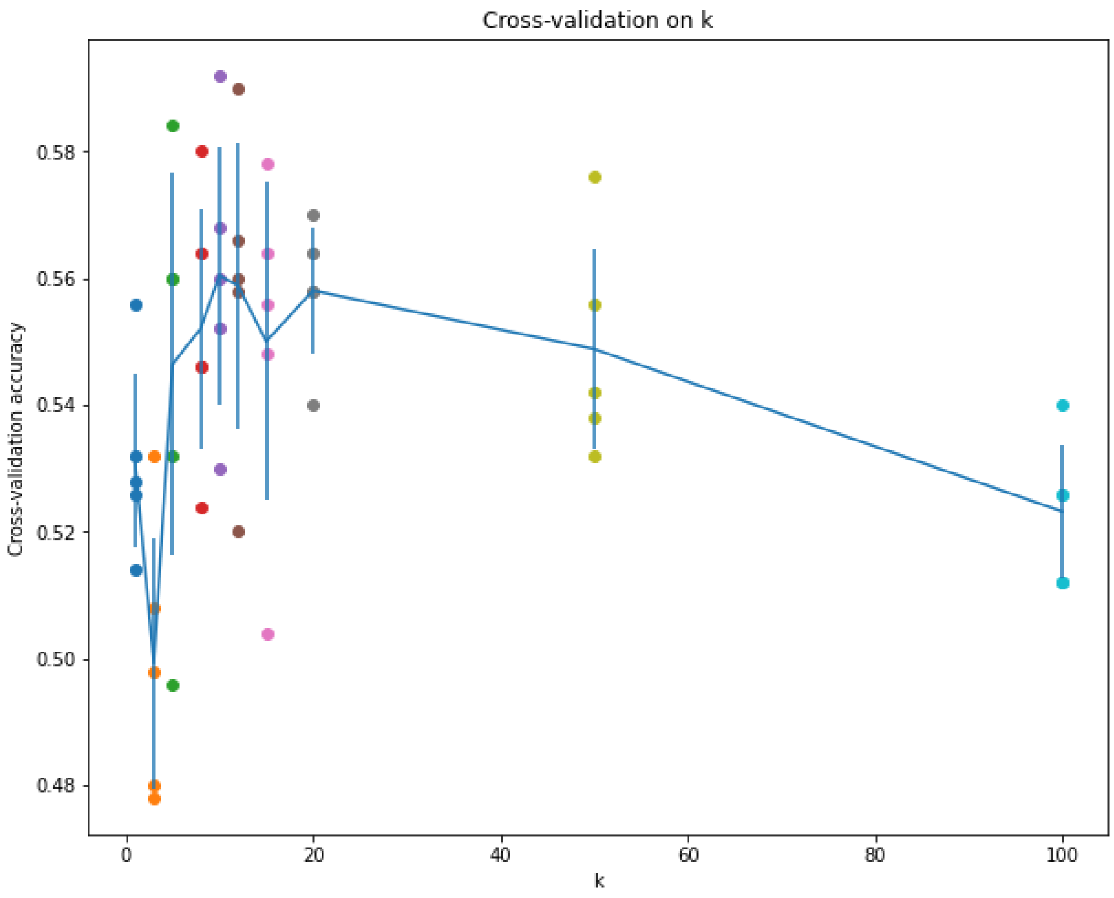
```
||||||||||||||||||||||||||||||||||||||||||||||
k = 1, accuracy = 0.526000
k = 1, accuracy = 0.514000
k = 1, accuracy = 0.528000
k = 1, accuracy = 0.556000
k = 1, accuracy = 0.532000
k = 3, accuracy = 0.478000
k = 3, accuracy = 0.498000
k = 3, accuracy = 0.480000
k = 3, accuracy = 0.532000
k = 3, accuracy = 0.508000
k = 5, accuracy = 0.496000
k = 5, accuracy = 0.532000
k = 5, accuracy = 0.560000
k = 5, accuracy = 0.584000
k = 5, accuracy = 0.560000
k = 8, accuracy = 0.524000
k = 8, accuracy = 0.564000
k = 8, accuracy = 0.546000
k = 8, accuracy = 0.580000
k = 8, accuracy = 0.546000
k = 10, accuracy = 0.530000
```

```
k = 10, accuracy = 0.592000
k = 10, accuracy = 0.552000
k = 10, accuracy = 0.568000
k = 10, accuracy = 0.560000
k = 12, accuracy = 0.520000
k = 12, accuracy = 0.590000
k = 12, accuracy = 0.558000
k = 12, accuracy = 0.566000
k = 12, accuracy = 0.560000
k = 15, accuracy = 0.504000
k = 15, accuracy = 0.578000
k = 15, accuracy = 0.556000
k = 15, accuracy = 0.564000
k = 15, accuracy = 0.548000
k = 20, accuracy = 0.540000
k = 20, accuracy = 0.558000
k = 20, accuracy = 0.558000
k = 20, accuracy = 0.564000
k = 20, accuracy = 0.570000
k = 50, accuracy = 0.542000
k = 50, accuracy = 0.576000
k = 50, accuracy = 0.556000
k = 50, accuracy = 0.538000
k = 50, accuracy = 0.532000
k = 100, accuracy = 0.512000
k = 100, accuracy = 0.540000
k = 100, accuracy = 0.526000
k = 100, accuracy = 0.512000
k = 100, accuracy = 0.526000
``````````````````````````````````````````````

[ ]:

```python
# 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('``````````````````````````````````````````````')
```

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



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

[ ]:
```
# 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.

#by observing the above plot, it is visible that 10 gives us the best accuracy
#of 28.2%
best_k = 10
classifier = KNearestNeighbor()
classifier.train(X_train, y_train)
```
```

```
y_test_pred = classifier.predict(X_test, k=best_k)

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

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

**Inline Question 3**

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

$*Your* Answer:$1,2,4

*YourExplanation* : 1.False. The decision boundary of the k-NN classifier is not linear. FOr different datasets with different classess, it produces different decision boundaries.

2.True. The training error of a 1-NN will always be lower than that of 5-NN because 1 nearest neighbor of training point is always itself, thus, error of 1-NN will be zero.

3.False. The test error of a 1-NN will not always be lower than 5-NN. This is more of a data dependent decision and the best method to choose this hyper parameter is by cross validation.

4.True. Classification of test example involves comparision with each train example. Thus, bigger the training set, larger is the amount of time required to compute this decision.
```

# svm

October 9, 2022

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

# Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment1/'
FOLDERNAME = 'ENPM809K/assignment1/assignment1/'
assert FOLDERNAME is not None, "[!] Enter the foldername."

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

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

```
Mounted at /content/drive
/content/drive/My Drive/ENPM809K/assignment1/assignment1/cs231n/datasets
/content/drive/My Drive/ENPM809K/assignment1/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

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

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

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

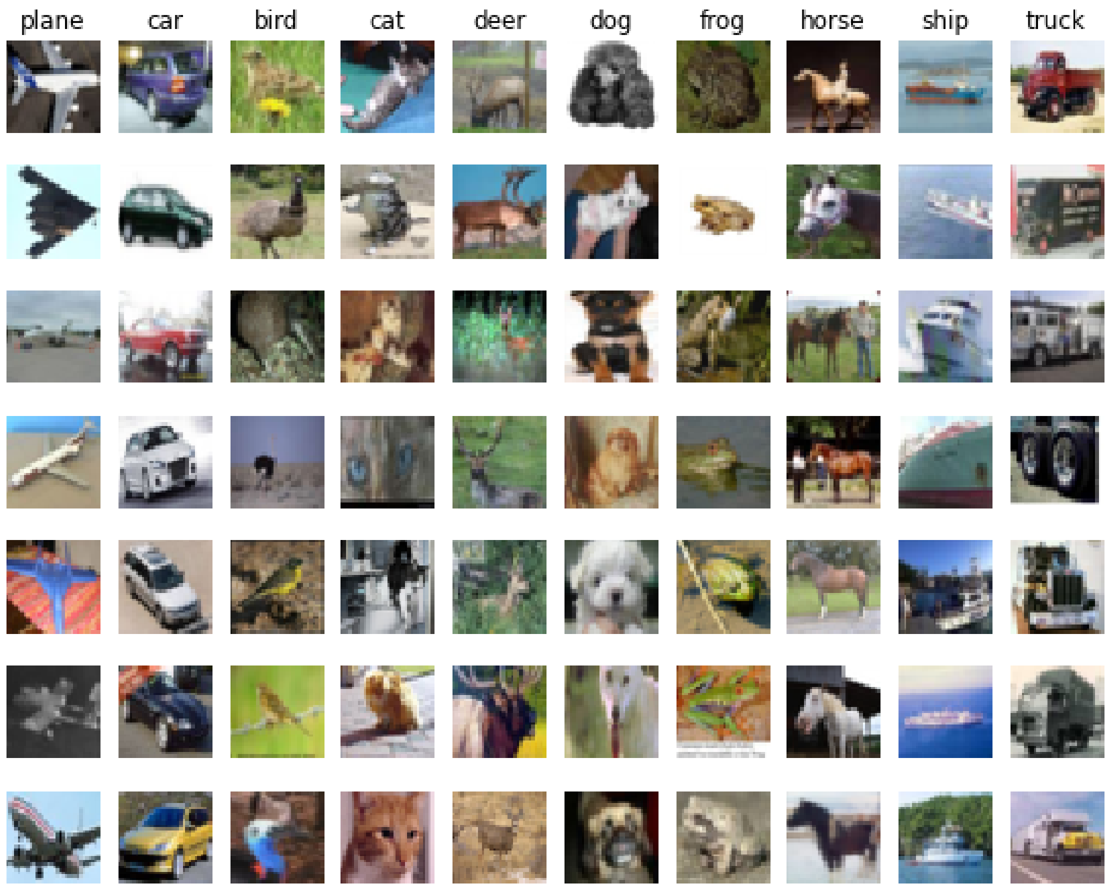
```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

## 1.1 CIFAR-10 Data Loading and Preprocessing

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

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

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

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

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

[5]:
```
# Visualize some examples from the dataset.
# We show a few examples of training images from each class.
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
 ↪'ship', 'truck']
num_classes = len(classes)
samples_per_class = 7
for y, cls in enumerate(classes):
    idxs = np.flatnonzero(y_train == y)
    idxs = np.random.choice(idxs, samples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt_idx = i * num_classes + y + 1
        plt.subplot(samples_per_class, num_classes, plt_idx)
        plt.imshow(X_train[idx].astype('uint8'))
        plt.axis('off')
        if i == 0:
            plt.title(cls)
plt.show()
```



[6]:
```
# Split the data into train, val, and test sets. In addition we will
# create a small development set as a subset of the training data;
# we can use this for development so our code runs faster.
num_training = 49000
num_validation = 1000
num_test = 1000
num_dev = 500

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

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

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

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

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

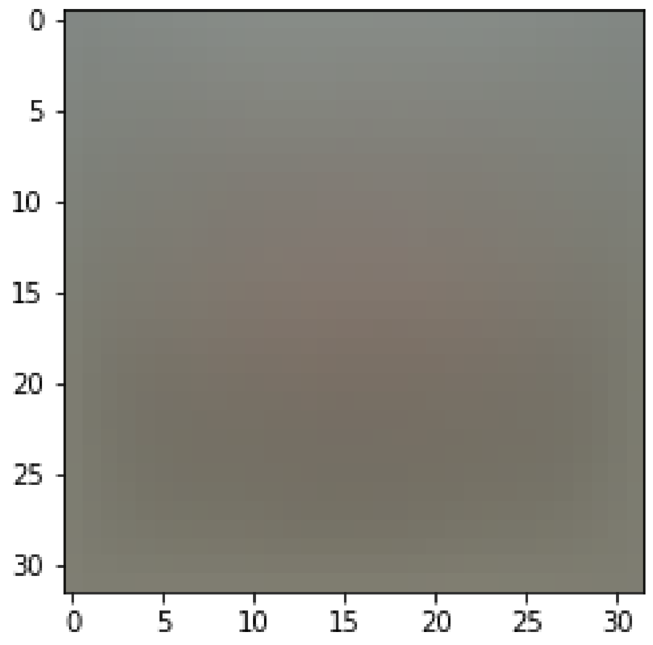
```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

[7]:
```
# Preprocessing: reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_val = np.reshape(X_val, (X_val.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

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

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

[8]:
```
# Preprocessing: subtract the mean image
# first: compute the image mean based on the training data
mean_image = np.mean(X_train, axis=0)
print(mean_image[:10]) # print a few of the elements
plt.figure(figsize=(4,4))
plt.imshow(mean_image.reshape((32,32,3)).astype('uint8')) # visualize the mean␣
 ↪image
plt.show()

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

# third: append the bias dimension of ones (i.e. bias trick) so that our 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.

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

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

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

```
loss: 9.189709
```

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:

[10]:
```
# Once you've implemented the gradient, recompute it with the code below
# and gradient check it with the function we provided for you

# Compute the loss and its gradient at W.
loss, grad = 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: 25.903085 analytic: 25.903085, relative error: 1.875157e-11
numerical: -6.792793 analytic: -6.792793, relative error: 1.849234e-11
numerical: 5.213168 analytic: 5.213168, relative error: 2.676231e-11
numerical: 14.209816 analytic: 14.209816, relative error: 2.142413e-11
numerical: 8.342600 analytic: 8.342600, relative error: 9.301674e-12
numerical: 2.916351 analytic: 2.916351, relative error: 1.168521e-10
numerical: 8.070231 analytic: 8.070231, relative error: 1.782689e-11
numerical: 7.809881 analytic: 7.809881, relative error: 9.337843e-12
numerical: -13.220084 analytic: -13.220084, relative error: 1.437496e-11
numerical: 17.491965 analytic: 17.508312, relative error: 4.670616e-04
numerical: 5.207403 analytic: 5.207403, relative error: 5.909937e-11
numerical: 9.171593 analytic: 9.171593, relative error: 1.696697e-11
numerical: 12.923375 analytic: 12.923375, relative error: 9.469138e-13
numerical: 2.557388 analytic: 2.529683, relative error: 5.446097e-03
numerical: 12.258478 analytic: 12.258478, relative error: 2.293031e-11
numerical: 4.499442 analytic: 4.519888, relative error: 2.266852e-03
numerical: -9.874880 analytic: -9.874880, relative error: 1.665254e-11
numerical: -19.944424 analytic: -19.944424, relative error: 2.116388e-11
numerical: 2.993930 analytic: 3.115212, relative error: 1.985269e-02
numerical: 9.480461 analytic: 9.480461, relative error: 5.430554e-12
```

### Inline Question 1

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

$Your\,Answer:$ * 1.Once in a while the a dimension in gradcheck may not match exactly due to the discontinuous nature of gradient of SVM loss function in some conditions. 2.It is not necessarily a reason of concern as this invalid gradient condition only takes place in the kink area and the possibility of the loss value lying in the same area/ points is highly unlikely. 3.max(0,0) is one simple example. Another example is We have a term f = max(0,x), which is not a continuous function, hence not fully ifferentiable. If x = 0.0001, then f = 0.0001 and df = 1. When calculating using gradcheck, f(x - h) can get lesser than 0, therefore, df will not be exactly. 4. INcreasing delta would increase scores, hence max( 0, function(scores)) would give more positive outputs and hence it may reduce the frequency of gradient check failing.

[11]:
```
# Next implement the function svm_loss_vectorized; for now only compute the␣
↪loss;
# we will implement the gradient in a moment.
tic = time.time()
loss_naive, grad_naive = svm_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Naive loss: %e computed in %fs' % (loss_naive, toc - tic))

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

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

```
Naive loss: 9.189709e+00 computed in 0.129900s
Vectorized loss: 9.189709e+00 computed in 0.012567s
difference: -0.000000
```

[15]:
```
# 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 0.127480s
Vectorized loss and gradient: computed in 0.012127s
difference: 0.000000
```

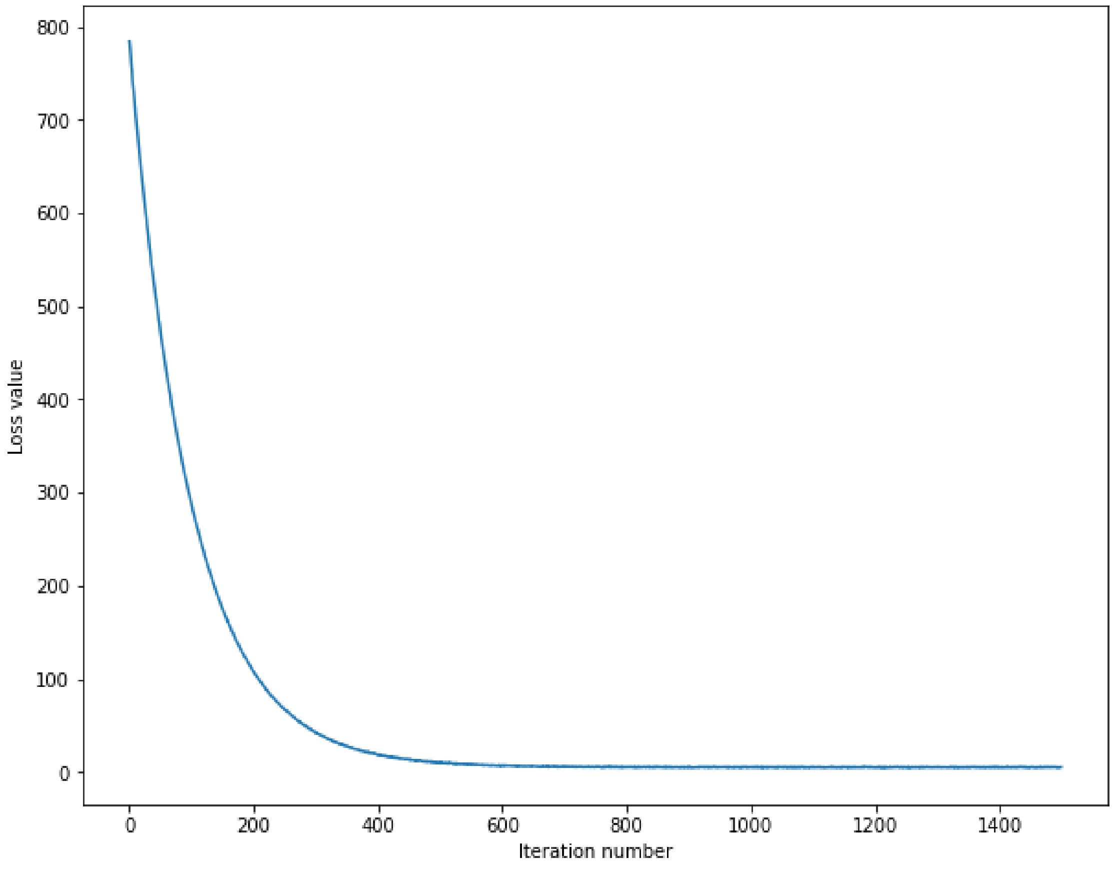
### 1.2.1 Stochastic Gradient Descent

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

[20]:
```
# 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 784.275509
iteration 100 / 1500: loss 286.412682
iteration 200 / 1500: loss 107.662329
iteration 300 / 1500: loss 43.159358
iteration 400 / 1500: loss 18.309788
iteration 500 / 1500: loss 10.013480
iteration 600 / 1500: loss 7.096349
iteration 700 / 1500: loss 6.354399
iteration 800 / 1500: loss 5.525041
iteration 900 / 1500: loss 5.158698
iteration 1000 / 1500: loss 5.430117
iteration 1100 / 1500: loss 5.370218
iteration 1200 / 1500: loss 5.148509
iteration 1300 / 1500: loss 5.732879
iteration 1400 / 1500: loss 5.297029
That took 7.140123s
```

[21]:
```
# 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()
```



[22]:
```
# 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), ))
```

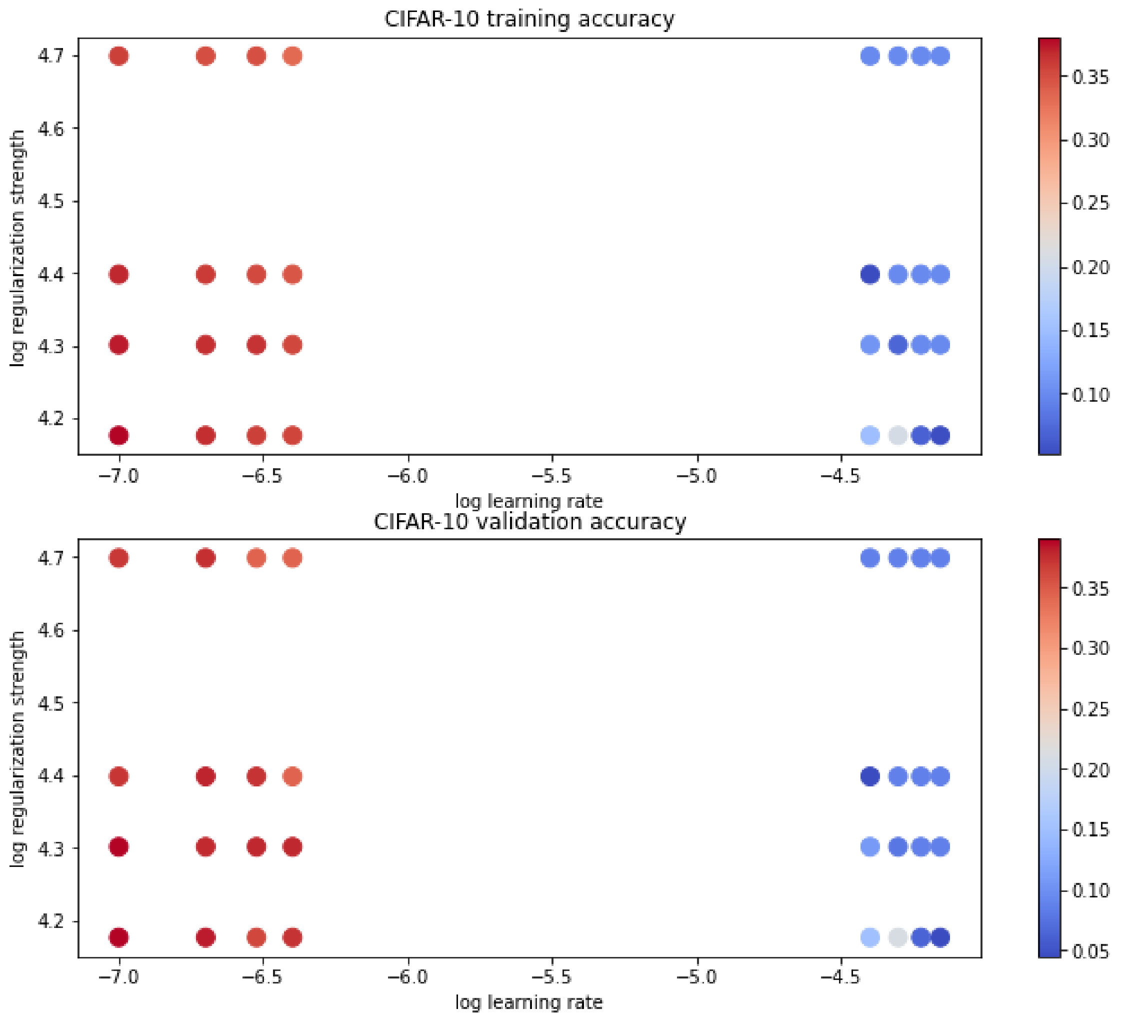
```
training accuracy: 0.365776
validation accuracy: 0.379000
```

[23]:
```
# 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.
best_lr = None
best_reg = None
################################################################################
# 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, 3e-7, 4e-7, 4e-5,5e-5,6e-5,7e-5]
regularization_strengths = [1.5e4,2e4,2.5e4, 5e4]

# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
#initiate the object svm of class LinearSVM
svm = LinearSVM()

#outer loop iterates over learning_rates
for lr in learning_rates:
    #inner loop to iterate over regularization_strengths (lambda)
    for reg in regularization_strengths:
        #initiate object
        svm = LinearSVM()
        #train the linear SVM
        loss_history = svm.train(X_train, y_train, learning_rate = lr, reg =␣
↪reg, num_iters = 5000)
        #find predictions for training
        y_train_pred = svm.predict(X_train)
        # computing the accuracy
        accuracy_train = np.mean(y_train_pred == y_train)
        # find predictions for validation
        y_val_pred = svm.predict(X_val)
        # compute the accurcay over validation data
        accuracy_val = np.mean(y_val_pred == y_val)
        if accuracy_val > best_val:
            best_lr = lr
            best_reg = reg
            best_val = accuracy_val
            best_svm = svm
        results[(lr, reg)] = accuracy_train, accuracy_val
# *****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)
```

```
/content/drive/My
Drive/ENPM809K/assignment1/assignment1/cs231n/classifiers/linear_svm.py:116:
RuntimeWarning: overflow encountered in double_scalars
  # Hint: Instead of computing the gradient from scratch, it may be easier    #
/usr/local/lib/python3.7/dist-packages/numpy/core/fromnumeric.py:86:
RuntimeWarning: overflow encountered in reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
/content/drive/My
Drive/ENPM809K/assignment1/assignment1/cs231n/classifiers/linear_svm.py:116:
RuntimeWarning: overflow encountered in multiply
  # Hint: Instead of computing the gradient from scratch, it may be easier    #
/content/drive/My
Drive/ENPM809K/assignment1/assignment1/cs231n/classifiers/linear_svm.py:107:
RuntimeWarning: overflow encountered in add
  loss += reg * np.sum(W * W)
/content/drive/My
Drive/ENPM809K/assignment1/assignment1/cs231n/classifiers/linear_svm.py:142:
RuntimeWarning: overflow encountered in multiply
/content/drive/My Drive/ENPM809K/assignment1/assignment1/cs231n/classifiers/line
ar_classifier.py:88: RuntimeWarning: invalid value encountered in add
  self.W += - learning_rate * grad
/content/drive/My
Drive/ENPM809K/assignment1/assignment1/cs231n/classifiers/linear_svm.py:107:
RuntimeWarning: invalid value encountered in add
  loss += reg * np.sum(W * W)
```

```
lr 1.000000e-07 reg 1.500000e+04 train accuracy: 0.380143 val accuracy: 0.389000
lr 1.000000e-07 reg 2.000000e+04 train accuracy: 0.373429 val accuracy: 0.390000
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.369735 val accuracy: 0.372000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.358061 val accuracy: 0.371000
lr 2.000000e-07 reg 1.500000e+04 train accuracy: 0.367306 val accuracy: 0.382000
lr 2.000000e-07 reg 2.000000e+04 train accuracy: 0.365102 val accuracy: 0.377000
lr 2.000000e-07 reg 2.500000e+04 train accuracy: 0.360918 val accuracy: 0.380000
lr 2.000000e-07 reg 5.000000e+04 train accuracy: 0.349776 val accuracy: 0.374000
lr 3.000000e-07 reg 1.500000e+04 train accuracy: 0.357939 val accuracy: 0.362000
lr 3.000000e-07 reg 2.000000e+04 train accuracy: 0.363898 val accuracy: 0.378000
lr 3.000000e-07 reg 2.500000e+04 train accuracy: 0.352469 val accuracy: 0.373000
lr 3.000000e-07 reg 5.000000e+04 train accuracy: 0.348224 val accuracy: 0.345000
lr 4.000000e-07 reg 1.500000e+04 train accuracy: 0.355388 val accuracy: 0.373000
lr 4.000000e-07 reg 2.000000e+04 train accuracy: 0.354184 val accuracy: 0.379000
lr 4.000000e-07 reg 2.500000e+04 train accuracy: 0.344286 val accuracy: 0.345000
lr 4.000000e-07 reg 5.000000e+04 train accuracy: 0.333122 val accuracy: 0.345000
lr 4.000000e-05 reg 1.500000e+04 train accuracy: 0.149796 val accuracy: 0.151000
lr 4.000000e-05 reg 2.000000e+04 train accuracy: 0.108694 val accuracy: 0.110000
lr 4.000000e-05 reg 2.500000e+04 train accuracy: 0.052306 val accuracy: 0.045000
lr 4.000000e-05 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 5.000000e-05 reg 1.500000e+04 train accuracy: 0.206061 val accuracy: 0.210000
lr 5.000000e-05 reg 2.000000e+04 train accuracy: 0.072633 val accuracy: 0.079000
lr 5.000000e-05 reg 2.500000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 5.000000e-05 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 6.000000e-05 reg 1.500000e+04 train accuracy: 0.068939 val accuracy: 0.064000
lr 6.000000e-05 reg 2.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 6.000000e-05 reg 2.500000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 6.000000e-05 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 7.000000e-05 reg 1.500000e+04 train accuracy: 0.053653 val accuracy: 0.046000
lr 7.000000e-05 reg 2.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 7.000000e-05 reg 2.500000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 7.000000e-05 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
best validation accuracy achieved during cross-validation: 0.390000
```

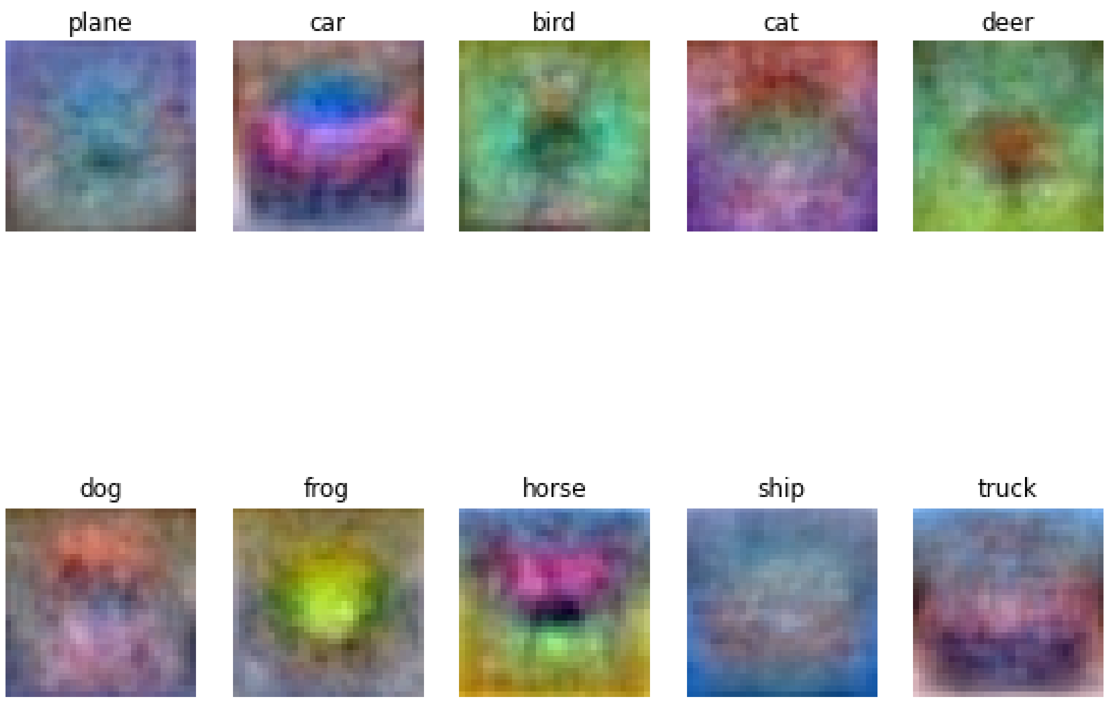
[24]:
```
# Visualize the cross-validation results
import math
import pdb

# 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()
```



[25]:
```
# 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: 0.371000
```

[26]:
```
# Visualize the learned weights for each class.
# Depending on your choice of learning rate and regularization strength, these␣
 ↪may
# or may not be nice to look at.
w = best_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.

$YourAnswer:$ *These SVM weights look like templates for each class. They look like that so as to provide maximum impulse to inner product of the specific class template to a data point of that class.*

# softmax

October 9, 2022

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

# Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment1/'
FOLDERNAME = 'ENPM809K/assignment1/assignment1/'
assert FOLDERNAME is not None, "[!] Enter the foldername."

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

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

```
Mounted at /content/drive
/content/drive/My Drive/ENPM809K/assignment1/assignment1/cs231n/datasets
/content/drive/My Drive/ENPM809K/assignment1/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: 2.371193
sanity check: 2.302585
```

**Inline Question 1**

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

$Your Answer:$ *Since we haven't started the training yet, all the classes are equally likely to be chosen. Since we have samples distributed into ten classes, the probability of the correct class will be 1/10 = 0.1. Therefore, the softmax loss is expected to be -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: 1.595245 analytic: 1.595245, relative error: 2.468730e-08
numerical: 1.001458 analytic: 1.001458, relative error: 8.074479e-08
numerical: 2.125066 analytic: 2.125066, relative error: 2.419642e-08
numerical: 0.104744 analytic: 0.104744, relative error: 4.962576e-07
numerical: 1.073973 analytic: 1.073973, relative error: 3.830552e-09
numerical: 0.745591 analytic: 0.745591, relative error: 9.934502e-08
numerical: -0.371764 analytic: -0.371764, relative error: 2.846215e-08
numerical: 1.468602 analytic: 1.468602, relative error: 2.710553e-08
numerical: -0.604993 analytic: -0.604993, relative error: 7.012845e-08
numerical: -0.037821 analytic: -0.037821, relative error: 1.269128e-06
numerical: -0.524170 analytic: -0.524170, relative error: 1.535719e-07
numerical: -0.553308 analytic: -0.553308, relative error: 5.698778e-09
numerical: 0.843525 analytic: 0.843525, relative error: 7.610537e-09
numerical: 1.188739 analytic: 1.188739, relative error: 4.132200e-08
numerical: 2.154791 analytic: 2.154791, relative error: 4.013855e-08
numerical: -5.292296 analytic: -5.292296, relative error: 2.023621e-08
numerical: -1.649226 analytic: -1.649226, relative error: 1.257342e-08
numerical: -0.696365 analytic: -0.696365, relative error: 9.623944e-09
numerical: -5.496393 analytic: -5.496394, relative error: 1.558244e-08
numerical: -1.559216 analytic: -1.559216, relative error: 1.418963e-08
```

[ ]:
```
# 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: 2.417757e+00 computed in 0.168233s
vectorized loss: 2.417757e+00 computed in 0.025619s
```

```
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-7,2e-7,3e-7, 5e-7,2e-6,3e-6,3.5e-6]
regularization_strengths = [4e2,5e2,3e3,2.5e4,5e4]

# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
softmax = Softmax()
for lr in learning_rates:
    for reg in regularization_strengths:
        loss_hist = softmax.train(X_train, y_train, learning_rate=lr, reg=reg,␣
↪num_iters=1500)

        y_train_pred = softmax.predict(X_train)
        acc_train = np.mean(y_train == y_train_pred)

        y_val_pred = softmax.predict(X_val)
        acc_val = np.mean(y_val == y_val_pred)

        results[(lr, reg)] = (acc_train, acc_val)

        if acc_val > best_val:
            best_val = acc_val
            best_softmax = softmax

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

# Print out results.
```

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

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

```
lr 1.000000e-07 reg 4.000000e+02 train accuracy: 0.257633 val accuracy: 0.260000
lr 1.000000e-07 reg 5.000000e+02 train accuracy: 0.300143 val accuracy: 0.310000
lr 1.000000e-07 reg 3.000000e+03 train accuracy: 0.354939 val accuracy: 0.376000
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.335939 val accuracy: 0.351000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.308796 val accuracy: 0.323000
lr 2.000000e-07 reg 4.000000e+02 train accuracy: 0.399796 val accuracy: 0.407000
lr 2.000000e-07 reg 5.000000e+02 train accuracy: 0.410755 val accuracy: 0.414000
lr 2.000000e-07 reg 3.000000e+03 train accuracy: 0.386143 val accuracy: 0.407000
lr 2.000000e-07 reg 2.500000e+04 train accuracy: 0.327714 val accuracy: 0.346000
lr 2.000000e-07 reg 5.000000e+04 train accuracy: 0.307735 val accuracy: 0.329000
lr 3.000000e-07 reg 4.000000e+02 train accuracy: 0.404510 val accuracy: 0.400000
lr 3.000000e-07 reg 5.000000e+02 train accuracy: 0.410673 val accuracy: 0.410000
lr 3.000000e-07 reg 3.000000e+03 train accuracy: 0.388714 val accuracy: 0.391000
lr 3.000000e-07 reg 2.500000e+04 train accuracy: 0.329000 val accuracy: 0.337000
lr 3.000000e-07 reg 5.000000e+04 train accuracy: 0.300755 val accuracy: 0.314000
lr 5.000000e-07 reg 4.000000e+02 train accuracy: 0.407265 val accuracy: 0.407000
lr 5.000000e-07 reg 5.000000e+02 train accuracy: 0.411816 val accuracy: 0.405000
lr 5.000000e-07 reg 3.000000e+03 train accuracy: 0.378163 val accuracy: 0.391000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.335592 val accuracy: 0.356000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.290327 val accuracy: 0.307000
lr 2.000000e-06 reg 4.000000e+02 train accuracy: 0.407612 val accuracy: 0.385000
lr 2.000000e-06 reg 5.000000e+02 train accuracy: 0.402020 val accuracy: 0.406000
lr 2.000000e-06 reg 3.000000e+03 train accuracy: 0.363735 val accuracy: 0.387000
lr 2.000000e-06 reg 2.500000e+04 train accuracy: 0.310327 val accuracy: 0.325000
lr 2.000000e-06 reg 5.000000e+04 train accuracy: 0.274837 val accuracy: 0.290000
lr 3.000000e-06 reg 4.000000e+02 train accuracy: 0.395571 val accuracy: 0.382000
lr 3.000000e-06 reg 5.000000e+02 train accuracy: 0.394490 val accuracy: 0.394000
lr 3.000000e-06 reg 3.000000e+03 train accuracy: 0.353898 val accuracy: 0.357000
lr 3.000000e-06 reg 2.500000e+04 train accuracy: 0.300735 val accuracy: 0.310000
lr 3.000000e-06 reg 5.000000e+04 train accuracy: 0.269490 val accuracy: 0.277000
lr 3.500000e-06 reg 4.000000e+02 train accuracy: 0.394878 val accuracy: 0.399000
lr 3.500000e-06 reg 5.000000e+02 train accuracy: 0.377878 val accuracy: 0.367000
lr 3.500000e-06 reg 3.000000e+03 train accuracy: 0.336592 val accuracy: 0.347000
lr 3.500000e-06 reg 2.500000e+04 train accuracy: 0.279490 val accuracy: 0.300000
lr 3.500000e-06 reg 5.000000e+04 train accuracy: 0.253959 val accuracy: 0.274000
best validation accuracy achieved during cross-validation: 0.414000
```

[9]:
```
# 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: 0.284000
```

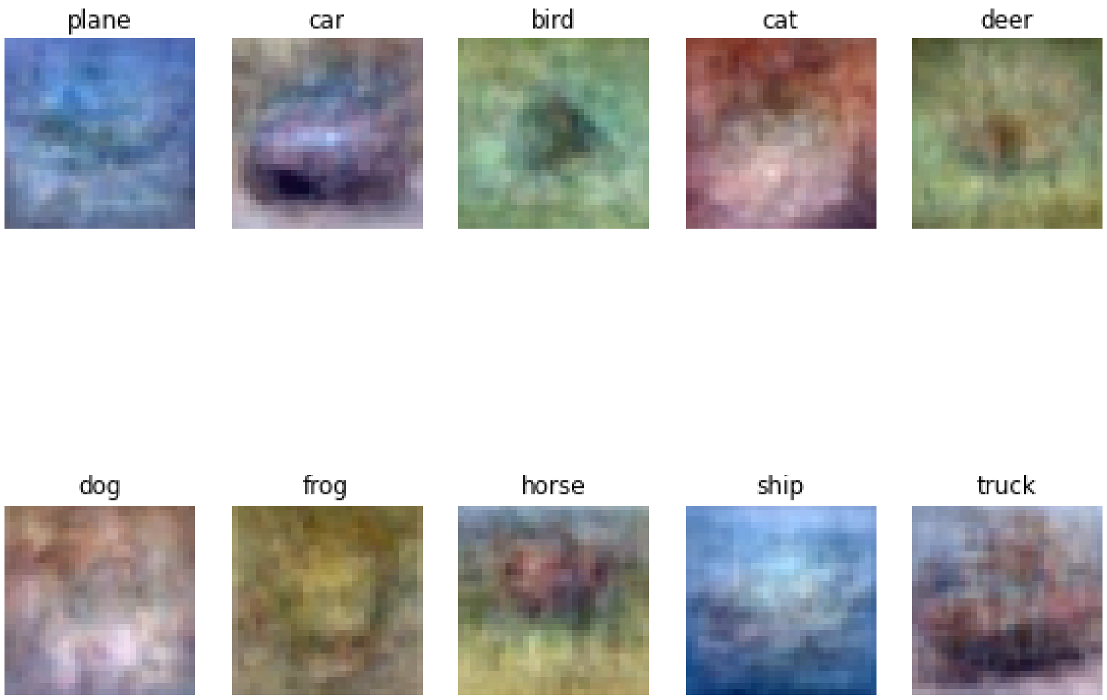
**Inline Question 2** - *True or False*

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.

$Your Answer :$ True

$Your Explanation :$ Because SVM loss stays unbothered as long as the new data point is not from the correct class and the incoorect class scores is less than the correct class score by at least ”delta”. However, since softmax considers probability, adding a new point decreases the probabilty of the correct class. That is adding a new point incurs more loss, since it decreases the score of correct class.

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

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

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

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


plane
car
bird
cat
deer
dog
frog
horse
ship
truck

```
[ ]:
```

# two_layer_net

October 9, 2022

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

# Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment1/'
FOLDERNAME = 'ENPM809K/assignment1/assignment1/'
assert FOLDERNAME is not None, "[!] Enter the foldername."

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

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

```
Mounted at /content/drive
/content/drive/My Drive/ENPM809K/assignment1/assignment1/cs231n/datasets
/content/drive/My Drive/ENPM809K/assignment1/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.

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

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

[17]:
```
# 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:

[18]:
```
# 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.

[24]:
```
# 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:

[25]:
```
# 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:

[26]:
```
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:

We know how the sigmoid function looks like an 'S', that is its slope is ~ 0 when the sigmoid is applied to a value close to 1 and 0. Since we know sigmoid is (1/1 + e^(-z)), Egs: if z is very very large, e^-z tends to 0, and the value is close to 1. Hence when taking gradient wrto inputs, the gradient will tend to zero as the graphical representation shows the slope of sigmoid to be almost zero at 1 and 0.

Similarly, Relu function which looks like an obtuse angle, has its slope ~ 0 when the function is applied to 0 and negative numbers. We know relu is max(0,z) therefore if we have a z which is negative( maybe -123), the gradient will tend to zero, thus creating issues during backprop.

I believe Leaky relu might not have this issue as it will always have a decent gradient when the function is applied to positive as well as negative values, that is 1 when positive and some constant when negative. Issues might only arise during backprop when value is 0.

## 6 "Sandwich" layers

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

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

[27]:
```
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:

[42]:
```
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.

[45]:
```
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.

[46]:
```
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)*****
# created an instance using the above completed sgd function from optim.py
solver = Solver(model, data,
                    update_rule='sgd',
                    optim_config={
                      'learning_rate': 1e-4,
                    },
                    lr_decay=0.95,
                    num_epochs=10, batch_size=100,
                    print_every=100)
solver.train()


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

```
(Iteration 1 / 4900) loss: 2.300089
(Epoch 0 / 10) train acc: 0.138000; val_acc: 0.138000
(Iteration 101 / 4900) loss: 2.240327
(Iteration 201 / 4900) loss: 2.113574
(Iteration 301 / 4900) loss: 2.026074
(Iteration 401 / 4900) loss: 2.048385
(Epoch 1 / 10) train acc: 0.297000; val_acc: 0.300000
(Iteration 501 / 4900) loss: 1.979085
(Iteration 601 / 4900) loss: 1.835247
(Iteration 701 / 4900) loss: 1.864985
(Iteration 801 / 4900) loss: 1.791849
(Iteration 901 / 4900) loss: 1.769555
(Epoch 2 / 10) train acc: 0.354000; val_acc: 0.360000
(Iteration 1001 / 4900) loss: 1.719080
(Iteration 1101 / 4900) loss: 1.742622
(Iteration 1201 / 4900) loss: 1.756344
(Iteration 1301 / 4900) loss: 1.795856
(Iteration 1401 / 4900) loss: 1.795612
(Epoch 3 / 10) train acc: 0.404000; val_acc: 0.384000
(Iteration 1501 / 4900) loss: 1.720783
(Iteration 1601 / 4900) loss: 1.765353
(Iteration 1701 / 4900) loss: 1.713281
(Iteration 1801 / 4900) loss: 1.684253
(Iteration 1901 / 4900) loss: 1.797089
(Epoch 4 / 10) train acc: 0.405000; val_acc: 0.412000
(Iteration 2001 / 4900) loss: 1.672947
(Iteration 2101 / 4900) loss: 1.636405
(Iteration 2201 / 4900) loss: 1.857021
(Iteration 2301 / 4900) loss: 1.465527
```

```
(Iteration 2401 / 4900) loss: 1.627937
(Epoch 5 / 10) train acc: 0.405000; val_acc: 0.428000
(Iteration 2501 / 4900) loss: 1.591481
(Iteration 2601 / 4900) loss: 1.691214
(Iteration 2701 / 4900) loss: 1.574824
(Iteration 2801 / 4900) loss: 1.728493
(Iteration 2901 / 4900) loss: 1.488696
(Epoch 6 / 10) train acc: 0.439000; val_acc: 0.438000
(Iteration 3001 / 4900) loss: 1.600789
(Iteration 3101 / 4900) loss: 1.546226
(Iteration 3201 / 4900) loss: 1.625871
(Iteration 3301 / 4900) loss: 1.519366
(Iteration 3401 / 4900) loss: 1.423495
(Epoch 7 / 10) train acc: 0.465000; val_acc: 0.443000
(Iteration 3501 / 4900) loss: 1.760888
(Iteration 3601 / 4900) loss: 1.415782
(Iteration 3701 / 4900) loss: 1.874383
(Iteration 3801 / 4900) loss: 1.583471
(Iteration 3901 / 4900) loss: 1.465142
(Epoch 8 / 10) train acc: 0.438000; val_acc: 0.459000
(Iteration 4001 / 4900) loss: 1.475411
(Iteration 4101 / 4900) loss: 1.691647
(Iteration 4201 / 4900) loss: 1.512262
(Iteration 4301 / 4900) loss: 1.582098
(Iteration 4401 / 4900) loss: 1.399120
(Epoch 9 / 10) train acc: 0.479000; val_acc: 0.452000
(Iteration 4501 / 4900) loss: 1.629088
(Iteration 4601 / 4900) loss: 1.461418
(Iteration 4701 / 4900) loss: 1.583787
(Iteration 4801 / 4900) loss: 1.501737
(Epoch 10 / 10) train acc: 0.438000; val_acc: 0.457000
```

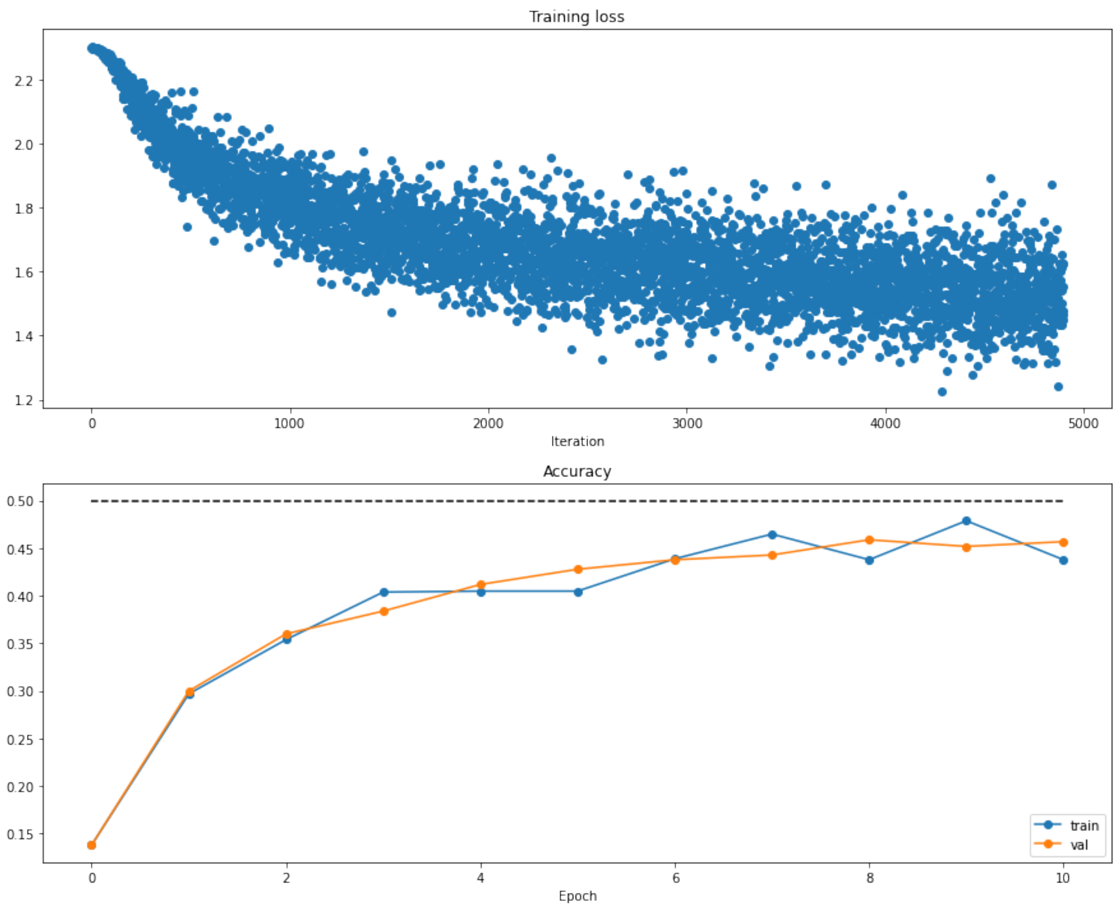
## 10 Debug the training

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.

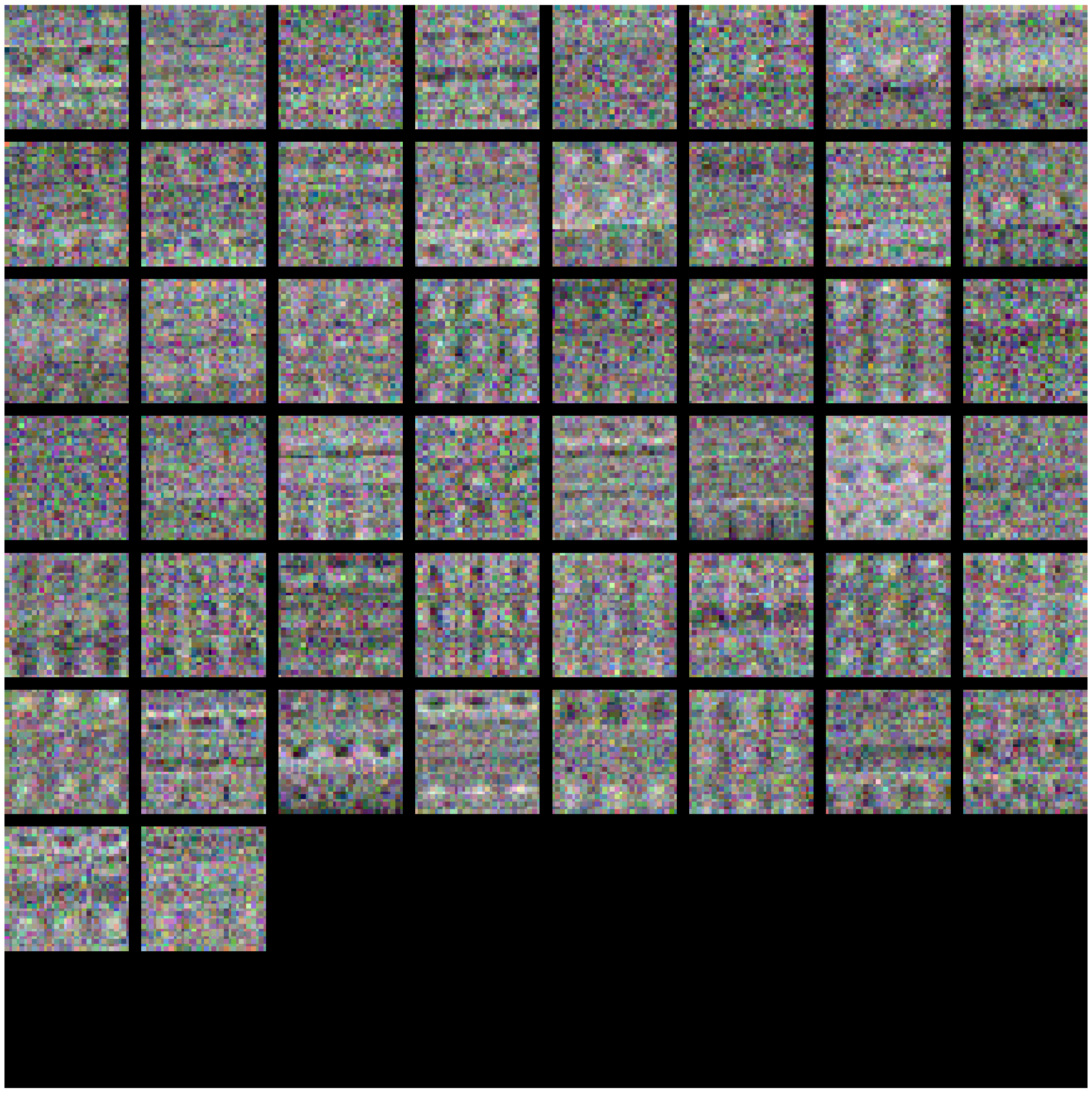
One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

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.

[47]:
```
# 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()
```



[48]:
```
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.).

[49]:
```
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)*****

results = {}
best_val = -1.0
best_model = None

#options for Learning rates and regulerization strengths
learning_rates=[0.0003, 0.003,0.03,0.0001]
regularization_strengths=[1e-05, 1e-04, 1e-03, 1e-02]

for l_r in learning_rates:
  for r_s in regularization_strengths:
    #model is
    model = TwoLayerNet(hidden_dim=128, reg=r_s)
    #Define an instance
    solver = Solver(model, data,
                    update_rule='sgd',
                    optim_config={
                      'learning_rate':l_r,
                    },
                    lr_decay=0.95,
                    num_epochs=10, batch_size=100,
                    print_every=100)
    solver.train()

    #Find accuracy using the solver api
    results[(l_r,r_s)]=solver.best_val_acc
    #always compares and sets the best acuracy.
    if best_val<results[(l_r,r_s)]:
      best_val=results[(l_r,r_s)]
      best_model=model
for lr, reg in sorted(results):
    val_accuracy = results[(lr, reg)]
    print('lr %e reg : %f val accuracy: %f' %  (lr, reg, val_accuracy))

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

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

```
(Iteration 1 / 4900) loss: 2.294756
(Epoch 0 / 10) train acc: 0.124000; val_acc: 0.124000
(Iteration 101 / 4900) loss: 2.096886
(Iteration 201 / 4900) loss: 1.977227
```

```
(Iteration 301 / 4900) loss: 1.731897
(Iteration 401 / 4900) loss: 1.703324
(Epoch 1 / 10) train acc: 0.378000; val_acc: 0.411000
(Iteration 501 / 4900) loss: 1.668206
(Iteration 601 / 4900) loss: 1.570562
(Iteration 701 / 4900) loss: 1.723392
(Iteration 801 / 4900) loss: 1.646265
(Iteration 901 / 4900) loss: 1.710248
(Epoch 2 / 10) train acc: 0.436000; val_acc: 0.452000
(Iteration 1001 / 4900) loss: 1.619902
(Iteration 1101 / 4900) loss: 1.640216
(Iteration 1201 / 4900) loss: 1.487124
(Iteration 1301 / 4900) loss: 1.316196
(Iteration 1401 / 4900) loss: 1.601113
(Epoch 3 / 10) train acc: 0.484000; val_acc: 0.464000
(Iteration 1501 / 4900) loss: 1.426193
(Iteration 1601 / 4900) loss: 1.351808
(Iteration 1701 / 4900) loss: 1.459730
(Iteration 1801 / 4900) loss: 1.396966
(Iteration 1901 / 4900) loss: 1.350157
(Epoch 4 / 10) train acc: 0.517000; val_acc: 0.489000
(Iteration 2001 / 4900) loss: 1.529192
(Iteration 2101 / 4900) loss: 1.436588
(Iteration 2201 / 4900) loss: 1.376303
(Iteration 2301 / 4900) loss: 1.462412
(Iteration 2401 / 4900) loss: 1.371197
(Epoch 5 / 10) train acc: 0.500000; val_acc: 0.494000
(Iteration 2501 / 4900) loss: 1.302286
(Iteration 2601 / 4900) loss: 1.548265
(Iteration 2701 / 4900) loss: 1.245254
(Iteration 2801 / 4900) loss: 1.360981
(Iteration 2901 / 4900) loss: 1.409725
(Epoch 6 / 10) train acc: 0.528000; val_acc: 0.503000
(Iteration 3001 / 4900) loss: 1.380514
(Iteration 3101 / 4900) loss: 1.573215
(Iteration 3201 / 4900) loss: 1.495966
(Iteration 3301 / 4900) loss: 1.439574
(Iteration 3401 / 4900) loss: 1.358185
(Epoch 7 / 10) train acc: 0.554000; val_acc: 0.497000
(Iteration 3501 / 4900) loss: 1.459783
(Iteration 3601 / 4900) loss: 1.349189
(Iteration 3701 / 4900) loss: 1.490410
(Iteration 3801 / 4900) loss: 1.412612
(Iteration 3901 / 4900) loss: 1.300477
(Epoch 8 / 10) train acc: 0.553000; val_acc: 0.499000
(Iteration 4001 / 4900) loss: 1.311037
(Iteration 4101 / 4900) loss: 1.313193
(Iteration 4201 / 4900) loss: 1.373106
```

```
(Iteration 4301 / 4900) loss: 1.289426
(Iteration 4401 / 4900) loss: 1.267751
(Epoch 9 / 10) train acc: 0.574000; val_acc: 0.505000
(Iteration 4501 / 4900) loss: 1.122695
(Iteration 4601 / 4900) loss: 1.216056
(Iteration 4701 / 4900) loss: 1.088488
(Iteration 4801 / 4900) loss: 1.155218
(Epoch 10 / 10) train acc: 0.549000; val_acc: 0.498000
(Iteration 1 / 4900) loss: 2.301164
(Epoch 0 / 10) train acc: 0.139000; val_acc: 0.120000
(Iteration 101 / 4900) loss: 2.037867
(Iteration 201 / 4900) loss: 1.816868
(Iteration 301 / 4900) loss: 1.715659
(Iteration 401 / 4900) loss: 1.761528
(Epoch 1 / 10) train acc: 0.395000; val_acc: 0.402000
(Iteration 501 / 4900) loss: 1.686904
(Iteration 601 / 4900) loss: 1.543582
(Iteration 701 / 4900) loss: 1.468454
(Iteration 801 / 4900) loss: 1.615527
(Iteration 901 / 4900) loss: 1.523456
(Epoch 2 / 10) train acc: 0.445000; val_acc: 0.462000
(Iteration 1001 / 4900) loss: 1.598658
(Iteration 1101 / 4900) loss: 1.620349
(Iteration 1201 / 4900) loss: 1.640968
(Iteration 1301 / 4900) loss: 1.318954
(Iteration 1401 / 4900) loss: 1.582075
(Epoch 3 / 10) train acc: 0.448000; val_acc: 0.465000
(Iteration 1501 / 4900) loss: 1.560436
(Iteration 1601 / 4900) loss: 1.435951
(Iteration 1701 / 4900) loss: 1.551070
(Iteration 1801 / 4900) loss: 1.349432
(Iteration 1901 / 4900) loss: 1.526112
(Epoch 4 / 10) train acc: 0.501000; val_acc: 0.494000
(Iteration 2001 / 4900) loss: 1.416872
(Iteration 2101 / 4900) loss: 1.310585
(Iteration 2201 / 4900) loss: 1.364590
(Iteration 2301 / 4900) loss: 1.176052
(Iteration 2401 / 4900) loss: 1.297875
(Epoch 5 / 10) train acc: 0.489000; val_acc: 0.491000
(Iteration 2501 / 4900) loss: 1.154520
(Iteration 2601 / 4900) loss: 1.366748
(Iteration 2701 / 4900) loss: 1.370802
(Iteration 2801 / 4900) loss: 1.387843
(Iteration 2901 / 4900) loss: 1.226963
(Epoch 6 / 10) train acc: 0.515000; val_acc: 0.491000
(Iteration 3001 / 4900) loss: 1.414206
(Iteration 3101 / 4900) loss: 1.268182
(Iteration 3201 / 4900) loss: 1.237369
```

```
(Iteration 3301 / 4900) loss: 1.142320
(Iteration 3401 / 4900) loss: 1.411195
(Epoch 7 / 10) train acc: 0.558000; val_acc: 0.499000
(Iteration 3501 / 4900) loss: 1.229914
(Iteration 3601 / 4900) loss: 1.325166
(Iteration 3701 / 4900) loss: 1.639310
(Iteration 3801 / 4900) loss: 1.219475
(Iteration 3901 / 4900) loss: 1.294610
(Epoch 8 / 10) train acc: 0.551000; val_acc: 0.497000
(Iteration 4001 / 4900) loss: 1.353362
(Iteration 4101 / 4900) loss: 1.291900
(Iteration 4201 / 4900) loss: 1.313074
(Iteration 4301 / 4900) loss: 1.274888
(Iteration 4401 / 4900) loss: 1.122169
(Epoch 9 / 10) train acc: 0.580000; val_acc: 0.508000
(Iteration 4501 / 4900) loss: 1.296192
(Iteration 4601 / 4900) loss: 1.134963
(Iteration 4701 / 4900) loss: 1.206072
(Iteration 4801 / 4900) loss: 1.225292
(Epoch 10 / 10) train acc: 0.594000; val_acc: 0.495000
(Iteration 1 / 4900) loss: 2.303151
(Epoch 0 / 10) train acc: 0.096000; val_acc: 0.123000
(Iteration 101 / 4900) loss: 2.146238
(Iteration 201 / 4900) loss: 1.888898
(Iteration 301 / 4900) loss: 1.719703
(Iteration 401 / 4900) loss: 1.693153
(Epoch 1 / 10) train acc: 0.399000; val_acc: 0.413000
(Iteration 501 / 4900) loss: 1.721672
(Iteration 601 / 4900) loss: 1.638116
(Iteration 701 / 4900) loss: 1.687138
(Iteration 801 / 4900) loss: 1.605081
(Iteration 901 / 4900) loss: 1.670455
(Epoch 2 / 10) train acc: 0.445000; val_acc: 0.461000
(Iteration 1001 / 4900) loss: 1.480779
(Iteration 1101 / 4900) loss: 1.358341
(Iteration 1201 / 4900) loss: 1.566854
(Iteration 1301 / 4900) loss: 1.469887
(Iteration 1401 / 4900) loss: 1.590604
(Epoch 3 / 10) train acc: 0.482000; val_acc: 0.455000
(Iteration 1501 / 4900) loss: 1.562382
(Iteration 1601 / 4900) loss: 1.502752
(Iteration 1701 / 4900) loss: 1.421909
(Iteration 1801 / 4900) loss: 1.383171
(Iteration 1901 / 4900) loss: 1.653718
(Epoch 4 / 10) train acc: 0.517000; val_acc: 0.468000
(Iteration 2001 / 4900) loss: 1.500934
(Iteration 2101 / 4900) loss: 1.377263
(Iteration 2201 / 4900) loss: 1.288379
```

```
(Iteration 2301 / 4900) loss: 1.297869
(Iteration 2401 / 4900) loss: 1.608880
(Epoch 5 / 10) train acc: 0.504000; val_acc: 0.481000
(Iteration 2501 / 4900) loss: 1.582997
(Iteration 2601 / 4900) loss: 1.261267
(Iteration 2701 / 4900) loss: 1.429974
(Iteration 2801 / 4900) loss: 1.526015
(Iteration 2901 / 4900) loss: 1.395960
(Epoch 6 / 10) train acc: 0.554000; val_acc: 0.495000
(Iteration 3001 / 4900) loss: 1.377277
(Iteration 3101 / 4900) loss: 1.308682
(Iteration 3201 / 4900) loss: 1.453303
(Iteration 3301 / 4900) loss: 1.267776
(Iteration 3401 / 4900) loss: 1.305942
(Epoch 7 / 10) train acc: 0.540000; val_acc: 0.500000
(Iteration 3501 / 4900) loss: 1.428083
(Iteration 3601 / 4900) loss: 1.355721
(Iteration 3701 / 4900) loss: 1.427412
(Iteration 3801 / 4900) loss: 1.268831
(Iteration 3901 / 4900) loss: 1.270191
(Epoch 8 / 10) train acc: 0.558000; val_acc: 0.504000
(Iteration 4001 / 4900) loss: 1.283741
(Iteration 4101 / 4900) loss: 1.394150
(Iteration 4201 / 4900) loss: 1.333314
(Iteration 4301 / 4900) loss: 1.260032
(Iteration 4401 / 4900) loss: 1.379422
(Epoch 9 / 10) train acc: 0.586000; val_acc: 0.511000
(Iteration 4501 / 4900) loss: 1.232680
(Iteration 4601 / 4900) loss: 1.314834
(Iteration 4701 / 4900) loss: 1.237595
(Iteration 4801 / 4900) loss: 1.159139
(Epoch 10 / 10) train acc: 0.591000; val_acc: 0.518000
(Iteration 1 / 4900) loss: 2.307533
(Epoch 0 / 10) train acc: 0.108000; val_acc: 0.093000
(Iteration 101 / 4900) loss: 2.039559
(Iteration 201 / 4900) loss: 1.771865
(Iteration 301 / 4900) loss: 1.893575
(Iteration 401 / 4900) loss: 1.732392
(Epoch 1 / 10) train acc: 0.389000; val_acc: 0.405000
(Iteration 501 / 4900) loss: 1.659042
(Iteration 601 / 4900) loss: 1.616332
(Iteration 701 / 4900) loss: 1.657626
(Iteration 801 / 4900) loss: 1.633303
(Iteration 901 / 4900) loss: 1.616044
(Epoch 2 / 10) train acc: 0.462000; val_acc: 0.467000
(Iteration 1001 / 4900) loss: 1.472532
(Iteration 1101 / 4900) loss: 1.603767
(Iteration 1201 / 4900) loss: 1.555852
```

```
(Iteration 1301 / 4900) loss: 1.711556
(Iteration 1401 / 4900) loss: 1.405805
(Epoch 3 / 10) train acc: 0.478000; val_acc: 0.479000
(Iteration 1501 / 4900) loss: 1.391202
(Iteration 1601 / 4900) loss: 1.522979
(Iteration 1701 / 4900) loss: 1.479173
(Iteration 1801 / 4900) loss: 1.454039
(Iteration 1901 / 4900) loss: 1.626017
(Epoch 4 / 10) train acc: 0.492000; val_acc: 0.489000
(Iteration 2001 / 4900) loss: 1.349871
(Iteration 2101 / 4900) loss: 1.497771
(Iteration 2201 / 4900) loss: 1.401483
(Iteration 2301 / 4900) loss: 1.545620
(Iteration 2401 / 4900) loss: 1.306978
(Epoch 5 / 10) train acc: 0.541000; val_acc: 0.494000
(Iteration 2501 / 4900) loss: 1.426658
(Iteration 2601 / 4900) loss: 1.190509
(Iteration 2701 / 4900) loss: 1.711136
(Iteration 2801 / 4900) loss: 1.193103
(Iteration 2901 / 4900) loss: 1.317829
(Epoch 6 / 10) train acc: 0.508000; val_acc: 0.503000
(Iteration 3001 / 4900) loss: 1.341375
(Iteration 3101 / 4900) loss: 1.242421
(Iteration 3201 / 4900) loss: 1.351006
(Iteration 3301 / 4900) loss: 1.273311
(Iteration 3401 / 4900) loss: 1.436704
(Epoch 7 / 10) train acc: 0.571000; val_acc: 0.509000
(Iteration 3501 / 4900) loss: 1.325845
(Iteration 3601 / 4900) loss: 1.399553
(Iteration 3701 / 4900) loss: 1.231667
(Iteration 3801 / 4900) loss: 1.373820
(Iteration 3901 / 4900) loss: 1.159502
(Epoch 8 / 10) train acc: 0.540000; val_acc: 0.527000
(Iteration 4001 / 4900) loss: 1.522265
(Iteration 4101 / 4900) loss: 1.331372
(Iteration 4201 / 4900) loss: 1.244598
(Iteration 4301 / 4900) loss: 1.201152
(Iteration 4401 / 4900) loss: 1.243727
(Epoch 9 / 10) train acc: 0.565000; val_acc: 0.520000
(Iteration 4501 / 4900) loss: 1.174260
(Iteration 4601 / 4900) loss: 1.569459
(Iteration 4701 / 4900) loss: 1.221230
(Iteration 4801 / 4900) loss: 1.397368
(Epoch 10 / 10) train acc: 0.587000; val_acc: 0.518000
(Iteration 1 / 4900) loss: 2.301584
(Epoch 0 / 10) train acc: 0.163000; val_acc: 0.166000
(Iteration 101 / 4900) loss: 1.809733
(Iteration 201 / 4900) loss: 2.051486
```

```
(Iteration 301 / 4900) loss: 1.755945
(Iteration 401 / 4900) loss: 2.243014
(Epoch 1 / 10) train acc: 0.362000; val_acc: 0.345000
(Iteration 501 / 4900) loss: 1.706716
(Iteration 601 / 4900) loss: 1.790794
(Iteration 701 / 4900) loss: 1.829594
(Iteration 801 / 4900) loss: 2.425740
(Iteration 901 / 4900) loss: 2.129500
(Epoch 2 / 10) train acc: 0.290000; val_acc: 0.311000
(Iteration 1001 / 4900) loss: 1.687654
(Iteration 1101 / 4900) loss: 1.903633
(Iteration 1201 / 4900) loss: 1.541249
(Iteration 1301 / 4900) loss: 1.738618
(Iteration 1401 / 4900) loss: 2.426602
(Epoch 3 / 10) train acc: 0.405000; val_acc: 0.376000
(Iteration 1501 / 4900) loss: 1.777144
(Iteration 1601 / 4900) loss: 1.980776
(Iteration 1701 / 4900) loss: 1.631863
(Iteration 1801 / 4900) loss: 2.018592
(Iteration 1901 / 4900) loss: 1.815881
(Epoch 4 / 10) train acc: 0.400000; val_acc: 0.372000
(Iteration 2001 / 4900) loss: 1.920685
(Iteration 2101 / 4900) loss: 2.081721
(Iteration 2201 / 4900) loss: 1.468807
(Iteration 2301 / 4900) loss: 1.683507
(Iteration 2401 / 4900) loss: 1.413741
(Epoch 5 / 10) train acc: 0.394000; val_acc: 0.377000
(Iteration 2501 / 4900) loss: 1.602968
(Iteration 2601 / 4900) loss: 1.645252
(Iteration 2701 / 4900) loss: 1.800870
(Iteration 2801 / 4900) loss: 1.624507
(Iteration 2901 / 4900) loss: 1.514651
(Epoch 6 / 10) train acc: 0.465000; val_acc: 0.393000
(Iteration 3001 / 4900) loss: 1.513670
(Iteration 3101 / 4900) loss: 1.873989
(Iteration 3201 / 4900) loss: 1.589814
(Iteration 3301 / 4900) loss: 1.659744
(Iteration 3401 / 4900) loss: 1.802075
(Epoch 7 / 10) train acc: 0.451000; val_acc: 0.411000
(Iteration 3501 / 4900) loss: 1.304300
(Iteration 3601 / 4900) loss: 1.723055
(Iteration 3701 / 4900) loss: 1.823142
(Iteration 3801 / 4900) loss: 1.657015
(Iteration 3901 / 4900) loss: 1.177468
(Epoch 8 / 10) train acc: 0.478000; val_acc: 0.433000
(Iteration 4001 / 4900) loss: 1.515602
(Iteration 4101 / 4900) loss: 1.396509
(Iteration 4201 / 4900) loss: 1.585498
```

```
(Iteration 4301 / 4900) loss: 1.417389
(Iteration 4401 / 4900) loss: 1.736300
(Epoch 9 / 10) train acc: 0.491000; val_acc: 0.455000
(Iteration 4501 / 4900) loss: 1.745255
(Iteration 4601 / 4900) loss: 1.202904
(Iteration 4701 / 4900) loss: 1.564746
(Iteration 4801 / 4900) loss: 1.419656
(Epoch 10 / 10) train acc: 0.491000; val_acc: 0.457000
(Iteration 1 / 4900) loss: 2.304390
(Epoch 0 / 10) train acc: 0.160000; val_acc: 0.167000
(Iteration 101 / 4900) loss: 2.038511
(Iteration 201 / 4900) loss: 1.885331
(Iteration 301 / 4900) loss: 2.263874
(Iteration 401 / 4900) loss: 1.791421
(Epoch 1 / 10) train acc: 0.354000; val_acc: 0.329000
(Iteration 501 / 4900) loss: 1.855531
(Iteration 601 / 4900) loss: 3.203121
(Iteration 701 / 4900) loss: 1.730827
(Iteration 801 / 4900) loss: 3.235753
(Iteration 901 / 4900) loss: 1.749649
(Epoch 2 / 10) train acc: 0.328000; val_acc: 0.311000
(Iteration 1001 / 4900) loss: 1.587497
(Iteration 1101 / 4900) loss: 1.740966
(Iteration 1201 / 4900) loss: 1.959966
(Iteration 1301 / 4900) loss: 2.853451
(Iteration 1401 / 4900) loss: 2.715777
(Epoch 3 / 10) train acc: 0.300000; val_acc: 0.286000
(Iteration 1501 / 4900) loss: 1.600983
(Iteration 1601 / 4900) loss: 1.878576
(Iteration 1701 / 4900) loss: 1.780932
(Iteration 1801 / 4900) loss: 2.280329
(Iteration 1901 / 4900) loss: 2.174110
(Epoch 4 / 10) train acc: 0.377000; val_acc: 0.395000
(Iteration 2001 / 4900) loss: 1.799959
(Iteration 2101 / 4900) loss: 2.919886
(Iteration 2201 / 4900) loss: 1.673813
(Iteration 2301 / 4900) loss: 1.606889
(Iteration 2401 / 4900) loss: 1.684546
(Epoch 5 / 10) train acc: 0.423000; val_acc: 0.379000
(Iteration 2501 / 4900) loss: 1.559342
(Iteration 2601 / 4900) loss: 1.967050
(Iteration 2701 / 4900) loss: 1.681152
(Iteration 2801 / 4900) loss: 1.450395
(Iteration 2901 / 4900) loss: 1.470042
(Epoch 6 / 10) train acc: 0.430000; val_acc: 0.404000
(Iteration 3001 / 4900) loss: 1.655618
(Iteration 3101 / 4900) loss: 1.484590
(Iteration 3201 / 4900) loss: 1.465274
```

```
(Iteration 3301 / 4900) loss: 1.509437
(Iteration 3401 / 4900) loss: 1.523482
(Epoch 7 / 10) train acc: 0.449000; val_acc: 0.386000
(Iteration 3501 / 4900) loss: 1.312494
(Iteration 3601 / 4900) loss: 1.793749
(Iteration 3701 / 4900) loss: 1.579906
(Iteration 3801 / 4900) loss: 1.660607
(Iteration 3901 / 4900) loss: 1.063238
(Epoch 8 / 10) train acc: 0.489000; val_acc: 0.452000
(Iteration 4001 / 4900) loss: 1.753595
(Iteration 4101 / 4900) loss: 1.735396
(Iteration 4201 / 4900) loss: 1.511912
(Iteration 4301 / 4900) loss: 1.457650
(Iteration 4401 / 4900) loss: 1.554647
(Epoch 9 / 10) train acc: 0.468000; val_acc: 0.433000
(Iteration 4501 / 4900) loss: 1.403935
(Iteration 4601 / 4900) loss: 1.206442
(Iteration 4701 / 4900) loss: 1.542194
(Iteration 4801 / 4900) loss: 1.472400
(Epoch 10 / 10) train acc: 0.431000; val_acc: 0.377000
(Iteration 1 / 4900) loss: 2.296728
(Epoch 0 / 10) train acc: 0.159000; val_acc: 0.161000
(Iteration 101 / 4900) loss: 1.946472
(Iteration 201 / 4900) loss: 1.843096
(Iteration 301 / 4900) loss: 1.793935
(Iteration 401 / 4900) loss: 1.856526
(Epoch 1 / 10) train acc: 0.337000; val_acc: 0.355000
(Iteration 501 / 4900) loss: 1.651961
(Iteration 601 / 4900) loss: 2.086303
(Iteration 701 / 4900) loss: 2.209022
(Iteration 801 / 4900) loss: 2.351192
(Iteration 901 / 4900) loss: 2.363333
(Epoch 2 / 10) train acc: 0.386000; val_acc: 0.352000
(Iteration 1001 / 4900) loss: 1.726074
(Iteration 1101 / 4900) loss: 1.539406
(Iteration 1201 / 4900) loss: 1.877664
(Iteration 1301 / 4900) loss: 1.989290
(Iteration 1401 / 4900) loss: 1.745633
(Epoch 3 / 10) train acc: 0.381000; val_acc: 0.355000
(Iteration 1501 / 4900) loss: 1.653286
(Iteration 1601 / 4900) loss: 2.411477
(Iteration 1701 / 4900) loss: 1.802680
(Iteration 1801 / 4900) loss: 1.758461
(Iteration 1901 / 4900) loss: 1.755498
(Epoch 4 / 10) train acc: 0.422000; val_acc: 0.384000
(Iteration 2001 / 4900) loss: 1.708149
(Iteration 2101 / 4900) loss: 1.564511
(Iteration 2201 / 4900) loss: 1.686197
```

```
(Iteration 2301 / 4900) loss: 1.710942
(Iteration 2401 / 4900) loss: 1.703469
(Epoch 5 / 10) train acc: 0.380000; val_acc: 0.359000
(Iteration 2501 / 4900) loss: 2.497693
(Iteration 2601 / 4900) loss: 1.641030
(Iteration 2701 / 4900) loss: 1.646606
(Iteration 2801 / 4900) loss: 1.365810
(Iteration 2901 / 4900) loss: 1.658106
(Epoch 6 / 10) train acc: 0.393000; val_acc: 0.347000
(Iteration 3001 / 4900) loss: 1.373584
(Iteration 3101 / 4900) loss: 1.197586
(Iteration 3201 / 4900) loss: 1.504555
(Iteration 3301 / 4900) loss: 1.557452
(Iteration 3401 / 4900) loss: 1.771105
(Epoch 7 / 10) train acc: 0.487000; val_acc: 0.431000
(Iteration 3501 / 4900) loss: 1.534493
(Iteration 3601 / 4900) loss: 1.401057
(Iteration 3701 / 4900) loss: 2.081837
(Iteration 3801 / 4900) loss: 1.324754
(Iteration 3901 / 4900) loss: 1.670669
(Epoch 8 / 10) train acc: 0.469000; val_acc: 0.438000
(Iteration 4001 / 4900) loss: 1.616466
(Iteration 4101 / 4900) loss: 1.361134
(Iteration 4201 / 4900) loss: 1.568589
(Iteration 4301 / 4900) loss: 1.704412
(Iteration 4401 / 4900) loss: 1.504902
(Epoch 9 / 10) train acc: 0.448000; val_acc: 0.399000
(Iteration 4501 / 4900) loss: 1.659839
(Iteration 4601 / 4900) loss: 1.367128
(Iteration 4701 / 4900) loss: 1.332496
(Iteration 4801 / 4900) loss: 1.794283
(Epoch 10 / 10) train acc: 0.498000; val_acc: 0.436000
(Iteration 1 / 4900) loss: 2.304678
(Epoch 0 / 10) train acc: 0.125000; val_acc: 0.123000
(Iteration 101 / 4900) loss: 2.072663
(Iteration 201 / 4900) loss: 1.900716
(Iteration 301 / 4900) loss: 1.834875
(Iteration 401 / 4900) loss: 3.265774
(Epoch 1 / 10) train acc: 0.333000; val_acc: 0.355000
(Iteration 501 / 4900) loss: 1.719888
(Iteration 601 / 4900) loss: 2.065430
(Iteration 701 / 4900) loss: 2.615474
(Iteration 801 / 4900) loss: 1.746472
(Iteration 901 / 4900) loss: 2.131590
(Epoch 2 / 10) train acc: 0.379000; val_acc: 0.376000
(Iteration 1001 / 4900) loss: 1.583737
(Iteration 1101 / 4900) loss: 2.329675
(Iteration 1201 / 4900) loss: 2.302075
```

```
(Iteration 1301 / 4900) loss: 1.802240
(Iteration 1401 / 4900) loss: 2.274936
(Epoch 3 / 10) train acc: 0.385000; val_acc: 0.412000
(Iteration 1501 / 4900) loss: 2.020589
(Iteration 1601 / 4900) loss: 2.113667
(Iteration 1701 / 4900) loss: 2.078151
(Iteration 1801 / 4900) loss: 1.800309
(Iteration 1901 / 4900) loss: 1.798945
(Epoch 4 / 10) train acc: 0.319000; val_acc: 0.279000
(Iteration 2001 / 4900) loss: 1.704897
(Iteration 2101 / 4900) loss: 1.901129
(Iteration 2201 / 4900) loss: 1.818330
(Iteration 2301 / 4900) loss: 1.823035
(Iteration 2401 / 4900) loss: 1.739809
(Epoch 5 / 10) train acc: 0.387000; val_acc: 0.386000
(Iteration 2501 / 4900) loss: 2.191871
(Iteration 2601 / 4900) loss: 1.655954
(Iteration 2701 / 4900) loss: 1.395673
(Iteration 2801 / 4900) loss: 1.812533
(Iteration 2901 / 4900) loss: 2.186493
(Epoch 6 / 10) train acc: 0.360000; val_acc: 0.312000
(Iteration 3001 / 4900) loss: 1.523929
(Iteration 3101 / 4900) loss: 1.551965
(Iteration 3201 / 4900) loss: 1.788125
(Iteration 3301 / 4900) loss: 1.666079
(Iteration 3401 / 4900) loss: 1.633065
(Epoch 7 / 10) train acc: 0.433000; val_acc: 0.395000
(Iteration 3501 / 4900) loss: 1.602623
(Iteration 3601 / 4900) loss: 1.657126
(Iteration 3701 / 4900) loss: 1.887193
(Iteration 3801 / 4900) loss: 1.719448
(Iteration 3901 / 4900) loss: 1.363195
(Epoch 8 / 10) train acc: 0.484000; val_acc: 0.440000
(Iteration 4001 / 4900) loss: 1.343153
(Iteration 4101 / 4900) loss: 1.622644
(Iteration 4201 / 4900) loss: 1.414973
(Iteration 4301 / 4900) loss: 2.020662
(Iteration 4401 / 4900) loss: 1.566603
(Epoch 9 / 10) train acc: 0.441000; val_acc: 0.426000
(Iteration 4501 / 4900) loss: 1.187798
(Iteration 4601 / 4900) loss: 1.614275
(Iteration 4701 / 4900) loss: 1.581674
(Iteration 4801 / 4900) loss: 1.482768
(Epoch 10 / 10) train acc: 0.536000; val_acc: 0.445000
(Iteration 1 / 4900) loss: 2.298092
(Epoch 0 / 10) train acc: 0.163000; val_acc: 0.145000
```

```
/content/drive/My Drive/ENPM809K/assignment1/assignment1/cs231n/layers.py:238:
```

```
RuntimeWarning: divide by zero encountered in log
  loss = -np.sum(np.log(scores[np.arange(num_train), y])) / num_train
```

```
(Iteration 101 / 4900) loss: inf
```

```
/content/drive/My Drive/ENPM809K/assignment1/assignment1/cs231n/layers.py:36:
RuntimeWarning: overflow encountered in matmul
  out = np.matmul(X,w) + b
/content/drive/My Drive/ENPM809K/assignment1/assignment1/cs231n/layers.py:232:
RuntimeWarning: overflow encountered in subtract
  scores = np.exp(x - np.max(x, axis=1, keepdims=True)) #numerator
/content/drive/My Drive/ENPM809K/assignment1/assignment1/cs231n/layers.py:241:
RuntimeWarning: overflow encountered in subtract
  #dx = scores.copy()
/content/drive/My Drive/ENPM809K/assignment1/assignment1/cs231n/layers.py:232:
RuntimeWarning: invalid value encountered in subtract
  scores = np.exp(x - np.max(x, axis=1, keepdims=True)) #numerator
/content/drive/My Drive/ENPM809K/assignment1/assignment1/cs231n/layers.py:241:
RuntimeWarning: invalid value encountered in subtract
  #dx = scores.copy()
```

```
(Iteration 201 / 4900) loss: nan
(Iteration 301 / 4900) loss: nan
(Iteration 401 / 4900) loss: nan
(Epoch 1 / 10) train acc: 0.087000; val_acc: 0.087000
(Iteration 501 / 4900) loss: nan
(Iteration 601 / 4900) loss: nan
(Iteration 701 / 4900) loss: nan
(Iteration 801 / 4900) loss: nan
(Iteration 901 / 4900) loss: nan
(Epoch 2 / 10) train acc: 0.086000; val_acc: 0.087000
(Iteration 1001 / 4900) loss: nan
(Iteration 1101 / 4900) loss: nan
(Iteration 1201 / 4900) loss: nan
(Iteration 1301 / 4900) loss: nan
(Iteration 1401 / 4900) loss: nan
(Epoch 3 / 10) train acc: 0.095000; val_acc: 0.087000
(Iteration 1501 / 4900) loss: nan
(Iteration 1601 / 4900) loss: nan
(Iteration 1701 / 4900) loss: nan
(Iteration 1801 / 4900) loss: nan
(Iteration 1901 / 4900) loss: nan
(Epoch 4 / 10) train acc: 0.097000; val_acc: 0.087000
(Iteration 2001 / 4900) loss: nan
(Iteration 2101 / 4900) loss: nan
(Iteration 2201 / 4900) loss: nan
(Iteration 2301 / 4900) loss: nan
(Iteration 2401 / 4900) loss: nan
(Epoch 5 / 10) train acc: 0.119000; val_acc: 0.087000
```

```
(Iteration 2501 / 4900) loss: nan
(Iteration 2601 / 4900) loss: nan
(Iteration 2701 / 4900) loss: nan
(Iteration 2801 / 4900) loss: nan
(Iteration 2901 / 4900) loss: nan
(Epoch 6 / 10) train acc: 0.098000; val_acc: 0.087000
(Iteration 3001 / 4900) loss: nan
(Iteration 3101 / 4900) loss: nan
(Iteration 3201 / 4900) loss: nan
(Iteration 3301 / 4900) loss: nan
(Iteration 3401 / 4900) loss: nan
(Epoch 7 / 10) train acc: 0.104000; val_acc: 0.087000
(Iteration 3501 / 4900) loss: nan
(Iteration 3601 / 4900) loss: nan
(Iteration 3701 / 4900) loss: nan
(Iteration 3801 / 4900) loss: nan
(Iteration 3901 / 4900) loss: nan
(Epoch 8 / 10) train acc: 0.098000; val_acc: 0.087000
(Iteration 4001 / 4900) loss: nan
(Iteration 4101 / 4900) loss: nan
(Iteration 4201 / 4900) loss: nan
(Iteration 4301 / 4900) loss: nan
(Iteration 4401 / 4900) loss: nan
(Epoch 9 / 10) train acc: 0.107000; val_acc: 0.087000
(Iteration 4501 / 4900) loss: nan
(Iteration 4601 / 4900) loss: nan
(Iteration 4701 / 4900) loss: nan
(Iteration 4801 / 4900) loss: nan
(Epoch 10 / 10) train acc: 0.092000; val_acc: 0.087000
(Iteration 1 / 4900) loss: 2.306943
(Epoch 0 / 10) train acc: 0.155000; val_acc: 0.169000
(Iteration 101 / 4900) loss: inf
(Iteration 201 / 4900) loss: nan
(Iteration 301 / 4900) loss: nan
(Iteration 401 / 4900) loss: nan
(Epoch 1 / 10) train acc: 0.118000; val_acc: 0.087000
(Iteration 501 / 4900) loss: nan
(Iteration 601 / 4900) loss: nan
(Iteration 701 / 4900) loss: nan
(Iteration 801 / 4900) loss: nan
(Iteration 901 / 4900) loss: nan
(Epoch 2 / 10) train acc: 0.115000; val_acc: 0.087000
(Iteration 1001 / 4900) loss: nan
(Iteration 1101 / 4900) loss: nan
(Iteration 1201 / 4900) loss: nan
(Iteration 1301 / 4900) loss: nan
(Iteration 1401 / 4900) loss: nan
(Epoch 3 / 10) train acc: 0.106000; val_acc: 0.087000
```

```
(Iteration 1501 / 4900) loss: nan
(Iteration 1601 / 4900) loss: nan
(Iteration 1701 / 4900) loss: nan
(Iteration 1801 / 4900) loss: nan
(Iteration 1901 / 4900) loss: nan
(Epoch 4 / 10) train acc: 0.107000; val_acc: 0.087000
(Iteration 2001 / 4900) loss: nan
(Iteration 2101 / 4900) loss: nan
(Iteration 2201 / 4900) loss: nan
(Iteration 2301 / 4900) loss: nan
(Iteration 2401 / 4900) loss: nan
(Epoch 5 / 10) train acc: 0.103000; val_acc: 0.087000
(Iteration 2501 / 4900) loss: nan
(Iteration 2601 / 4900) loss: nan
(Iteration 2701 / 4900) loss: nan
(Iteration 2801 / 4900) loss: nan
(Iteration 2901 / 4900) loss: nan
(Epoch 6 / 10) train acc: 0.097000; val_acc: 0.087000
(Iteration 3001 / 4900) loss: nan
(Iteration 3101 / 4900) loss: nan
(Iteration 3201 / 4900) loss: nan
(Iteration 3301 / 4900) loss: nan
(Iteration 3401 / 4900) loss: nan
(Epoch 7 / 10) train acc: 0.100000; val_acc: 0.087000
(Iteration 3501 / 4900) loss: nan
(Iteration 3601 / 4900) loss: nan
(Iteration 3701 / 4900) loss: nan
(Iteration 3801 / 4900) loss: nan
(Iteration 3901 / 4900) loss: nan
(Epoch 8 / 10) train acc: 0.106000; val_acc: 0.087000
(Iteration 4001 / 4900) loss: nan
(Iteration 4101 / 4900) loss: nan
(Iteration 4201 / 4900) loss: nan
(Iteration 4301 / 4900) loss: nan
(Iteration 4401 / 4900) loss: nan
(Epoch 9 / 10) train acc: 0.114000; val_acc: 0.087000
(Iteration 4501 / 4900) loss: nan
(Iteration 4601 / 4900) loss: nan
(Iteration 4701 / 4900) loss: nan
(Iteration 4801 / 4900) loss: nan
(Epoch 10 / 10) train acc: 0.099000; val_acc: 0.087000
(Iteration 1 / 4900) loss: 2.300245
(Epoch 0 / 10) train acc: 0.092000; val_acc: 0.096000
(Iteration 101 / 4900) loss: inf
(Iteration 201 / 4900) loss: nan
(Iteration 301 / 4900) loss: nan
(Iteration 401 / 4900) loss: nan
(Epoch 1 / 10) train acc: 0.100000; val_acc: 0.087000
```

```
(Iteration 501 / 4900) loss: nan
(Iteration 601 / 4900) loss: nan
(Iteration 701 / 4900) loss: nan
(Iteration 801 / 4900) loss: nan
(Iteration 901 / 4900) loss: nan
(Epoch 2 / 10) train acc: 0.087000; val_acc: 0.087000
(Iteration 1001 / 4900) loss: nan
(Iteration 1101 / 4900) loss: nan
(Iteration 1201 / 4900) loss: nan
(Iteration 1301 / 4900) loss: nan
(Iteration 1401 / 4900) loss: nan
(Epoch 3 / 10) train acc: 0.110000; val_acc: 0.087000
(Iteration 1501 / 4900) loss: nan
(Iteration 1601 / 4900) loss: nan
(Iteration 1701 / 4900) loss: nan
(Iteration 1801 / 4900) loss: nan
(Iteration 1901 / 4900) loss: nan
(Epoch 4 / 10) train acc: 0.115000; val_acc: 0.087000
(Iteration 2001 / 4900) loss: nan
(Iteration 2101 / 4900) loss: nan
(Iteration 2201 / 4900) loss: nan
(Iteration 2301 / 4900) loss: nan
(Iteration 2401 / 4900) loss: nan
(Epoch 5 / 10) train acc: 0.104000; val_acc: 0.087000
(Iteration 2501 / 4900) loss: nan
(Iteration 2601 / 4900) loss: nan
(Iteration 2701 / 4900) loss: nan
(Iteration 2801 / 4900) loss: nan
(Iteration 2901 / 4900) loss: nan
(Epoch 6 / 10) train acc: 0.118000; val_acc: 0.087000
(Iteration 3001 / 4900) loss: nan
(Iteration 3101 / 4900) loss: nan
(Iteration 3201 / 4900) loss: nan
(Iteration 3301 / 4900) loss: nan
(Iteration 3401 / 4900) loss: nan
(Epoch 7 / 10) train acc: 0.095000; val_acc: 0.087000
(Iteration 3501 / 4900) loss: nan
(Iteration 3601 / 4900) loss: nan
(Iteration 3701 / 4900) loss: nan
(Iteration 3801 / 4900) loss: nan
(Iteration 3901 / 4900) loss: nan
(Epoch 8 / 10) train acc: 0.095000; val_acc: 0.087000
(Iteration 4001 / 4900) loss: nan
(Iteration 4101 / 4900) loss: nan
(Iteration 4201 / 4900) loss: nan
(Iteration 4301 / 4900) loss: nan
(Iteration 4401 / 4900) loss: nan
(Epoch 9 / 10) train acc: 0.093000; val_acc: 0.087000
```

```
(Iteration 4501 / 4900) loss: nan
(Iteration 4601 / 4900) loss: nan
(Iteration 4701 / 4900) loss: nan
(Iteration 4801 / 4900) loss: nan
(Epoch 10 / 10) train acc: 0.091000; val_acc: 0.087000
(Iteration 1 / 4900) loss: 2.307130
(Epoch 0 / 10) train acc: 0.130000; val_acc: 0.149000
(Iteration 101 / 4900) loss: inf
(Iteration 201 / 4900) loss: nan
(Iteration 301 / 4900) loss: nan
(Iteration 401 / 4900) loss: nan
(Epoch 1 / 10) train acc: 0.103000; val_acc: 0.087000
(Iteration 501 / 4900) loss: nan
(Iteration 601 / 4900) loss: nan
(Iteration 701 / 4900) loss: nan
(Iteration 801 / 4900) loss: nan
(Iteration 901 / 4900) loss: nan
(Epoch 2 / 10) train acc: 0.110000; val_acc: 0.087000
(Iteration 1001 / 4900) loss: nan
(Iteration 1101 / 4900) loss: nan
(Iteration 1201 / 4900) loss: nan
(Iteration 1301 / 4900) loss: nan
(Iteration 1401 / 4900) loss: nan
(Epoch 3 / 10) train acc: 0.096000; val_acc: 0.087000
(Iteration 1501 / 4900) loss: nan
(Iteration 1601 / 4900) loss: nan
(Iteration 1701 / 4900) loss: nan
(Iteration 1801 / 4900) loss: nan
(Iteration 1901 / 4900) loss: nan
(Epoch 4 / 10) train acc: 0.103000; val_acc: 0.087000
(Iteration 2001 / 4900) loss: nan
(Iteration 2101 / 4900) loss: nan
(Iteration 2201 / 4900) loss: nan
(Iteration 2301 / 4900) loss: nan
(Iteration 2401 / 4900) loss: nan
(Epoch 5 / 10) train acc: 0.102000; val_acc: 0.087000
(Iteration 2501 / 4900) loss: nan
(Iteration 2601 / 4900) loss: nan
(Iteration 2701 / 4900) loss: nan
(Iteration 2801 / 4900) loss: nan
(Iteration 2901 / 4900) loss: nan
(Epoch 6 / 10) train acc: 0.099000; val_acc: 0.087000
(Iteration 3001 / 4900) loss: nan
(Iteration 3101 / 4900) loss: nan
(Iteration 3201 / 4900) loss: nan
(Iteration 3301 / 4900) loss: nan
(Iteration 3401 / 4900) loss: nan
(Epoch 7 / 10) train acc: 0.085000; val_acc: 0.087000
```

```
(Iteration 3501 / 4900) loss: nan
(Iteration 3601 / 4900) loss: nan
(Iteration 3701 / 4900) loss: nan
(Iteration 3801 / 4900) loss: nan
(Iteration 3901 / 4900) loss: nan
(Epoch 8 / 10) train acc: 0.092000; val_acc: 0.087000
(Iteration 4001 / 4900) loss: nan
(Iteration 4101 / 4900) loss: nan
(Iteration 4201 / 4900) loss: nan
(Iteration 4301 / 4900) loss: nan
(Iteration 4401 / 4900) loss: nan
(Epoch 9 / 10) train acc: 0.119000; val_acc: 0.087000
(Iteration 4501 / 4900) loss: nan
(Iteration 4601 / 4900) loss: nan
(Iteration 4701 / 4900) loss: nan
(Iteration 4801 / 4900) loss: nan
(Epoch 10 / 10) train acc: 0.122000; val_acc: 0.087000
(Iteration 1 / 4900) loss: 2.304758
(Epoch 0 / 10) train acc: 0.136000; val_acc: 0.129000
(Iteration 101 / 4900) loss: 2.223126
(Iteration 201 / 4900) loss: 2.078230
(Iteration 301 / 4900) loss: 2.010792
(Iteration 401 / 4900) loss: 1.864607
(Epoch 1 / 10) train acc: 0.308000; val_acc: 0.315000
(Iteration 501 / 4900) loss: 1.922179
(Iteration 601 / 4900) loss: 1.956725
(Iteration 701 / 4900) loss: 1.914176
(Iteration 801 / 4900) loss: 1.742278
(Iteration 901 / 4900) loss: 1.782953
(Epoch 2 / 10) train acc: 0.383000; val_acc: 0.382000
(Iteration 1001 / 4900) loss: 1.805279
(Iteration 1101 / 4900) loss: 1.745999
(Iteration 1201 / 4900) loss: 1.592096
(Iteration 1301 / 4900) loss: 1.690957
(Iteration 1401 / 4900) loss: 1.662429
(Epoch 3 / 10) train acc: 0.410000; val_acc: 0.404000
(Iteration 1501 / 4900) loss: 1.678729
(Iteration 1601 / 4900) loss: 1.791128
(Iteration 1701 / 4900) loss: 1.515391
(Iteration 1801 / 4900) loss: 1.626185
(Iteration 1901 / 4900) loss: 1.595610
(Epoch 4 / 10) train acc: 0.415000; val_acc: 0.439000
(Iteration 2001 / 4900) loss: 1.642055
(Iteration 2101 / 4900) loss: 1.699875
(Iteration 2201 / 4900) loss: 1.775241
(Iteration 2301 / 4900) loss: 1.682254
(Iteration 2401 / 4900) loss: 1.514390
(Epoch 5 / 10) train acc: 0.410000; val_acc: 0.453000
```

```
(Iteration 2501 / 4900) loss: 1.794629
(Iteration 2601 / 4900) loss: 1.574521
(Iteration 2701 / 4900) loss: 1.485470
(Iteration 2801 / 4900) loss: 1.549082
(Iteration 2901 / 4900) loss: 1.511934
(Epoch 6 / 10) train acc: 0.449000; val_acc: 0.457000
(Iteration 3001 / 4900) loss: 1.636535
(Iteration 3101 / 4900) loss: 1.387115
(Iteration 3201 / 4900) loss: 1.511933
(Iteration 3301 / 4900) loss: 1.621381
(Iteration 3401 / 4900) loss: 1.641809
(Epoch 7 / 10) train acc: 0.442000; val_acc: 0.466000
(Iteration 3501 / 4900) loss: 1.451504
(Iteration 3601 / 4900) loss: 1.547168
(Iteration 3701 / 4900) loss: 1.673798
(Iteration 3801 / 4900) loss: 1.488817
(Iteration 3901 / 4900) loss: 1.396684
(Epoch 8 / 10) train acc: 0.464000; val_acc: 0.467000
(Iteration 4001 / 4900) loss: 1.670077
(Iteration 4101 / 4900) loss: 1.494393
(Iteration 4201 / 4900) loss: 1.599193
(Iteration 4301 / 4900) loss: 1.485654
(Iteration 4401 / 4900) loss: 1.564264
(Epoch 9 / 10) train acc: 0.466000; val_acc: 0.467000
(Iteration 4501 / 4900) loss: 1.535453
(Iteration 4601 / 4900) loss: 1.447870
(Iteration 4701 / 4900) loss: 1.463869
(Iteration 4801 / 4900) loss: 1.430009
(Epoch 10 / 10) train acc: 0.470000; val_acc: 0.472000
(Iteration 1 / 4900) loss: 2.300228
(Epoch 0 / 10) train acc: 0.113000; val_acc: 0.115000
(Iteration 101 / 4900) loss: 2.216721
(Iteration 201 / 4900) loss: 2.113766
(Iteration 301 / 4900) loss: 2.005158
(Iteration 401 / 4900) loss: 1.929637
(Epoch 1 / 10) train acc: 0.331000; val_acc: 0.315000
(Iteration 501 / 4900) loss: 1.872320
(Iteration 601 / 4900) loss: 1.866983
(Iteration 701 / 4900) loss: 1.798934
(Iteration 801 / 4900) loss: 1.841603
(Iteration 901 / 4900) loss: 1.753169
(Epoch 2 / 10) train acc: 0.369000; val_acc: 0.365000
(Iteration 1001 / 4900) loss: 1.849599
(Iteration 1101 / 4900) loss: 1.596789
(Iteration 1201 / 4900) loss: 1.693240
(Iteration 1301 / 4900) loss: 1.785053
(Iteration 1401 / 4900) loss: 1.684652
(Epoch 3 / 10) train acc: 0.410000; val_acc: 0.402000
```

```
(Iteration 1501 / 4900) loss: 1.720840
(Iteration 1601 / 4900) loss: 1.652196
(Iteration 1701 / 4900) loss: 1.613290
(Iteration 1801 / 4900) loss: 1.782301
(Iteration 1901 / 4900) loss: 1.540895
(Epoch 4 / 10) train acc: 0.456000; val_acc: 0.433000
(Iteration 2001 / 4900) loss: 1.630145
(Iteration 2101 / 4900) loss: 1.653773
(Iteration 2201 / 4900) loss: 1.610843
(Iteration 2301 / 4900) loss: 1.677977
(Iteration 2401 / 4900) loss: 1.439042
(Epoch 5 / 10) train acc: 0.441000; val_acc: 0.452000
(Iteration 2501 / 4900) loss: 1.647404
(Iteration 2601 / 4900) loss: 1.611867
(Iteration 2701 / 4900) loss: 1.621616
(Iteration 2801 / 4900) loss: 1.606220
(Iteration 2901 / 4900) loss: 1.506740
(Epoch 6 / 10) train acc: 0.444000; val_acc: 0.447000
(Iteration 3001 / 4900) loss: 1.665328
(Iteration 3101 / 4900) loss: 1.535739
(Iteration 3201 / 4900) loss: 1.645629
(Iteration 3301 / 4900) loss: 1.429707
(Iteration 3401 / 4900) loss: 1.511601
(Epoch 7 / 10) train acc: 0.450000; val_acc: 0.459000
(Iteration 3501 / 4900) loss: 1.578318
(Iteration 3601 / 4900) loss: 1.483252
(Iteration 3701 / 4900) loss: 1.343452
(Iteration 3801 / 4900) loss: 1.667896
(Iteration 3901 / 4900) loss: 1.460921
(Epoch 8 / 10) train acc: 0.456000; val_acc: 0.468000
(Iteration 4001 / 4900) loss: 1.554044
(Iteration 4101 / 4900) loss: 1.604686
(Iteration 4201 / 4900) loss: 1.563902
(Iteration 4301 / 4900) loss: 1.470168
(Iteration 4401 / 4900) loss: 1.524675
(Epoch 9 / 10) train acc: 0.482000; val_acc: 0.456000
(Iteration 4501 / 4900) loss: 1.429106
(Iteration 4601 / 4900) loss: 1.538824
(Iteration 4701 / 4900) loss: 1.698398
(Iteration 4801 / 4900) loss: 1.440547
(Epoch 10 / 10) train acc: 0.511000; val_acc: 0.467000
(Iteration 1 / 4900) loss: 2.305479
(Epoch 0 / 10) train acc: 0.090000; val_acc: 0.108000
(Iteration 101 / 4900) loss: 2.236068
(Iteration 201 / 4900) loss: 2.066004
(Iteration 301 / 4900) loss: 1.997241
(Iteration 401 / 4900) loss: 1.957149
(Epoch 1 / 10) train acc: 0.335000; val_acc: 0.330000
```

```
(Iteration 501 / 4900) loss: 1.987671
(Iteration 601 / 4900) loss: 1.940614
(Iteration 701 / 4900) loss: 1.801880
(Iteration 801 / 4900) loss: 1.762915
(Iteration 901 / 4900) loss: 1.710307
(Epoch 2 / 10) train acc: 0.367000; val_acc: 0.391000
(Iteration 1001 / 4900) loss: 1.819662
(Iteration 1101 / 4900) loss: 1.792777
(Iteration 1201 / 4900) loss: 1.639426
(Iteration 1301 / 4900) loss: 1.662824
(Iteration 1401 / 4900) loss: 1.582918
(Epoch 3 / 10) train acc: 0.387000; val_acc: 0.418000
(Iteration 1501 / 4900) loss: 1.791390
(Iteration 1601 / 4900) loss: 1.742085
(Iteration 1701 / 4900) loss: 1.778860
(Iteration 1801 / 4900) loss: 1.643810
(Iteration 1901 / 4900) loss: 1.586224
(Epoch 4 / 10) train acc: 0.424000; val_acc: 0.435000
(Iteration 2001 / 4900) loss: 1.736672
(Iteration 2101 / 4900) loss: 1.612642
(Iteration 2201 / 4900) loss: 1.810753
(Iteration 2301 / 4900) loss: 1.588428
(Iteration 2401 / 4900) loss: 1.641517
(Epoch 5 / 10) train acc: 0.444000; val_acc: 0.450000
(Iteration 2501 / 4900) loss: 1.561753
(Iteration 2601 / 4900) loss: 1.620253
(Iteration 2701 / 4900) loss: 1.485298
(Iteration 2801 / 4900) loss: 1.646596
(Iteration 2901 / 4900) loss: 1.645215
(Epoch 6 / 10) train acc: 0.445000; val_acc: 0.456000
(Iteration 3001 / 4900) loss: 1.543519
(Iteration 3101 / 4900) loss: 1.632578
(Iteration 3201 / 4900) loss: 1.504950
(Iteration 3301 / 4900) loss: 1.331146
(Iteration 3401 / 4900) loss: 1.348330
(Epoch 7 / 10) train acc: 0.480000; val_acc: 0.459000
(Iteration 3501 / 4900) loss: 1.419616
(Iteration 3601 / 4900) loss: 1.657689
(Iteration 3701 / 4900) loss: 1.481830
(Iteration 3801 / 4900) loss: 1.642483
(Iteration 3901 / 4900) loss: 1.635472
(Epoch 8 / 10) train acc: 0.455000; val_acc: 0.466000
(Iteration 4001 / 4900) loss: 1.443110
(Iteration 4101 / 4900) loss: 1.617428
(Iteration 4201 / 4900) loss: 1.472123
(Iteration 4301 / 4900) loss: 1.612798
(Iteration 4401 / 4900) loss: 1.518606
(Epoch 9 / 10) train acc: 0.462000; val_acc: 0.463000
```

```
(Iteration 4501 / 4900) loss: 1.444890
(Iteration 4601 / 4900) loss: 1.571176
(Iteration 4701 / 4900) loss: 1.501993
(Iteration 4801 / 4900) loss: 1.543162
(Epoch 10 / 10) train acc: 0.489000; val_acc: 0.459000
(Iteration 1 / 4900) loss: 2.305448
(Epoch 0 / 10) train acc: 0.076000; val_acc: 0.075000
(Iteration 101 / 4900) loss: 2.247428
(Iteration 201 / 4900) loss: 2.025084
(Iteration 301 / 4900) loss: 1.982769
(Iteration 401 / 4900) loss: 1.921144
(Epoch 1 / 10) train acc: 0.302000; val_acc: 0.317000
(Iteration 501 / 4900) loss: 1.899791
(Iteration 601 / 4900) loss: 1.877590
(Iteration 701 / 4900) loss: 2.014586
(Iteration 801 / 4900) loss: 1.913724
(Iteration 901 / 4900) loss: 1.934842
(Epoch 2 / 10) train acc: 0.372000; val_acc: 0.360000
(Iteration 1001 / 4900) loss: 1.778209
(Iteration 1101 / 4900) loss: 1.753290
(Iteration 1201 / 4900) loss: 1.645462
(Iteration 1301 / 4900) loss: 1.825807
(Iteration 1401 / 4900) loss: 1.794580
(Epoch 3 / 10) train acc: 0.391000; val_acc: 0.407000
(Iteration 1501 / 4900) loss: 1.790274
(Iteration 1601 / 4900) loss: 1.665759
(Iteration 1701 / 4900) loss: 1.703080
(Iteration 1801 / 4900) loss: 1.751070
(Iteration 1901 / 4900) loss: 1.577968
(Epoch 4 / 10) train acc: 0.396000; val_acc: 0.428000
(Iteration 2001 / 4900) loss: 1.722812
(Iteration 2101 / 4900) loss: 1.808069
(Iteration 2201 / 4900) loss: 1.619903
(Iteration 2301 / 4900) loss: 1.605196
(Iteration 2401 / 4900) loss: 1.505462
(Epoch 5 / 10) train acc: 0.437000; val_acc: 0.445000
(Iteration 2501 / 4900) loss: 1.653831
(Iteration 2601 / 4900) loss: 1.526251
(Iteration 2701 / 4900) loss: 1.592141
(Iteration 2801 / 4900) loss: 1.378183
(Iteration 2901 / 4900) loss: 1.507629
(Epoch 6 / 10) train acc: 0.428000; val_acc: 0.462000
(Iteration 3001 / 4900) loss: 1.611194
(Iteration 3101 / 4900) loss: 1.455304
(Iteration 3201 / 4900) loss: 1.429438
(Iteration 3301 / 4900) loss: 1.624573
(Iteration 3401 / 4900) loss: 1.577728
(Epoch 7 / 10) train acc: 0.458000; val_acc: 0.473000
```

```
(Iteration 3501 / 4900) loss: 1.485967
(Iteration 3601 / 4900) loss: 1.609769
(Iteration 3701 / 4900) loss: 1.458615
(Iteration 3801 / 4900) loss: 1.405736
(Iteration 3901 / 4900) loss: 1.484267
(Epoch 8 / 10) train acc: 0.463000; val_acc: 0.475000
(Iteration 4001 / 4900) loss: 1.398840
(Iteration 4101 / 4900) loss: 1.677998
(Iteration 4201 / 4900) loss: 1.544853
(Iteration 4301 / 4900) loss: 1.498721
(Iteration 4401 / 4900) loss: 1.434036
(Epoch 9 / 10) train acc: 0.498000; val_acc: 0.477000
(Iteration 4501 / 4900) loss: 1.584359
(Iteration 4601 / 4900) loss: 1.470731
(Iteration 4701 / 4900) loss: 1.370955
(Iteration 4801 / 4900) loss: 1.375111
(Epoch 10 / 10) train acc: 0.478000; val_acc: 0.480000
lr 1.000000e-04 reg : 0.000010 val accuracy: 0.472000
lr 1.000000e-04 reg : 0.000100 val accuracy: 0.468000
lr 1.000000e-04 reg : 0.001000 val accuracy: 0.466000
lr 1.000000e-04 reg : 0.010000 val accuracy: 0.480000
lr 3.000000e-04 reg : 0.000010 val accuracy: 0.505000
lr 3.000000e-04 reg : 0.000100 val accuracy: 0.508000
lr 3.000000e-04 reg : 0.001000 val accuracy: 0.518000
lr 3.000000e-04 reg : 0.010000 val accuracy: 0.527000
lr 3.000000e-03 reg : 0.000010 val accuracy: 0.457000
lr 3.000000e-03 reg : 0.000100 val accuracy: 0.452000
lr 3.000000e-03 reg : 0.001000 val accuracy: 0.438000
lr 3.000000e-03 reg : 0.010000 val accuracy: 0.445000
lr 3.000000e-02 reg : 0.000010 val accuracy: 0.145000
lr 3.000000e-02 reg : 0.000100 val accuracy: 0.169000
lr 3.000000e-02 reg : 0.001000 val accuracy: 0.096000
lr 3.000000e-02 reg : 0.010000 val accuracy: 0.149000
best validation accuracy achieved is: 0.527000
```

## 12 Test your model!

Run your best model on the validation and test sets. You should achieve above 48% accuracy on the validation set and the test set.

[50]:
```
y_val_pred = np.argmax(best_model.loss(data['X_val']), axis=1)
print('Validation set accuracy: ', (y_val_pred == data['y_val']).mean())
```

```
Validation set accuracy:  0.527
```

[51]:
```
y_test_pred = np.argmax(best_model.loss(data['X_test']), axis=1)
print('Test set accuracy: ', (y_test_pred == data['y_test']).mean())
```

```
Test set accuracy:  0.521
```

### 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.

$\mathit{Your Answer:}$ 1, 3

$\mathit{Your Explanation:}$ A situation where [training accuracy > testing accuracy] implies there is a problem of overfitting which is caused by high variance, where the network has learnt to overfit the training data and is therefore unable to create a good approximation for the test data points.

Hence, option 1 is a good solution to try. Adding more data to the training data points will lead to a better generalised approximation, and help solving this overfitting problem.

Option 2 is a bad choice, because as we increase the number of hidden units, the hypothesis function is more likely to learn the intricate features to fit the training data even more which defeats the purpose, which is decreasing the overfitting.

Option 3 might be a good option to try as increasing the regularization factor or strength leads to a more generalised or more approximated hypothesis which decreases the overfitting or high variance problems.

[ ]:
```

```

# features

October 9, 2022

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

# Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment1/'
FOLDERNAME = 'ENPM809K/assignment1/assignment1/'
assert FOLDERNAME is not None, "[!] Enter the foldername."

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

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

```
Mounted at /content/drive
/content/drive/My Drive/ENPM809K/assignment1/assignment1/cs231n/datasets
/content/drive/My Drive/ENPM809K/assignment1/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)*****


for l_r in learning_rates:
  for r_s in regularization_strengths:
    #creating svm instance
    svm=LinearSVM()
    loss=svm.train(X_train_feats, y_train, learning_rate=l_r, reg=r_s,
                   num_iters=1500, verbose=True)
    #predicting over the training and validation
    y_train_pred=svm.predict(X_train_feats)
    y_val_pred=svm.predict(X_val_feats)
    #Finding accuracy for training and validation by comparing labels
    train_accuracy=(np.mean(y_train == y_train_pred))
    val_accuracy=(np.mean(y_val == y_val_pred))
    #Storing accuracy in results dictionary
    results[(l_r,r_s)]=(train_accuracy,val_accuracy)
    #always compares and sets the best acuracy.
    if best_val<val_accuracy:
      best_val=val_accuracy
      best_svm=svm
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

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

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

```
iteration 0 / 1500: loss 89.234174
iteration 100 / 1500: loss 87.623140
iteration 200 / 1500: loss 86.068807
iteration 300 / 1500: loss 84.529311
iteration 400 / 1500: loss 83.057378
iteration 500 / 1500: loss 81.581365
iteration 600 / 1500: loss 80.132454
iteration 700 / 1500: loss 78.742381
iteration 800 / 1500: loss 77.366722
iteration 900 / 1500: loss 75.995625
iteration 1000 / 1500: loss 74.683620
iteration 1100 / 1500: loss 73.375396
iteration 1200 / 1500: loss 72.082977
iteration 1300 / 1500: loss 70.857310
iteration 1400 / 1500: loss 69.631357
iteration 0 / 1500: loss 766.925783
iteration 100 / 1500: loss 629.481006
iteration 200 / 1500: loss 516.947576
iteration 300 / 1500: loss 424.841144
iteration 400 / 1500: loss 349.428513
iteration 500 / 1500: loss 287.684797
iteration 600 / 1500: loss 237.144103
iteration 700 / 1500: loss 195.767702
iteration 800 / 1500: loss 161.901498
iteration 900 / 1500: loss 134.169802
iteration 1000 / 1500: loss 111.468900
iteration 1100 / 1500: loss 92.886611
iteration 1200 / 1500: loss 77.672153
iteration 1300 / 1500: loss 65.217616
iteration 1400 / 1500: loss 55.028252
iteration 0 / 1500: loss 7632.826210
iteration 100 / 1500: loss 1030.435849
iteration 200 / 1500: loss 145.852286
iteration 300 / 1500: loss 27.335094
iteration 400 / 1500: loss 11.456506
iteration 500 / 1500: loss 9.329150
iteration 600 / 1500: loss 9.044084
iteration 700 / 1500: loss 9.005906
iteration 800 / 1500: loss 9.000788
iteration 900 / 1500: loss 9.000101
iteration 1000 / 1500: loss 9.000011
iteration 1100 / 1500: loss 8.999998
iteration 1200 / 1500: loss 8.999998
iteration 1300 / 1500: loss 8.999997
iteration 1400 / 1500: loss 8.999997
iteration 0 / 1500: loss 85.742146
iteration 100 / 1500: loss 71.829841
iteration 200 / 1500: loss 60.418958
iteration 300 / 1500: loss 51.102320
iteration 400 / 1500: loss 43.467206
iteration 500 / 1500: loss 37.223292
iteration 600 / 1500: loss 32.099579
iteration 700 / 1500: loss 27.904699
iteration 800 / 1500: loss 24.470388
iteration 900 / 1500: loss 21.671952
iteration 1000 / 1500: loss 19.376906
iteration 1100 / 1500: loss 17.491799
iteration 1200 / 1500: loss 15.953563
iteration 1300 / 1500: loss 14.691277
iteration 1400 / 1500: loss 13.662551
iteration 0 / 1500: loss 760.953912
iteration 100 / 1500: loss 109.743445
iteration 200 / 1500: loss 22.496614
iteration 300 / 1500: loss 10.807968
iteration 400 / 1500: loss 9.242360
iteration 500 / 1500: loss 9.032386
iteration 600 / 1500: loss 9.004324
iteration 700 / 1500: loss 9.000545
iteration 800 / 1500: loss 9.000044
iteration 900 / 1500: loss 8.999976
iteration 1000 / 1500: loss 8.999966
iteration 1100 / 1500: loss 8.999964
iteration 1200 / 1500: loss 8.999974
iteration 1300 / 1500: loss 8.999970
iteration 1400 / 1500: loss 8.999966
iteration 0 / 1500: loss 8447.995592
iteration 100 / 1500: loss 9.000003
iteration 200 / 1500: loss 8.999997
iteration 300 / 1500: loss 8.999996
iteration 400 / 1500: loss 8.999996
iteration 500 / 1500: loss 8.999997
iteration 600 / 1500: loss 8.999997
iteration 700 / 1500: loss 8.999996
iteration 800 / 1500: loss 8.999996
iteration 900 / 1500: loss 8.999996
iteration 1000 / 1500: loss 8.999997
iteration 1100 / 1500: loss 8.999997
iteration 1200 / 1500: loss 8.999996
iteration 1300 / 1500: loss 8.999996
iteration 1400 / 1500: loss 8.999996
iteration 0 / 1500: loss 82.758030
iteration 100 / 1500: loss 18.885288
iteration 200 / 1500: loss 10.324761
iteration 300 / 1500: loss 9.176976
iteration 400 / 1500: loss 9.023430
iteration 500 / 1500: loss 9.002766
iteration 600 / 1500: loss 9.000129
iteration 700 / 1500: loss 8.999752
iteration 800 / 1500: loss 8.999676
iteration 900 / 1500: loss 8.999694
iteration 1000 / 1500: loss 8.999648
iteration 1100 / 1500: loss 8.999646
iteration 1200 / 1500: loss 8.999652
iteration 1300 / 1500: loss 8.999692
iteration 1400 / 1500: loss 8.999693
iteration 0 / 1500: loss 773.936101
iteration 100 / 1500: loss 8.999960
iteration 200 / 1500: loss 8.999968
iteration 300 / 1500: loss 8.999969
iteration 400 / 1500: loss 8.999966
iteration 500 / 1500: loss 8.999967
iteration 600 / 1500: loss 8.999969
iteration 700 / 1500: loss 8.999968
iteration 800 / 1500: loss 8.999963
iteration 900 / 1500: loss 8.999975
iteration 1000 / 1500: loss 8.999967
iteration 1100 / 1500: loss 8.999970
iteration 1200 / 1500: loss 8.999971
iteration 1300 / 1500: loss 8.999968
iteration 1400 / 1500: loss 8.999975
iteration 0 / 1500: loss 7398.233211
iteration 100 / 1500: loss 8.999999
iteration 200 / 1500: loss 9.000000
iteration 300 / 1500: loss 9.000000
iteration 400 / 1500: loss 9.000001
iteration 500 / 1500: loss 9.000001
iteration 600 / 1500: loss 8.999999
iteration 700 / 1500: loss 9.000002
iteration 800 / 1500: loss 9.000000
iteration 900 / 1500: loss 9.000000
iteration 1000 / 1500: loss 9.000000
iteration 1100 / 1500: loss 9.000000
iteration 1200 / 1500: loss 8.999999
iteration 1300 / 1500: loss 8.999999
iteration 1400 / 1500: loss 9.000000
lr 1.000000e-09 reg 5.000000e+04 train accuracy: 0.095020 val accuracy: 0.099000
lr 1.000000e-09 reg 5.000000e+05 train accuracy: 0.128694 val accuracy: 0.121000
lr 1.000000e-09 reg 5.000000e+06 train accuracy: 0.413959 val accuracy: 0.418000
lr 1.000000e-08 reg 5.000000e+04 train accuracy: 0.097837 val accuracy: 0.105000
lr 1.000000e-08 reg 5.000000e+05 train accuracy: 0.414837 val accuracy: 0.423000
lr 1.000000e-08 reg 5.000000e+06 train accuracy: 0.420306 val accuracy: 0.424000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.413918 val accuracy: 0.410000
lr 1.000000e-07 reg 5.000000e+05 train accuracy: 0.391122 val accuracy: 0.405000
lr 1.000000e-07 reg 5.000000e+06 train accuracy: 0.328816 val accuracy: 0.312000
best validation accuracy achieved: 0.424000
```

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

```
0.427
```

[7]:
```python
# An important way to gain intuition about how an algorithm works is to
# visualize the mistakes that it makes. In this visualization, we show examples
# of images that are misclassified by our current system. The first column
# shows images that our system labeled as "plane" but whose true label is
# something other than "plane".

examples_per_class = 8
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
↪'ship', 'truck']
for cls, cls_name in enumerate(classes):
    idxs = np.where((y_test != cls) & (y_test_pred == cls))[0]
    idxs = np.random.choice(idxs, examples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt.subplot(examples_per_class, len(classes), i * len(classes) + cls +␣
↪1)
        plt.imshow(X_test[idx].astype('uint8'))
        plt.axis('off')
        if i == 0:
            plt.title(cls_name)
plt.show()
```

### 1.3.1 Inline question 1:

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

*Your Answer :* Yes, I think misclassification which has occured when SVM has been applied over the HOG features makes sense because while extracting features or historgrams, we do not take the context into consideration, such as there are many features which can cause the classifier to mislabel the testing data such as similar backgrounds or textures or more common features such as wheels between truck and car or ears between cat and dog which might have led to such poor classification.

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

print(X_train_feats.shape)
```

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

[9]:
```python
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)*****

results = {}
best_val = -1
best_net = None

# only the Data is changed here
data = {
    'X_train': X_train_feats,
    'X_val': X_val_feats,
    'X_test': X_test_feats,
    'y_train': y_train,
    'y_val': y_val,
    'y_test': y_test
}



learning_rates=[0.0003, 0.003,0.03,0.0002]
```

```
regularization_strengths=[1e-05, 1e-04, 1e-03]


for l_r in (learning_rates):
  for r_s in regularization_strengths:
    #Instance of TwoLayerNet
    model =TwoLayerNet(input_dim, hidden_dim, num_classes,reg=r_s)
    #Instance of solver
    solver = Solver(model, data,
                    update_rule='sgd',
                    optim_config={
                      'learning_rate':l_r,
                    },
                    lr_decay=0.95,
                    num_epochs=20, batch_size=100,
                    print_every=100,verbose=False)
    solver.train()

    # using solver api to find best validation accuracy and stoeing in the␣
 ↪results
    #dictionary
    results[(l_r,r_s)]=solver.best_val_acc
    #always compares and sets the best acuracy.
    if best_val<results[(l_r,r_s)]:
      best_val=results[(l_r,r_s)]
      best_net=model

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

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

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

```
lr 2.000000e-04 reg : 0.000010 val accuracy: 0.239000
lr 2.000000e-04 reg : 0.000100 val accuracy: 0.222000
lr 2.000000e-04 reg : 0.001000 val accuracy: 0.171000
lr 3.000000e-04 reg : 0.000010 val accuracy: 0.269000
lr 3.000000e-04 reg : 0.000100 val accuracy: 0.252000
lr 3.000000e-04 reg : 0.001000 val accuracy: 0.251000
lr 3.000000e-03 reg : 0.000010 val accuracy: 0.341000
lr 3.000000e-03 reg : 0.000100 val accuracy: 0.334000
lr 3.000000e-03 reg : 0.001000 val accuracy: 0.344000
lr 3.000000e-02 reg : 0.000010 val accuracy: 0.561000
lr 3.000000e-02 reg : 0.000100 val accuracy: 0.564000
lr 3.000000e-02 reg : 0.001000 val accuracy: 0.566000
best validation accuracy achieved during cross-validation: 0.566000
```

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

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

```
0.543
```